An Instruction-Driven Adaptive Clock Management Through Dynamic Phase Scaling and Compiler Assistance for a Low Power Microprocessor

Tianyu Jia, *Student Member, IEEE,* Russ Joseph, *Member, IEEE,* and Jie Gu, *Senior Member, IEEE*

***Abstract*— This paper presents an instruction-driven adaptive clock management scheme using a dynamic phase scaling (DPS) operation and compiler-assisted cross-layer design methodology for a low power microprocessor. The intrinsic instruction-level timing variation is explored on an ARMv7 ISA pipeline architecture. The clock period can be dynamically adjusted by a multi-phase all-digital PLL, with the timing encoded into the instruction set at the compiler level. Special compiler optimization schemes are also presented through reorganizing the runtime instruction sequences to better exploit the dynamic timing slack. In addition, an instruction timing calibration scheme is proposed to characterize the instruction delay under process, voltage, and temperature (PVT) variations, which can be integrated with the conventional dynamic voltage and frequency scaling (DVFS). The implementation of 55-nm CMOS process shows a 20% performance improvement from the proposed instruction-driven adaptive clock management. The performance improvement can be equivalently converted up to 32% energy saving benefit.**

***Index Terms*— Adaptive clock, all-digital PLL (ADPLL), cross-layer design, dynamic timing slack (DTS), phase scaling operation.**

I. Introduction

THE increasing prominence of the low-power computing applications, such as IoT or wearable devices, requires a renewed focus on the system-level energy management. The conventional dynamic voltage and frequency scaling (DVFS) has become a widely utilized power management scheme to scale down the core voltage and frequency to perform program-level power optimization [1], [2]. As reported in [1], there is a total of 48 voltage domains, with each regulated by an on-chip voltage regulator to provide sufficient flexibility on the power management for DVFS. Although the prior schemes achieve significant energy saving, the operation speed of conventional DVFS is bounded at the program level. In other words, the longest exercised critical path delay within the whole program determines the minimum voltage $V_{\min}$ that can be scaled to for each voltage domain.

Manuscript received December 22, 2018; revised March 9, 2019; accepted April 7, 2019. This paper was approved by Associate Editor Vivek De. This paper was supported in part by the National Science Foundation under Grants CCF-1533656 and CCF-1618065. *(Corresponding author: Tianyu Jia.)*

The authors are with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60208 USA (e-mail: tianyujia2015@u.northwestern.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSSC.2019.2912510

In the previous work, the minimum safety voltage $V_{\min}$ for different application programs have been studied for various processor chips, such as an ultra-low-power MCU chip [3] or an ARMv8 CPU chip [4]. These studies observed that the $V_{\min}$ value varies significantly across different benchmark programs. Similar program-dependent $V_{\min}$ behavior was also observed in GPUs [5]. These prior studies show the opportunities to exploit the program-level DVFS for extra energy saving. However, the granularity of the previous DVFS schemes was at the program level and cannot fully exploit the critical path timing slack at the finer-grained cycle-by-cycle instruction level. In fact, it has been observed that the critical paths are not always exercised during real-time execution, leading to the dynamic timing slack (DTS) existed within every instruction cycle [6]. The error detection and correction (EDAC) techniques have been widely explored to detect timing errors in real-time and coupled with DVFS schemes to virtually eliminate the timing slack margins for fast-dynamic and local variation [7]–[11]. The *in-situ* EDAC circuits can monitor the timing violations of critical paths and can react the operation failure by pipeline replay. The error detection circuits can be realized by a double-sampling mechanism using a flip-flop and a latch, tunable replica circuits (TRCs), or the Razor techniques.

Although the EDAC techniques achieve notable benefits by removing the timing margins, they are relying on circuit-level timing speculation with allowing timing violation happened at logic critical paths. In fact, there is significant deterministic timing slack depending on the variation of runtime instructions. In this paper, an instruction-driven adaptive clock management scheme which operates at instruction level is presented to exploit the runtime slack variation. The proposed scheme can be used for the non-speculation processor, i.e., no EDAC required. By identifying the DTS for different instructions, the clock period can be dynamically adjusted cycle-by-cycle based on the runtime instructions with no timing violation, which leads to significant performance improvement benefit beyond the conventional program-level clock management. The proposed design methodology also complies with conventional design flow.

Previously, the adaptive clock techniques have been explored to provide the timing adjustment under the events of voltage supply droop with response time tens of nanoseconds

[12], [13]. To exploit the deterministic timing slack based on runtime instructions, the clock period is required to dynamically adjust cycle-by-cycle. In [14], a look-up-table-based cycle-by-cycle adaptive clock technique was presented for a simple open-source pipeline design with most critical paths located inside the EX stage. However, in the more complicated pipeline architecture, such as the ARMv7 ISA core used in this paper, most critical paths are located across pipeline stages and heavily depend on the in-flight instruction sequences. As a result, the scheme in [14] cannot handle a more complex microprocessor design. Besides, the use of a look-up table leads to the additional area and power overhead.

To exploit the intrinsic instruction-level timing slack, a dynamic phase scaling (DPS) operation scheme is proposed, where the clock period is modulated in real-time by instructions being processed inside the processor pipeline [15]. As the extension from the previous preliminary results [15], this paper presents the complete design vision and the approach for the instruction-driven clock management techniques. The contributions are summarized as follows.

1) In this paper, we present a complete design view of exploiting instruction-level DTS for a 32-bit six-stage ARMv7 in-order pipeline core. A cross-layer design methodology through the compiler, architecture, and circuit is demonstrated in a 55-nm test chip.
2) A zero-overhead timing encoding strategy is proposed to encode the timing control into the instruction set. The utilization of compiler for clock management enables a new end-to-end clock design paradigm.
3) Compiler optimization strategies are proposed for the first time to optimize the runtime instruction sequences and gain additional performance benefit. This also realizes our vision of cross-layer software/hardware co-optimization.
4) A low-overhead instruction timing calibration technique is proposed to capture the process, voltage, and temperature (PVT) variation impacts. The dynamic timing of each runtime instruction can be calibrated to further improve the performance benefit. The calibration scheme can be integrated with the conventional DVFS to compensate for real-time PVT variations.

The rest of this paper is organized as follows. The instruction-level timing slack and cross-layer design flow is introduced in Section II. Section III presents the overall scheme of the proposed instruction-driven clock management and the details of DPS operation. Compiler-level assistance including timing encoding and optimizations is illustrated in Section IV. Section V introduces the proposed timing calibration scheme. The measurement results obtained from the test chip are shown and discussed in Section VI, followed by the conclusion in Section VII.

II. Timing Slack at Instruction Level

A. Instruction-Level Dynamic Timing Slack

The conventional static timing analysis (STA) evaluates the worst-case critical path inside each processor pipeline stage and sets the critical path delay as the timing bound of the operation (i.e., clock period). However, the DTS exists during the program runtime when the critical paths are not exercised. In this paper, we explore the DTS in our test vehicle, a 32-bit six-stage in-order pipeline architecture using ARMv7 ISA, with the pipeline stages instruction fetch (IF), instruction decode (ID), operand fetch (OF), execution (EX), memory access (MEM), and write back (WB).

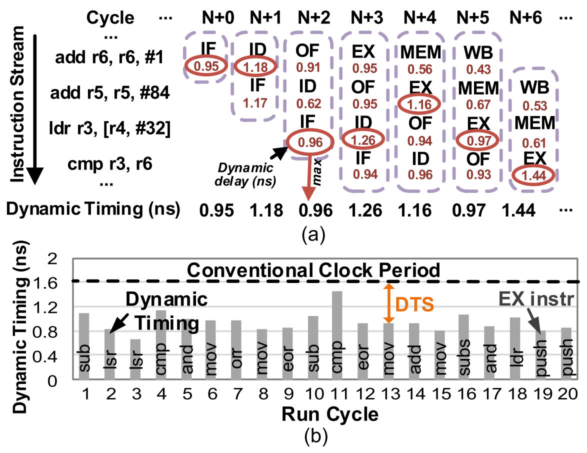


Fig. 1. Example of (a) dynamic delay simulation for six-stage pipeline and (b) runtime DTS in comparison with the conventional clock period based on STA.

