Greybox Design Methodology: A Program Driven Hardware Co-optimization with Ultra-Dynamic Clock Management

Tianyu Jia, Russ Joseph, and Jie Gu

EECS Department, Northwestern University, Evanston, IL, USA
tianyujia2015@u.northwestern.edu, rjoseph@eecs.northwestern.edu, jgu@northwestern.edu

ABSTRACT

In this paper, a novel Greybox design methodology is proposed to establish a design and co-optimization flow across the boundary of conventional software and hardware design. The dynamic timing of each software instruction is simulated and associated with processor hardware design, which provides the basis of ultra-dynamic clock management. The proposed scheme effectively implements the instruction-based clock management and achieves 21.71% frequency speedup. Besides, a novel program-driven hardware optimization flow is proposed, in which software operations are mapped with hardware gate netlist and sorted by the usage frequency. The experiments on an ARM based pipeline design in commercial 65nm CMOS process show an extra 10% frequency speedup is obtained with high optimization efficiency. Overall, the proposed Greybox design method achieves frequency speedup by 31.56%, comparing with conventional design method.

1. INTRODUCTION

As CMOS technology scaling has slowed down significantly, innovative systematic approaches for low power design become crucial to solve the energy bottleneck of many emerging applications, such as wearable electronics, Internet-of-Things, and biomedical devices. Dynamic voltage frequency scaling (DVFS) has been widely utilized as one of the main approaches to achieve energy efficient computing. In recent years, fine grained DVFS which integrates multiple on-chip regulators for multi-core processors becomes popular and provides significant flexibility for energy optimization [1-2]. Beyond the system level scaling for each core, researchers started to explore the architecture and circuit level co-optimization based on sophisticated insight into software programs. For example, an autonomous DVFS scheme was proposed in which regulator achieves fast transient response to support the dynamic workload changing [3]. Recently, an instruction-based voltage scaling scheme was introduced by dynamically adapting supply voltage for each instruction in ultra-low power scenario [4].

Conventional static timing analysis (STA) for the synchronous digital circuit normally determines the operation frequency based on the worst critical logic paths to avoid any timing violations. In real applications, pronounced locality of the instruction execution time has been observed [5]. It has been found that dynamic timing slack exists at instruction level [6-7]. To remove the safety margin and speedup the clock frequency, hardware design like Razor is proposed to use error detection mechanisms and achieving a 10~30% power saving beyond conventional design techniques [8]. However, additional error detection logics are needed to recover the pipeline when the error is detected. An instruction based frequency scaling scheme utilizing dynamic timing slack was proposed in [6]. However, the hardware implication of dynamic timing slack is not explored, and there is a missing consideration of the link between software usage and actual hardware performance.

It is important to notice that in the conventional design flow, software and hardware design are normally performed by separate design groups and treat each other as a “blackbox”. For instance, the backend design team on gate level optimization of the microprocessor does not consider particular program/instruction usage during the hardware design. Similarly, engineers working on compiler optimization will not have the knowledge of gate level performance of each instruction set. As will be shown in the paper, if the software operation behavior of the target applications could be analyzed and incorporated into the hardware design phase, it can provide a new opportunity of cross-layer design optimization, which renders significant performance enhancement. In this paper, we proposed a novel HW/SW co-design methodology, “Greybox methodology”, which targets to break the boundary of conventional HW/SW design flow by creating a mapping between software instructions and processor hardware design. Fig. 1 shows an overview of the proposed “Greybox Design Methodology”. At hardware side, instruction-based ultra-dynamic clock management scheme based on a dynamic phase selecting all-digital phase-locked-loop (PLL) is engaged to enable a real-time collaboration between instructions and processor dynamic clock period control. At architecture and system side, the program level knowledge such as instruction usage frequency is incorporated into the design of microprocessor through a sophisticated mapping process. As a result, a novel program-driven hardware optimization flow is proposed to provide a new opportunity of hardware co-optimization which has not been considered in conventional design flow.

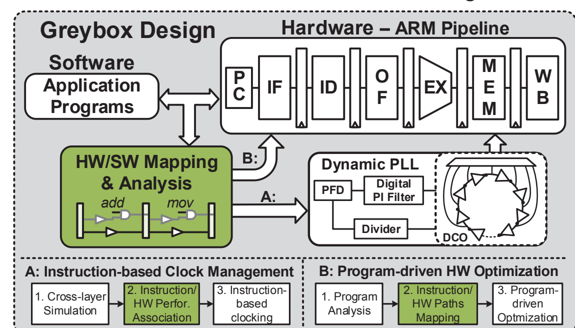

Fig. 1. Overview of the Greybox design methodology.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

DAC '17, June 18-22, 2017, Austin, TX, USA

© 2017 ACM. ISBN 978-1-4503-4927-7/17/06…$15.00

DOI: http://dx.doi.org/10.1145/3061639.3062255

This work uses an ARMv5 ISA microprocessor design to evaluate the performance of the proposed scheme [9]. In order to obtain the instruction dynamic timing information, a cross-layer simulation environment was developed, in which architecture level Gem5 simulator [10], gate level simulator VCS and timing analysis tool Primetime are integrated into a joint simulator in which each software program is simulated with backend annotated gate level netlist at every clock cycle with accurate timing information. In parallel, Cadence Analog Mixed-Signal (AMS) simulations is utilized for gate level pipeline netlist and PLL circuits to verify transistor level timing and clocking control. More detailed contribution of this work are highlighted below: (1) Based on the large amount of instruction level timing analysis, an association between software instructions and the hardware design netlist is created; (2) The timing distribution for each instruction operation is analyzed and the critical instructions are classified into particular instruction categories for creating the operation clock bound for dynamic clock management; (3) An instruction-based ultra-dynamic clock management scheme is developed and significant processor clock frequency speedup is achieved; (4) A novel program-driven hardware co-optimization flow is proposed, in which the critical paths mapped with frequently-used instruction and sorted based on a sophisticated mathematical weighting model. Optimization is then performed based on ranking of the instruction set weight, which renders significant improvement on efficiency of hardware optimization; (5) The whole hardware system including all-digital PLL are designed using commercial CMOS process with thorough verification down to the transistor level simulations.