A cross-layer simulation environment is built to perform dynamic timing analysis for the runtime instruction traces. The architecture simulator GEM5 [16] is configured to run simultaneously with the gate-level simulation to correlate software instruction with the gate-level delay information at cycle-by-cycle basis [17]. In the gate-level simulation, the inputs of all the pipeline registers, i.e., D-flip-flop, are tracked for the latest transition within every clock cycle to identify the exercised critical paths delay, as shown in Fig. 1(a). For every clock cycle, the longest dynamic delay across all the pipeline stages is extracted and identified as the minimum required clock period, i.e., dynamic timing. Fig. 1(b) shows the cycle-by-cycle gate-level dynamic timing for a sequence of instructions based on the cross-layer environment above. The SPEC CPU2006 benchmark program 403.gcc is used for demonstration in this example [18]. The gate-level timing analysis was performed on the final backend design of the processor core using a 55-nm CMOS process. As can be seen, a different DTS is observed within every clock cycle inside the pipeline. The excitation of the critical path only appears in small fraction cycles, e.g., triggered by 5%–10% of instructions.

Fig. 2(a) shows the STA results in the logic heavy pipeline stages (we exclude less timing-critical MEM and WB stages for simplicity). Industry EDA tools were utilized for the front-end synthesis and backend place and route design flow. The sign off frequency of the baseline microprocessor core is 625 MHz (1.6 ns). As shown by the STA, every pipeline stage contributes to the critical paths, which indicates that the pipeline has balanced critical paths across all stages (IF: 26%, ID: 19%, OF: 19%, and EX: 35%). Approximately 74% of delay paths from STA are fairly long (longer than 1.2 ns or 75% of critical path delay). Further

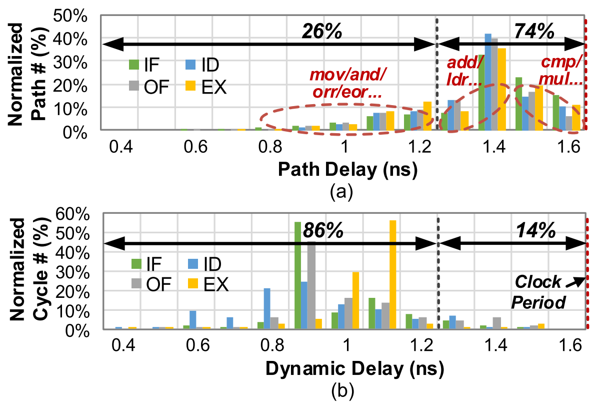


Fig. 2. Timing analysis histogram of the pipeline. (a) STA analysis highlighting the difference of timing distribution among instructions. (b) Cycle-by-cycle dynamic timing from gate-level simulation, only a small amount of cycles observed with long delay EX.

synthesis and place and route with tighter timing constraint does not improve the frequency of the design but incurs significant area increase. Hence, this design has been pushed to its best performance.

Fig. 2(b) shows the histogram for cycle-by-cycle gate-level simulation in the processor. Although STA reports 74% of logic path delays are relatively long, gate-level dynamic timing simulation shows only 14% of runtime exercises these long delay paths. The remainder clock cycles only exercise short delay paths. This is because many instructions are associated with short delay operations, such as instruction mov. Fig. 2(a) shows the distribution of a few selected instructions. Some instructions have fundamentally longer delay than others, such as cmp is significantly slower than mov, which is because the latter performs a much simpler arithmetic operation in the EX stage. Therefore, the attempts to equalize the instruction-level timing are unlikely to change this result as the timing variation among instructions is often due to their intrinsic difference in logic complexity.

B. Architecture-Level Study of DTS

To fully understand the root reasons behind the observed long delay instructions, we scrutinized critical paths inside the backend netlist design of the processor. Each instruction with long dynamic delay is mapped into gate-level netlist and layout. It is interesting to observe that only the long delay instructions at the EX stage are opcode dependent. The instruction delays in the remaining stages are highly sequence dependent. Fig. 3 shows several representative critical paths after scrutinizing final backend design. MEM and WB stages are excluded as instruction delays are fairly short within these two stages. The correlations between the critical paths and the dynamic delay are explained in the following.

1) Instruction Fetch: This stage normally only performs the program counter (PC) “+4” operation leading to a relatively short delay. However, under the branch conditions, a PC update is issued from the EX unit within the same cycle. As a result, any branch operation entering EX stage will trigger a long delay path ended at the IF stage. A similar situation is observed whenever PC is specially updated, such as the PC

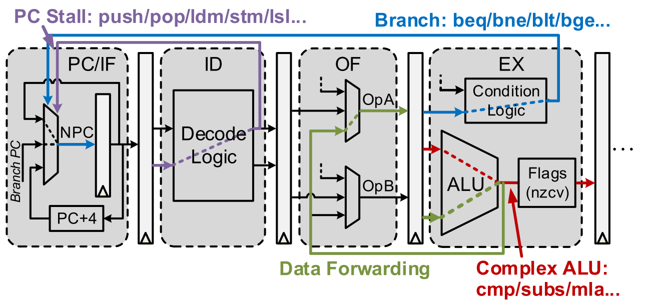


Fig. 3. Several representative critical paths mapped by the long delay instructions in the pipeline design.

stall cases for special instructions like push and pop. The PC stall instructions will exercise the long delay paths initiated from the ID stage.

2) Instruction Decode: This stage only has a very small amount of critical paths, which are highly related to the runtime sequence of instruction EXs. The root cause is the specific instruction sequences exercise the value transitions of particular pipeline registers, which will trigger the critical path toggling. Therefore, the dynamic delay of the ID stage is determined by the successive runtime instructions.

3) Operand Fetch: The long dynamic delays in OF stage are mainly caused by the instructions requiring operand forwarding operations. For example, the instruction adds $r1$, $r2$, and $r3$ followed by sub $r2$, $r1$, and $r5$ where the operand of register $r1$ needs to be forwarded from the prior EX stage. Such data dependence instruction sequences form the read-after-write (RAW) operation and normally trigger the long delay paths.

4) Execution Stage: The long dynamic delays in the EX stage are caused by special instruction types, such as cmp and subs which require some additional efforts to set up conditional flags. Multiplication will also trigger long delay paths due to complex computation. Instructions such as mov, ldr, str, and XOR will not require costly ALU operation and thus only incur a short delay. In addition, for the branch scenario, potential instruction sequences formed from the branch being taken or not taken may also cause longer dynamic delays.

It is worth to mention that the interfaces between the pipeline core and the caches have also been considered during the timing analysis. The cache (SRAM) access time needs to be taken into account for critical path delay calculation. For the data cache, it is only accessed when the data need to be read/write in the SRAM by load/store instructions. For the instruction cache, the data need to be fetched every cycle into the IF stage, except the PC stall cases. Different SRAM sizes lead to various access time and impact the dynamic timing of the pipeline core. However, for our targeted low power microprocessor, the clock period is relatively longer than the SRAM access time. In addition, conservative timing constraints are added during the synthesis flow to guarantee the interface setup/hold timing. Therefore, the critical paths inside the pipeline dominate the longest dynamic timing during the workload runtime.

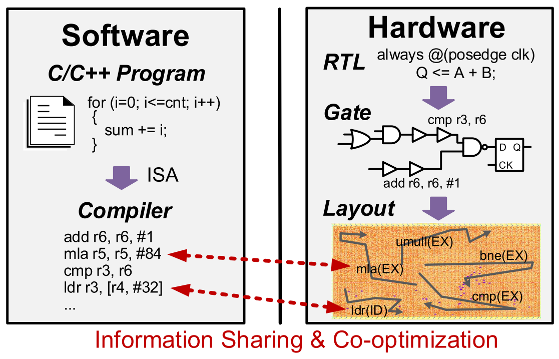


Fig. 4. High-level view of cross-layer hardware and software co-design to enable end-to-end exploitation of the instruction-level timing slack.

C. Exploiting Instruction-Level DTS by Cross-Layer Design

As the localized design optimizations at any single level of the system stack have been well-studied, cross-layer design and optimizations present interesting opportunities for improving the energy saving and performance. Fig. 4 presents a high-level overview of the developed approach. In contrast to the conventional design strategies which attempt to decouple many elements of software and hardware design, this paper intends to provide circuit-level dynamic timing information (hardware) to the compiler (software), which fully exploit properties of the hardware. The co-design method enables an end-to-end clock management paradigm, which utilizes the high-level compiler to manage and optimize processor adaptive clock and gain performance/energy benefit. The development at each design layer is summarized as below.

At EDA level, methodologies of cross-layer DTS analysis and generation of dynamic timing profiles are developed. This technique captures comprehensive instruction-level dynamic timing information and provides the baseline timing control for the instruction-level adaptive clock.

At the compiler and architecture levels, a zero-overhead instruction timing encoding strategy is developed as an extension to the ARMv7 ISA. The dynamic timing is encoded into every instruction and is used to guide the real-time cycle-by-cycle clock period adjustment. In addition, compiler optimization methods are proposed to optimize the runtime instruction sequence to gain additional performance benefit based on the dynamic timing observations from the hardware.

At the circuit level, the dynamic clock period adjustment is realized based on the developed multi-phase all-digital PLL (ADPLL). The clock circuit allows the clock period to be dynamically stretched or shrunk by 5%–40% of the clock period at every cycle using the proposed DPS operation. A special instruction timing calibration technique is developed to track the timing variations during runtime and provide timing compensation margin.

III. Instruction-Driven Clock Management

A. System Scheme

Fig. 5 shows the overall implementation of the proposed instruction-driven clock management scheme. The dynamic clock period for every cycle is determined from the gate-level

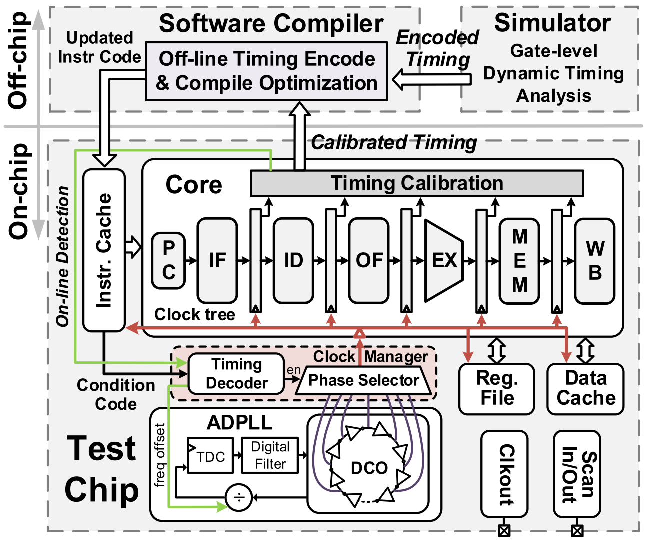


Fig. 5. Overview of the proposed system scheme implementation.

dynamic timing simulation and encoded into each instruction code by the software compiler during compile time. In the test chip, a 32-bit six-stage in-order pipeline design using ARMv7 ISA is implemented. A multi-phase ADPLL is designed to support the DPS operation and provide the adaptive clock period for all modules. Timing closure was carefully performed across multiple modules, e.g., pipeline, caches, register file, and PLL to guarantee no timing violation under the aggressive clock adjustment. During the processor operation, the instruction is fetched from instruction cache to the IF stage, with its condition code sent to the clock manager. The timing decoder decodes the condition code and issues phase selection for the equally spaced phases from PLL DCO.

Instead of only relying on the simulated instruction timing, an online instruction timing calibration scheme is implemented to characterize the on-chip instruction timing under PVT variations. In addition, the proposed timing calibration scheme can be embedded into the normal program instruction to detect the real-time timing variations and react by scaling down the ADPLL frequency, i.e., change the PLL divider ratio, as the conventional DVFS approach [7].

B. Multi-Phase ADPLL Implementation

To exploit the instruction-level timing slack, DPS operation is proposed. A multi-phase ADPLL is designed to support the DPS operation, as shown in Fig. 6(a). A multi-phase DCO is implemented by an 11-stage tri-state ring oscillator array, which is similar to [19] and [20], as shown in Fig. 6(b). The frequency of DCO is proportional to the drain current of the ring array and inversely proportional to the loading capacitance. There are 4-bit coarse tuning and 7-bit fine tuning to control active rings and loading capacitance, which achieves the coarse and fine resolution 30 and 0.1 MHz. The key feature of this multi-phase DCO is to provide 22 evenly delayed phases, i.e., delay step $t_{\text{step}} = T_{\text{PLL}}/22$, which will be used for the phase selection. Identical fine capacitance loads are distributed at each phase node and carefully matched. When the

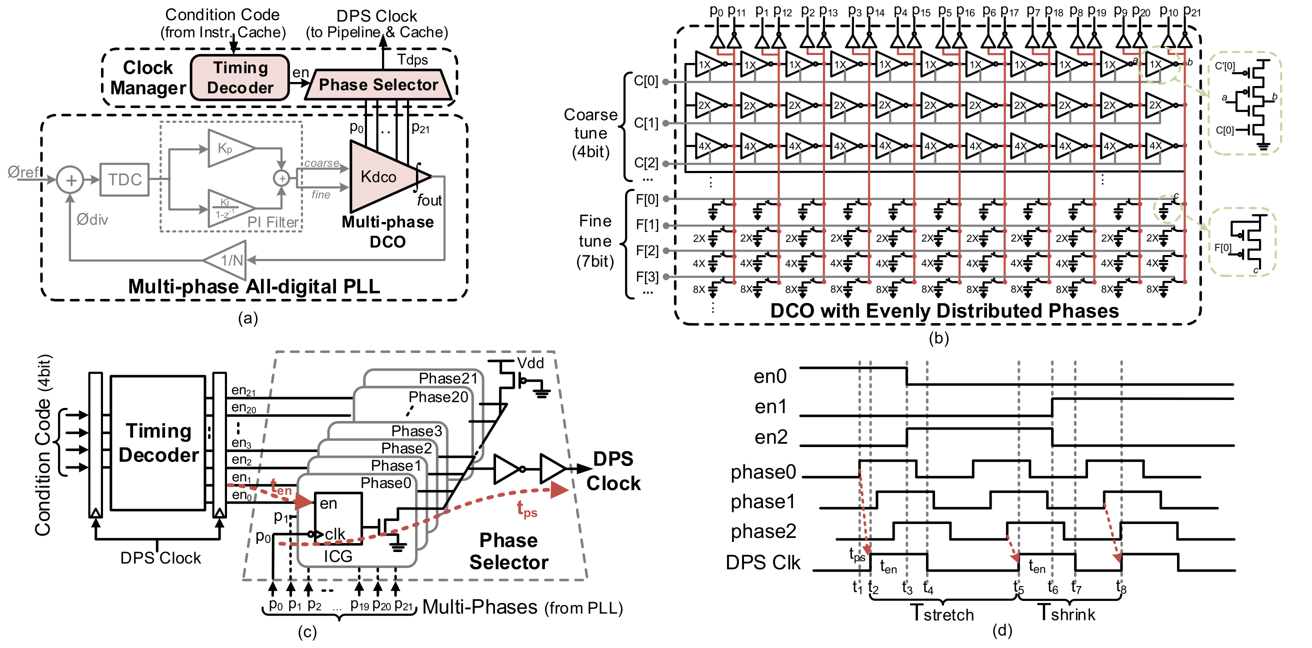


Fig. 6. (a) Top-level block diagram of the multi-phase ADPLL, detail schematic of (b) multi-phase DCO and (c) clock manager, and (d) waveforms of the DPS clock stretching and shrinking operations.

runtime instructions dynamically require a longer or shorter clock period, the phase selection is changed accordingly to generate $T_{\text{stretch}} = T_{\text{clk}} + n \times t_{\text{delay}}$ or $T_{\text{shrink}} = T_{\text{clk}} - n \times t_{\text{delay}}$, where n is determined by the encoded timing control. Note that the ADPLL only needs to lock to the target frequency at the beginning of the operation. During the microprocessor operation, the ADPLL remains in the locked frequency and only the selection of DCO phases is changed. Therefore, the clock period can be adjusted at the instruction level for every clock cycle without relocking the PLL loop.

The clock manager module including a timing decoder logic and a clock phase selector is utilized to dynamically select out one delayed phase and scale the output DPS clock period, as shown in Fig. 6(c). The phase selection information is encoded into the instruction condition code, which is fetched from the instruction cache and sent to the clock manager. The condition code is decoded by a timing decoder to generate the phase selection (enable) signal. Every delayed phase from the PLL DCO is connected to an integrated clock gating (ICG) cell to avoid clock glitch. The ICG cell for each clock phase is followed by a pull-down nMOS, with all connected to a shared pull-up pMOS [21]. The decoded enable signals will only select one ICG out of the total 22 ICG cells.

C. Dynamic Phase Scaling Operation

The detail DPS operation is shown by waveforms in Fig. 6(d). The output DPS clock can either stretch the clock period T_{PLL} by scaling to later phases, e.g., from phase 0 to phase 2, or shrink the clock period by scaling to earlier phases, e.g., from phase 2 to phase 1. At the beginning time t_1, the rising edge of phase 0 is selected to be the DPS output as the en0 signal is high. At time t_2, i.e., t_{ps} propagation delay after t_1, the rising edge of the phase 0 is propagated to the DPS output node. This DPS clock will trigger the timing decoder logic and pass the decoded phase enable signals for the current clock cycle to the ICG cells. At time t_3, the enable signal of the new selected phase, i.e., phase 2, is toggled to high. The ICG is a negative edge triggered cell, i.e., the selected phase will be updated at time t_4. Therefore, the rising edge of phase 2 will be selected as the following DPS rising edge at time t_5. The overall clock period is hence stretched to be longer than the constant clock period. Similarly, for shrinking the clock period, the DPS phase needs to be scaled to earlier clock phases.

There is a latency of two clock cycles required to allow the encoded timing to take effect and adjust the clock period. Within the first cycle, the instruction condition code is fetched and sent to the clock manager. At the second cycle, the clock manager decodes the condition code and guide the DPS phase selection. The dynamic clock period is then taken effect at the third cycle. As the microprocessor is in-order execution, the execution order can be predicted at the compiler level. Hence, the timing control is encoded two cycles earlier than its actual execution. The unpredicted operations such as the branch instructions are assigned the most conservative timing control, i.e., stretched to the longest clock period.

IV. Compiler Assistance for DTS

A. Encoding Timing Into Instruction Set

As the executed instruction of the processor is statically scheduled, the compiler is able to predict all instructions that occur in the pipeline and the upcoming instructions. The gate-level dynamic timing analysis is used to generate the dynamic timing profile by inspecting the sequence of operations. The compiler encodes the dynamic timing information into the instruction code to guide the real-time adaptive clock management. This method avoids increasing the size of the instruction

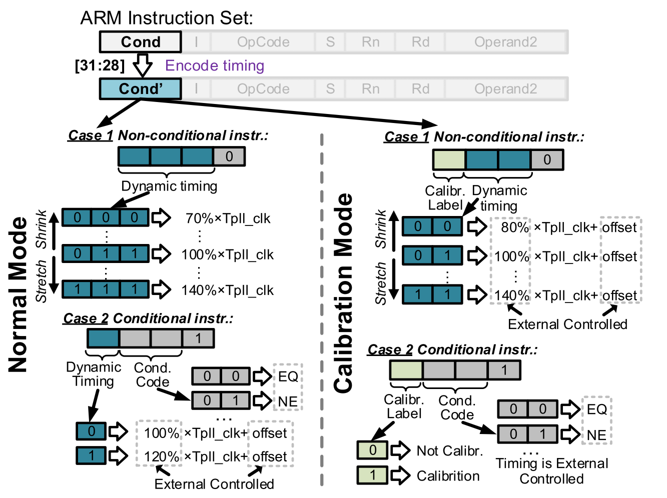
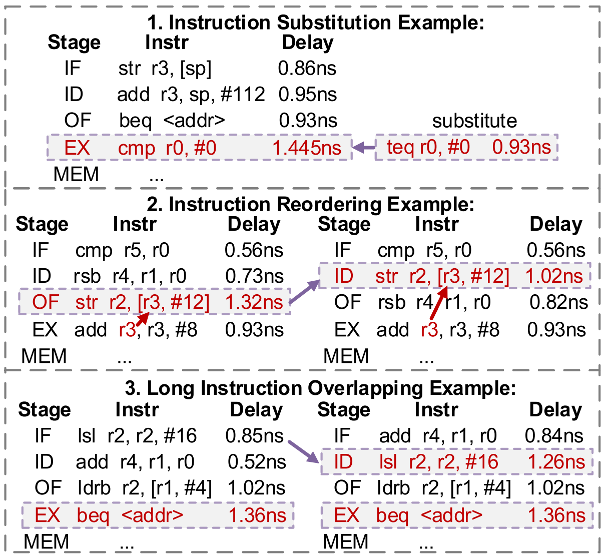


Fig. 7. Instruction timing encoding strategy within the modified instruction condition code field.


1. Instruction Substitution Example:

Stage	Instr	Delay		
IF	str r3, [sp]	0.86ns		
ID	add r3, sp, #112	0.95ns		
OF	beq <addr>	0.93ns	substitute	
EX	cmp r0, #0	1.445ns	teq r0, #0	0.93ns
MEM	...			

2. Instruction Reordering Example:

Stage	Instr	Delay	Stage	Instr	Delay
IF	cmp r5, r0	0.56ns	IF	cmp r5, r0	0.56ns
ID	rsb r4, r1, r0	0.73ns	ID	str r2, [r3, #12]	1.02ns
OF	str r2, [r3, #12]	1.32ns	OF	rsb r4, r1, r0	0.82ns
EX	add r3, r3, #8	0.93ns	EX	add r3, r3, #8	0.93ns
MEM	...		MEM	...	

3. Long Instruction Overlapping Example:

Stage	Instr	Delay	Stage	Instr	Delay
IF	lsl r2, r2, #16	0.85ns	IF	add r4, r1, r0	0.84ns
ID	add r4, r1, r0	0.52ns	ID	lsl r2, r2, #16	1.26ns
OF	ldrb r2, [r1, #4]	1.02ns	OF	ldrb r2, [r1, #4]	1.02ns
EX	beq <addr>	1.36ns	EX	beq <addr>	1.36ns
MEM	...		MEM	...	

Fig. 8. Examples of the compiler-level instruction optimization.

binary or dedicating additional hardware to store the clock control.

In the 32-bit ARMv7 ISA, the condition code takes on MSB bits [31:28] to represent 16 condition cases. However, in many programs, the majority of instructions will not utilize the condition codes. For the instructions utilizing condition codes, the condition cases like equal (EQ) and not equal (NE) are the most frequently used (∼75% of the cases). The rest condition cases are rarely utilized. Therefore, in this paper, the dynamic timing is encoded into the 4-bit condition code by remapping the condition code without increasing the instruction size.

As shown in Fig. 7, the LSB of condition code bit [28] is used as a mode selector to identify the condition code usage. If bit [28] == 0, the instruction will be executed unconditionally and bits [31:29] specify a 3-bit clock control. The encoded values allow the clock period T_{PLL} to be scaled in the range of -30% to $+40\%$ for unconditional instructions. If bit [28] == 1, i.e., indicating a conditional instruction, the instruction will be executed using bits [30:29] to specify the condition cases EQ/NE/GT/LE. Bit [31] is used to select the clock phases. For conditional instructions, a binary choice of scaling between 0% and 40% is implemented. Note the most stretched clock period $1.4T_{\mathrm{PLL}}$ is set equal to the conventional clock period $T_{\mathrm{clk,STA}}$ based on STA. Hence, the ADPLL is locked at 40% faster than the conventional operation mode with a maximum speed up of 70% compared with the conventional clock period.

To support the proposed timing calibration scheme, in the special timing calibration mode, the MSB bit [31] of condition code is utilized as a label for calibration instructions. The bit [31] == 1 represents the calibration cycle, in which a shorter clock period is provided to calibrate the minimum timing requirement. Bit [28] is still used as a mode selector as the normal operation mode. The bits [30:29] are utilized to carry dynamic timing information for non-conditional instructions. In special cases when condition codes are used in calibration mode, the phase selection is controlled by the external control signals. During the timing calibration, the target instructions or instruction sequences will be repetitively operated with shrunk dynamic clock period. The minimum clock period that guarantees instruction execution correctness is recorded on-chip and read out as the calibrated timing.

B. Compiler-Assisted Optimization

We also introduce the compiler optimization schemes to unlock additional gains from DTS. We leverage the dynamic timing analysis and derive a few simple but useful code transformations, which have no hardware overhead and are easy to implement in the compiler. We implemented three peephole optimizations: instruction substitution, instruction reordering, and instruction overlapping, as shown in Fig. 8. We implemented these optimization strategies in LLVM compiler [22] and evaluated their impacts in our test chip.

1) Instruction Substitution: The first optimization strategy identifies long delay instructions and replaces them with shorter delay instructions with the same semantics. This is based on the observation that some instructions with the same semantic show significant different delay time due to different hardware mechanism. For example, in ARMv7 ISA, the checking of "EQ to" relationship could be implemented using either cmp or teq instructions. They are equivalent semantically, while implementation-wise are quite different. teq sets the zero status register if two operands are equal, which can be simply realized by XOR gate, while cmp checks the relationship of greater, equal or less than between two values and generally requires subtraction using adders. As a result, teq can be operated much faster than cmp as no subtraction is involved. While there is no reward for such compiler optimization in the conventional processor, it provides a reward for our DTS exploitation.

2) Instruction Reordering: The second optimization strategy finds the instruction groups which can be re-sequenced. One usage example of this optimization is the data dependence case, where the results need to be forwarded from an instruction to its immediate successor, as described in Section II. Reordering instruction sequence will remove the adjacent data dependence and create some new timing slack. As compilers

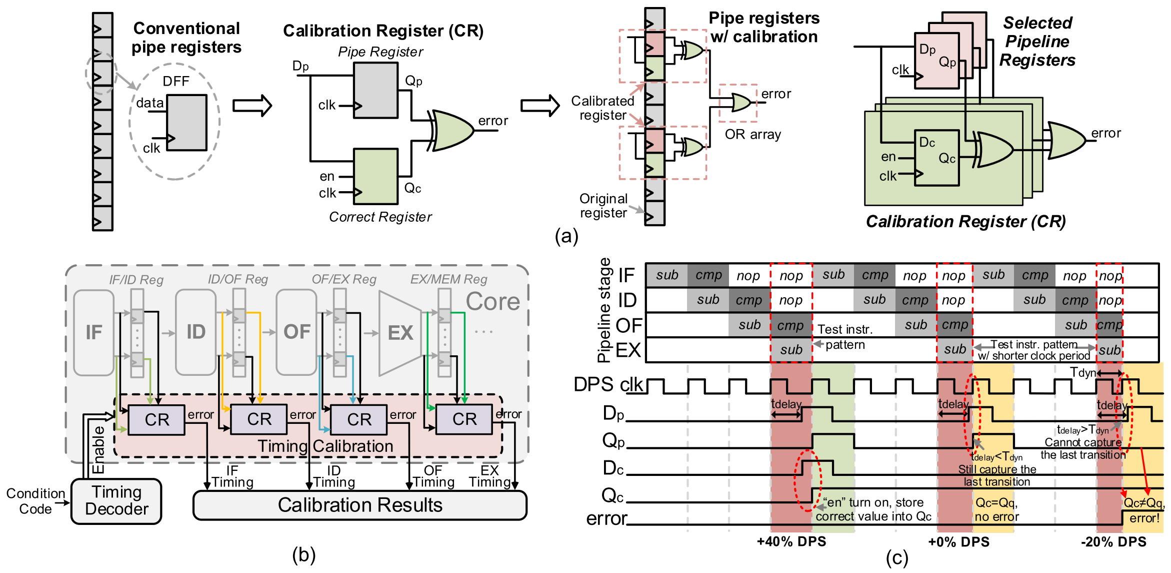


Fig. 9. (a) CR design for selected pipeline registers. (b) Implementation of the CRs inside the pipeline. (c) Operation waveforms of the calibration process.

typically apply re-orderings like this for other performance reasons, we are merely introducing a new application.

3) Instruction Overlapping: The third optimization strategy tries to overlap the occurrence of multiple critical path stressing operations within the same clock cycle. For example, if the critical paths are exercised at different pipeline stages in successive cycles, this optimization attempts to reschedule the instruction EX order so that the critical paths will be exercised simultaneously. This allows the compiler to bundle two or more clock stretching operations, i.e., long delay instructions, into a single clock cycle.

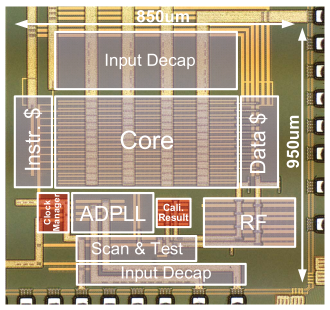
V. Instruction Timing Calibration

To deal with PVT variations, an instruction timing calibration scheme is proposed. As shown in Fig. 9(a), different from the conventional pipeline register, the calibration register (CR) is designed with one more correct register appended to the original register to store the correct operation value. During the timing calibration, the original pipeline register holds the register values at Q_P under different clock periods, while the correct register holds the correct EX values at Q_C using a conventional clock period. These two values are compared by the following XOR gate, which generates an error signal if two register values are different. The pipeline registers at the end of the critical paths are selected by STA method and replaced by the CR. As shown in Fig. 9(b), the critical paths at each pipeline stage are carefully analyzed during the timing analysis and can be calibrated by the CRs. During the core implementation, 336 flip-flops out of total 2120 flip-flops are replaced by the CRs, which leads about 3.8% area overhead of the microprocessor design. By leveraging the calibration flip-flops, about 24% critical paths inside the pipeline core can be calibrated.

For each instruction or instruction sequence to be calibrated, the test instruction patterns will be repetitively sent into the pipeline. As an example shown in Fig. 9(c), the instruction sequence sub(EX)–cmp(OF) is calibrated, which is signified by the calibration label, i.e., MSB of condition code = 1, for the execution cycle. All the rest in-flight instructions are replaced by no operation (nop) instruction to remove potential delay impact. The first-round execution of the test pattern is given a longest stretched clock period, i.e., +40% phase scaling, to avoid timing violation. As the longest clock period is assigned, the correct operation data are captured by the pipeline register at D_P and the CR D_C. As shown in the waveform, the CR holds the correct value at Q_C after the first-round execution. In the subsequent run, a shrunk clock period is given and the output of the pipeline registers will be compared with the stored correct results through XOR gates. In the example in Fig. 9(c), the DPS clock period is shrunk to +0% and −20% for the same instruction sequence execution. The +0% DPS clock period still can capture the last transition at D_P node, while −20% DPS clock period is too short to latch the proper register value leading the error signal appears. An auto testing program flow was built to automatically send multiple test instruction patterns into the chip for the timing calibration.

During the instruction timing calibration, the error signals generated by these CRs due to the timing violation will be sent and saved in a small on-chip register file and readout as the calibration results. Therefore, both the timing violation value and pipeline stage can be obtained during the off-chip analysis. The calibrated instruction timing will be further used as the encoded timing at the compiler level.

It is worth to mention again that the dynamic timing is mainly determined by the instruction sequences, e.g., the data dependence cases. Even for single instruction, its dynamic timing is also related to the instruction in the previous cycle. It is common that the activities on input pipe registers are constrained from its past state, i.e., the history of the previous instruction execution. Therefore, during the timing calibration, we selected several representative instruction sequences to calibrate the instruction timing. Each calibrated instruction sequence will be sent into the pipeline repetitively, with using



Implementation Details

Process	55nm CMOS
Core Power	106.8mW
Baseline Freq.	650MHz
Nominal Vdd	1.2V
Total Area	0.807mm^2
Instr. Cache	2KB SRAM
Data Cache	2KB SRAM
Register File	128B
Cali. Result	32B

Fig. 10. Die photograph and implementation details of the chip.

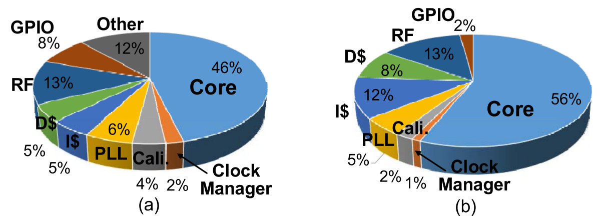


Fig. 11. Test chip (a) area and (b) power breakdown.

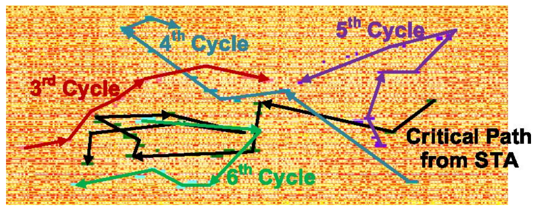


Fig. 12. Example of using dynamic delay simulation to show the cycle-by-cycle critical path in the layout.

"nop" instructions added in between, as shown in Fig. 9(c). Therefore, the exact same pipe register conditions can guarantee to exercise the same critical paths during each round of the timing calibration.

VI. Implementation and Measurement

A. Chip Implementation

The proposed microprocessor design is fabricated in a 55-nm low-power CMOS process, as the die photo and the implementation details shown in Fig. 10. The overall microprocessor area is about 0.807 mm^2. The instruction cache and data cache are implemented by commercial SRAM compiler. The baseline operation frequency is about 650 MHz, which is the measured maximum operation frequency using the conventional clocking scheme without timing violation. The internal cache/RF values were scanned out to evaluate the proper program execution.

Fig. 11 also shows the area and power breakdown for the chip implementation. The area overhead of the clock manager module, as the red square in the photo, is only 1.6%, which could be even smaller for a larger microprocessor design. The timing calibration scheme takes about 3.8% area, which mainly contributed by the storage memory for the calibration results. The power overhead for the clock manager and calibration are less than 1% and 2%, respectively.

Fig. 12 shows an example to show the difference between STA result and the runtime dynamic timing at the same

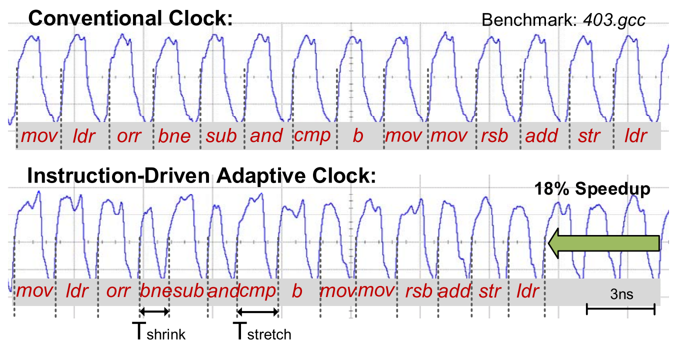


Fig. 13. Measured clock waveforms in the conventional normal operation mode and the instruction-driven adaptive clock mode for the same instructions.

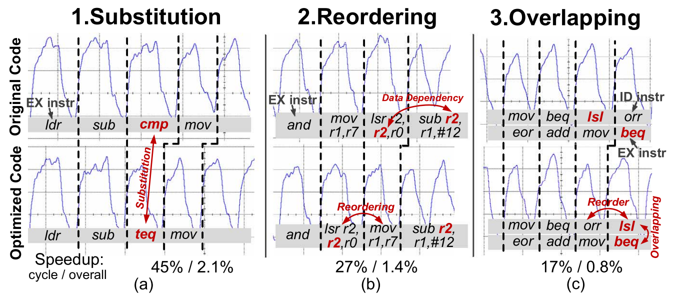


Fig. 14. Measured clock waveforms using the proposed compiler optimization techniques. (a) Instruction substitution. (b) Reordering. (c) Overlapping.

pipeline stage. When using the conventional STA method to report the longest delay, the STA pessimistically assume all the input start point can toggle the critical path and then find out the longest delay path. However, in real operation, the maximum dynamic delays vary significantly at each cycle due to different hardware root causes. The exercised critical paths, i.e., longest delay paths, for each cycle are highlighted for the example in Fig. 12. The layout view shows that the exercised critical path for each cycle is quite different, while they are all shorter than the critical path reported by the conventional STA method.

B. DPS Clock Measurement

The ADPLL clock generated can operate with either a constant clock period in a normal operation mode or dynamic clock period at the DTS exploitation mode. As shown in Fig. 13, during the normal operation mode, the processor was operating at the baseline operation frequency using conventional clock scheme. During the DTS exploitation mode, the clock period is guided by the encoded timing inside every runtime instruction and dynamically shrunk or stretched for short or long delay instructions. As an example shown in Fig. 13, the proposed instruction-driven adaptive clock can be successfully scaled based on the encoded dynamic timing information and achieve the performance improvement by about 18% for the same segment of instructions. The distortion in the clock measurement waveform is due to impedance mismatching on PCB traces and is not present inside the chip.

The proposed compiler assistance schemes for instruction sequence optimization were also verified on the test chip. The measured instruction substitution, reordering, and overlapping examples are shown in Fig. 14. By substituting a long delay instruction with a shorter one, up to 45% speedup can be

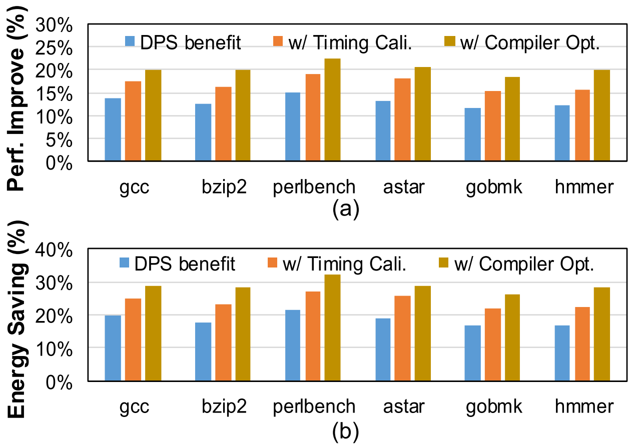


Fig. 15. (a) Measured performance improvement and (b) energy saving using six benchmark programs.

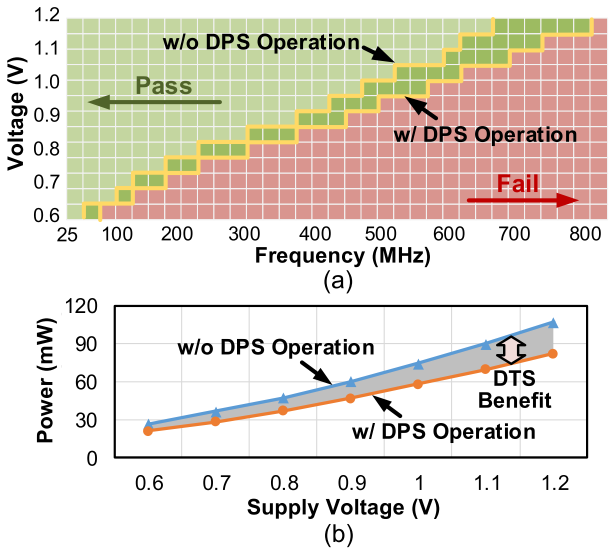


Fig. 16. (a) Measured Shmoo plot and (b) power consumption with scaled voltages with and without enabling the proposed DPS operation.

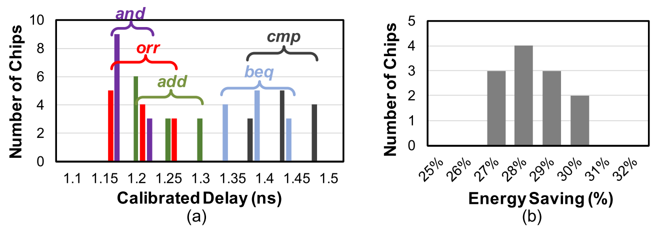


Fig. 17. (a) Measured instruction timing. (b) Power saving benefit across 12 test chips.

obtained within one single clock cycle. Overall, the compiler assistance provides more than 4% performance improvement over the entire program execution. This compiler optimization opportunity is unique to our DTS exploitation scheme and does not exist in conventional worst-case bounded pipeline design.

C. Performance Measurement

Fig. 15 shows the performance and energy benefits by leveraging the proposed DPS operation for different benchmark programs. Six benchmark programs from the SPEC CPU2006 benchmark suite were utilized during testing to represent different application categories [18]. All benchmarks were cross-compiled for the ARM architecture using LLVM compiler [22]. As shown in Fig. 15(a), the proposed instruction-driven adaptive clock scheme achieved about 14% performance improvement based on the simulated dynamic timing profile. The proposed timing calibration scheme was utilized to obtain more accurate instruction dynamic timing under process variation. In the experiment, the instruction timing calibration leads to additional performance improvement by 3%–5% or about 8% additional energy saving. With leveraging the compiler to optimize the runtime instruction sequences, the performance improvement is further increased by up to 4%. Overall, the proposed DPS scheme achieved up to 22% performance improvement with an average of 20% across different test programs. The proposed DPS scheme can also obtain energy saving benefit by scaling to a lower supply voltage. For the energy benefit measurement, we drop supply voltage while keeping the effective clock speed the same as conventional clock speed to maintain the same total execution time. The benchmark is executed under scaled voltage levels. The lowest voltage level with correct program execution is the minimum voltage level, and power is measured at this setting to obtain effective energy saving. In the experiment, an average of 28% and up to 32% energy saving was achieved by the proposed DPS operation scheme, as shown in Fig. 15(b).

Shmoo plot and measured power with the supply voltage scaling from 1.2 down to 0.6 V are shown in Fig. 16(a). Without DPS operation, the test chip can operate correctly up to 650 MHz at a supply voltage of 1.2 V. With enabling DPS operation, the effective operating frequency is improved to 780 MHz by 20%. When applying the scaled supply voltage, the instruction timing is calibrated at each voltage level, and the performance improvement is maintained. At 0.6 V, about 17% of performance improvement can be obtained. Fig. 16(b) shows the power consumption of the processor with scaled supply voltages. With enabling the DPS operation, about 28% and 24% power reduction can be achieved at nominal 1.2 V and low voltage 0.6 V, respectively.

D. Instruction Timing Calibration

The timing of individual instruction or instruction patterns is calibrated across 12 test chips with chip–chip variation. Fig. 17(a) shows some representative instruction timing calibration results. It is observed that the instruction timing only varies within a relatively small timing range. The simulated individual instruction delay at typical condition is compared with the measurement for a sequence of instructions, as shown in Fig. 18. A very small constant frequency offset was observed, most likely due to the test chip being fabricated at a slightly fast corner. Although there is a small variation between simulation and measurements on different chips, the overall timing profiles are very close. This shows the simulated dynamic timing profile can be used to generate the general timing profiling. Fig. 17(b) shows the measured energy saving benefit across chips using benchmark gcc. The proposed DPS

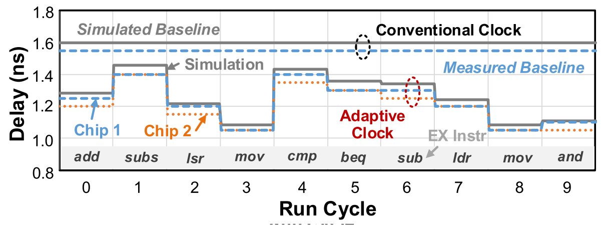


Fig. 18. Measured and simulated instruction delay in an instruction sequence of benchmark gcc.

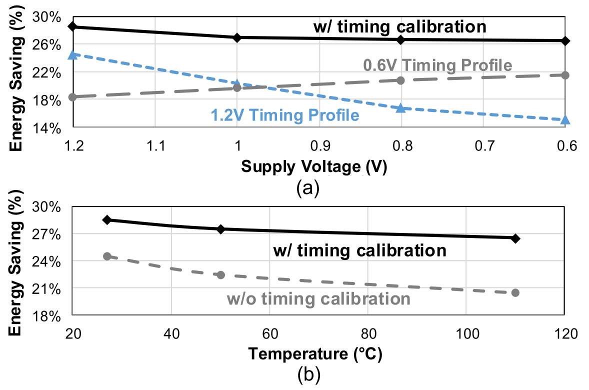


Fig. 19. Power saving versus (a) voltage scaling and (b) temperature change with and without timing calibration scheme.

operation maintains similar benefit, with at least 27% energy saving, across multiple chips.

The benefits of the calibration scheme under different supply voltage and temperature conditions were also tested and shown in Fig. 19. An interesting observation is that the benefit varies significantly based on which timing profile is used. The dynamic timing profile generated based on the timing library at 1.2 V has energy saving benefit dropped from 25% to 15% at 0.6 V due to different instruction timing characteristics at different supply voltages. On the other hand, the 0.6-V timing profile generated using low-voltage timing library was optimized for 0.6 V but does not produce the maximum benefits at 1.2 V. As a result, the proposed timing calibration solution can be used to calibrate and adjust the timing profile for large voltage ranges showing the optimal results at all conditions. As shown in Fig. 19, for both voltage and temperature changes, the timing calibration scheme remains energy saving above 26% from 0.6 to 1.2 V and from 27 °C to 110 °C. Note that only large supply voltage change creates a large variation to the instruction timings based on our measurement. For small variation, e.g., temperature or supply noise, a single timing profile is sufficient for use.

The proposed timing calibration can be integrated online with conventional DVFS operations. As an example shown in Fig. 20, the special calibration instructions are periodically embedding into program runtime to detect the real-time timing variations. Therefore, when executing programs, the instruction timing is periodically calibrated, and the error signal is generated to indicate whether there is a loss of timing margins. In our experiment, the calibration instructions will take 8 clock cycles to complete the timing calibration. These calibration

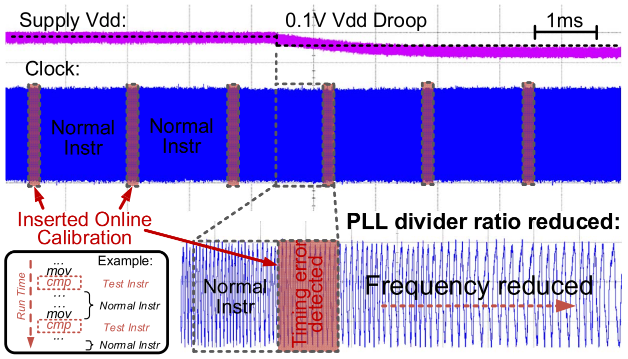


Fig. 20. Online timing calibration scheme integrated with the DVFS to adjust PLL frequency under noise droop event.

instructions have been inserted into the workload by every five thousand instructions, which leads negligible performance penalty. The proposed calibration scheme can be utilized to detect the slow real-time PVT variation. Whenever the timing violation is detected, the calibration error is sent to the clock manager guiding adjustment of the PLL divider ratio. As an example in Fig. 20, the PLL reacts to the supply noise droop by reducing the PLL divider ratio following the error message detected by calibration instructions.

E. Comparison and Discussion

1) Conventional DVFS [1]: As discussed earlier, the proposed technique is orthogonal to conventional DVFS which is constrained to low-speed program-level frequency adjustment and only works on the worst-case critical path of a design. The fine-grained instruction-driven adaptive clock management in this work can be applied to the top of existing DVFS to achieve additional saving.

2) EDAC/Razor Techniques [7]–[11]: Table. I listed the comparison between the proposed work and the prior EDAC techniques. EDAC/Razor technique is speculative and brings benefits of error tolerance that our proposed technique does not have. However, there are significant design overhead and challenges in EDAC techniques including new error-detection flip-flop/latch design, min-delay padding, exacerbated hold timing risks and special timing closure flow. In addition, the architecture needs to be modified to support the pipeline replay after the timing error detected. In our design, both digital front-end and back-end implementation are identical to the classic ASIC design. Hence, it is much easier to implement our technique compared with Razor. The proposed instruction-driven clock management technique is exploiting the deterministic timing slack difference between different instructions and does not require error detection and pipeline recovery. The proposed technique is also orthogonal to Razor technique as Razor can be implemented with the proposed scheme to achieve additional saving.

3) Droop-Based Adaptive Clock [12], [13]: The prior adaptive clocks schemes can detect the dynamic voltage droop and mitigate the voltage guard band by adaptive adjusting the operation frequency. The voltage variation monitor (DVM) design, i.e., TRCs, is required to monitor the voltage droop. In addition, the mismatch between the replica circuit and the

TABLE I
PERFORMANCE SUMMARY AND COMPARISON

	[7] 11'JSSC	[9] 08'ISSCC	[10] 13'ISSCC	[11] 16'ISSCC	[12] 15'ISSCC	[14] 16'ESSCIRC	This work
Process	45 nm	130 nm	45 nm	40 nm	16 nm	28 nm	55 nm
Architecture	LEON-3	Alpha	Alpha	Cortex-R4	MAC	OpenRISC	ARM CPU
Pipeline Stages	7	7	7	8	--	6	6
Frequency	1.45 GHz	185 MHz	1.2 GHz	843 MHz	2.5 GHz	1106 MHz	625 MHz
Power	135 mW	94.3 mW	Not reported	113 mW	250 mW	88 mW	107 mW
Timing Error Detection	EDS Latch	Razor II Latch	Razor-lite Flip-Flop	iRazor Latch	Replica DVM	Lookup Table	Compiler Encoding
Timing Error Management	Pipeline Replay	Pipeline Replay	Pipeline Replay	Clock Gating	Reduce to 1/2 Fclk	Adaptive Clock	DPS Adaptive Clock
Core Recovery	Yes	Yes	Yes	No	No	No	No
Special Latch/FF	Latch	Latch	Flip-Flop	Latch	--	--	Calibration FF
Modified FF # / Total FF #	12%	121/ 826	492/ 2482	1115/ 12875	--	--	336/ 2120
Energy Saving	22%	35%	45.4%	41%	13%	15%	28%
Area Overhead	3.8%	Not reported	4.42%	13.6%	4.4%	~7.3%	1.6% for DPS 3.8% for Cali.

pipeline critical paths may lead to pessimistic benefit and needs to be carefully calibrated [12]. Compared with the previous supply droop based adaptive clock, this paper focusing on exploiting the deterministic instruction based DTS.

4) Prior Instruction-Level DTS Technique [14]: The previous study is based on a simple open-source pipeline architecture, with most critical paths located inside either Fetch or EX stage. Therefore, a simple instruction based clock solution with a look table design is sufficient to exploit the instruction timing slack. However, in the more complicated pipeline architecture, most critical paths are located across pipeline stages and heavily depend on in-flight instruction sequences. The proposed timing encoding scheme which leveraging high-level compiler can efficiently deal with the instruction sequence-dependent clock management and does not incur look-up table overhead. In addition, the online calibration scheme can further characterize the instruction timing under PVT variations.

5) Prior Program-Level DTS Technique [3]: The architecture used in [3] was the openMSP430 processor, which is a simple 16-bit single cycle architecture. The achieved DTS benefits rely on no EX of the critical path in a complete program. In our analysis on ARM operation, a critical path is frequently executed in 10%–20% of the time rendering no benefit from the previous technique due to the slow DVFS speed. Hence, our proposed technique is more applicable to a general pipeline architecture.

6) Other Related Work: Increasingly, the cross-layer optimizations that couple modifications in architecture with corresponding changes in the circuit implementation and code generation, are applied to maximize the performance of the system. In Blueshift and DynaTune, timing speculation was built from the ground up coordinating circuit-level characteristics like gate-sizing and Vth assignment to control error rates and maximize the benefits of the architecture [23], [24]. Relax introduced a framework that supports software recovery of hardware faults including ISA extensions and compiler support [25]. Speculative dynamic timing exploitation was proposed where improvement on performance was achieved using Razor based technique [6]. The design that ties opcodes with critical path activity and allowing the system to operate in more aggressive modes was proposed, however, without further support from software compiler and required additional hardware to store operation information [14]. Compared with the above work, our work is the first demonstration of a full-layer implementation of the DTS exploitation microprocessor design with validation of real silicon operation and PVT variation tolerance. The cross-layer co-design methodology bridges the low-level hardware timing information and the high-level compiler leading to a new end-to-end design approach for software and hardware co-optimization.

VII. CONCLUSION

In this paper, an instruction driven DPS operation is presented to exploit the DTS of instructions on an ARMv7 microprocessor. By building a cross-layer design environment and utilizing the multi-phase ADPLL, the circuit-level delay information is delivered to the compiler to provide runtime timing control dynamically at the instruction level. In addition, an online instruction timing calibration scheme is used to characterize chip variations. The proposed instruction-driven clock management technique was implemented in a 55-nm test chip achieving 20% performance improvement or 28% energy saving benefit.

REFERENCES

[1] Z. Toprak-Deniz *et al.*, "Distributed system of digitally controlled microregulators enabling per-core DVFS for the POWER8 microprocessor," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2014, pp. 98–99.

[2] M. Cho *et al.*, "Postsilicon voltage guard-band reduction in a 22 nm graphics execution core using adaptive voltage scaling and dynamic power gating," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 50–63, Jan. 2017.

[3] H. Cherupalli, R. Kumar, and J. Sartori, "Exploiting dynamic timing slack for energy efficiency in ultra-low-power embedded systems," in *Proc. 43rd Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 671–681.

[4] G. Papadimitriou, M. Kaliorakis, A. Chatzidimitriou, D. Gizopoulos, P. Lawthers, and S. Das, "Harnessing voltage margins for energy efficiency in multicore CPUs," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2017, pp. 503–516.

[5] J. Leng, A. Buyuktosunoglu, R. Bertran, P. Bose, and V. J. Reddi, "Safe limits on voltage reduction efficiency in GPUs: A direct measurement approach," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2015, pp. 294–307.

[6] J. Xin and R. Joseph, "Identifying and predicting timing-critical instructions to boost timing speculation," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2011, pp. 128–139.

[7] K. A. Bowman *et al.*, "A 45 nm resilient microprocessor core for dynamic variation tolerance," *IEEE J. Solid-State Circuits*, vol. 46, no. 1, pp. 194–208, Jan. 2011.

[8] J. P. Kulkarni *et al.*, "A 409 GOPS/W adaptive and resilient domino register file in 22 nm tri-gate CMOS featuring *in-situ* timing margin and error detection for tolerance to within-die variation, voltage droop, temperature and aging," *IEEE J. Solid-State Circuits*, vol. 51, no. 1, pp. 117–129, Jan. 2016.

[9] D. Blaauw *et al.*, "Razor II: *In situ* error detection and correction for PVT and SER tolerance," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2008, pp. 400–622.

[10] S. Kim, I. Kwon, D. Fick, M. Kim, Y.-P. Chen, and D. Sylvester, "Razor-lite: A side-channel error-detection register for timing-margin recovery in 45 nm SOI CMOS," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2013, pp. 264–265.

[11] Y. Zhang *et al.*, "iRazor: 3-transistor current-based error detection and correction in an ARM Cortex-R4 processor," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Jan./Feb. 2016, pp. 160–162.

[12] K. Bowman *et al.*, "A 16 nm auto-calibrating dynamically adaptive clock distribution for maximizing supply-voltage-droop tolerance across a wide operating range," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2015, pp. 1–3.

[13] M. S. Floyd *et al.*, "Adaptive clocking in the POWER9 processor for voltage droop protection," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2017, pp. 444–445.

[14] J. Constantin, A. Bonetti, A. Teman, C. Müller, L. Schmid, and A. Burg, "DynOR: A 32-bit microprocessor in 28 nm FD-SOI with cycle-by-cycle dynamic clock adjustment," in *Proc. Eur. Solid-State Circuits Conf. (ESSCIRC)*, Sep. 2016, pp. 261–264.

[15] T. Jia, R. Joseph, and J. Gu, "An instruction driven adaptive clock phase scaling with timing encoding and online instruction calibration for a low power microprocessor," in *Proc. IEEE 44th Eur. Solid State Circuits Conf. (ESSCIRC)*, Sep. 2018, pp. 94–97.

[16] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, May 2011.

[17] T. Jia, R. Joseph, and J. Gu, "Greybox design methodology: A program driven hardware co-optimization with ultra-dynamic clock management," in *Proc. 54th Annu. Design Automat. Conf. (DAC)*, Jun. 2017, Art. no. 48.

[18] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[19] J. A. Tierno, A. V. Rylyakov, and D. J. Friedman, "A wide power supply range, wide tuning range, all static CMOS all digital PLL in 65 nm SOI," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 42–51, Jan. 2008.

[20] N. August, H.-J. Lee, M. Vandepas, and R. Parker, "A TDC-less ADPLL with 200-to-3200 MHz range and 3 mW power dissipation for mobile SoC clocking in 22 nm CMOS," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2012, pp. 246–248.

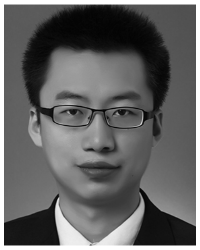
[21] M. Lee and A. A. Abidi, "A 9 b, 1.25 ps resolution coarse–fine time-to-digital converter in 90 nm CMOS that amplifies a time residue," *IEEE J. Solid-State Circuits*, vol. 43, no. 4, pp. 769–777, Apr. 2008.

[22] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim.*, Mar. 2004, p. 75.

[23] B. Greskamp *et al.*, "Blueshift: Designing processors for timing speculation from the ground up," in *Proc. IEEE 15th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2009, pp. 213–224.

[24] L. Wan and D. Chen, "DynaTune: Circuit-level optimization for timing speculation considering dynamic path behavior," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2009, pp. 172–179.

[25] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *Proc. 37th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2010, pp. 497–508.



Tianyu Jia (S'15) received the B.S. and M.S. degrees from the Beijing University of Posts and Telecommunications, Beijing, China, in 2011 and 2014, respectively. He is currently pursuing the Ph.D. degree in computer engineering with Northwestern University, Evanston, IL, USA.

He was interned at IBM Thomas J. Watson Research Center in 2018. His current research interests include the efficient power and clock management circuit design and the low-power microprocessor design.

Mr. Jia was awarded the National Graduate Scholarship of China in 2012.

Russ Joseph (M'04) received the B.S. degree in electrical and computer engineering, with an additional major in computer science, from Carnegie Mellon University, Pittsburgh, PA, USA, in 1999, and the Ph.D. degree in electrical engineering from Princeton University, Princeton, NJ, USA, in 2004.

He is currently an Associate Professor of electrical engineering and computer science with Northwestern University, Evanston, IL, USA. His research focuses on power-aware and reliability-aware computer architecture.

Jie Gu (SM'19) received the B.S. degree from Tsinghua University, Beijing, China, the M.S. degree from Texas A&M University, College Station, TX, USA, and the Ph.D. degree from the University of Minnesota, Minneapolis, MN, USA, in 2008.

From 2008 to 2010, he was an IC Design Engineer with Texas Instruments, Dallas, TX, USA, where he focused on ultra-low-voltage mobile processor design and integrated power management techniques. From 2011 to 2014, he was a Senior Staff Engineer with Maxlinear, Inc., Carlsbad, CA, USA, where he focused on low power mixed-signal broadband SoC design. He is currently an Assistant Professor with Northwestern University, Evanston, IL, USA. His research interests include ultra-dynamic clock and power management for microprocessor and accelerators, emerging mixed-signal computing circuit, and design of machine learning capable edge devices.

Dr. Gu has served as program committees and conference co-chairs for numerous low power design conference and journals, such as ISPLED, DAC, ICCAD, and ICCD.