2. INSTRUCTION-BASED ULTRA-DYNAMIC CLOCK MANAGEMENT

A single-issue ARMv5 microprocessor design is used as our test vehicle due to its popularity and relatively simple structure, as shown in Fig. 1. The pipeline design includes 6 pipeline stages: instruction fetch (IF), instruction decode (ID), operand fetch (OF), execution (EX), Memory (MEM) and write back (WB). Following the instruction set architecture (ISA) defined for ARMv5 architecture, the target pipeline is designed in commercial 65nm CMOS technology. The design has a nominal supply of 1.2V, with the conventional synthesized operating frequency 750MHz, which means the clock period $T_{clk} \approx 1.33ns$.

The design flow of the proposed instruction-based ultra-dynamic clock management as first part of overall Greybox design is shown in the left bottom (A) of Fig. 1. First, the cross-layer simulations which include both architecture and circuit level simulations are performed to obtain the instruction level dynamic timing information. Second, to facilitate the analysis and prediction of the instruction based hardware delay, software instructions are classified intro several instruction categories based on different hardware operation mechanism through thorough reading into the ISA definition and synthesized gate level netlist. Finally, an instruction-based clock management is created by assigning dynamic clock period to each individual instruction set.

2.1 Instruction Dynamic Timing

Fig. 2 shows several simulation examples of the instruction dynamic delay time distribution at different pipeline stages from program *403.gcc* in SPEC CPU2006 benchmark suite [11]. It reveals an important observation that instruction delay varies significantly depending on instruction type, pipeline stage and operand values. In conventional design, the worst-case delay is always used as final clock period T_{clk} regardless whether the worst-case delay is executed by actual software operations or not. In our proposed "Greybox" scheme, we create an ultra-dynamic clock design based on all-digital PLL (details in Section 4) to fully track the delay variation among instructions, so as to remove the extra pessimism from the large delay distribution of each instruction.

Among the simulated instruction dynamic timing results, we define the instruction dynamic timing as instruction *dynamic delay* T_d within each pipeline stage. Across the pipeline stages, we define *pipeline dynamic clock period* $T_{d,pipe}$ to be the longest instruction dynamic delay across all pipeline stages in the same cycle, as equation (1), which represent the minimum clock period to avoid timing violation.

$$T_{d,pipe} = \max\{T_{d,IF}, T_{d,ID}, T_{d,OF}, T_{d,EX}, T_{d,MEM}, T_{d,WB}\} \quad (1)$$

Note that the dynamic delay T_d varies even for the same instruction because the operand values vary. Based on the T_d distribution for each instruction, the maximum instruction dynamic delay is defined as *dynamic delay bound*, as (2), which denotes the worst execution time for the particular instruction.

$$T_{bound} = \max\{T_d\} \quad (2)$$

In Fig. 2, the dynamic delay of instruction *ldr* at EX stage *ldr*(EX) varies from 0.5ns to 1.1ns and thus $T_{bound} = 1.1ns$. Similarly, the T_{bound} for both *ldr*(OF) and *cmp*(EX) is 1.2ns and 1.3ns, respectively. The observations show that not only different instruction at the same pipeline stage may take different longest dynamic delay, e.g. T_{bound}, the same instruction at different stages also show different timing results. The efficient solution to better understand the T_{bound} of each instruction is to build the association between the instruction timing and hardware design, which is the HW/SW association process will be introduced in the next section.

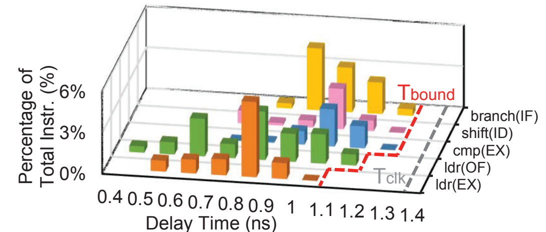

Fig. 2. Simulation examples of individual instruction dynamic delay distribution for benchmark *403.gcc*.

2.2 Instruction Timing Association with Hardware

The dynamic delay bound T_{bound} for each instruction is analyzed and associated with the hardware design in this section. Based on the PLL clock design, the dynamic clocking range is set from 0.8ns to T_{clk} with the clock adjustment step 0.1ns. Thus all the instructions with dynamic delay larger than 0.8ns are analyzed and associated with the corresponding T_{bound} by the following instruction categories. For instructions without T_d beyond 0.8ns, $T_{bound} = 0.8ns$ is assigned.

Category 1: Complex logic operations

The instructions in this category most happen at the execution (EX) stage due to the complex logic operation, such as instructions *cmp* (Compare), *mla* (Multiplication), *subs* (Subtraction with Compare), etc. For example, *cmp* at EX stage will exercise both ALU operation and computation of conditional flags, which will deterministically take longer time than regular ALU instructions. The instructions in this category contribute to the conventional critical path, i.e. worst case delay in the pipeline and determine the clock period T_{clk} in conventional processor design.

Category 2: Data dependency operations

Instructions which use values produced by the previous instructions are said to have read-after-write (RAW) dependencies. The data dependency will likely cause a long delay at OF stage, as the instruction will wait for the previous instruction operand results. In this scenario, the operand forwarding paths are triggered.

Category 3: PC fetch related operations

Program counter (PC) fetch and computation logic normally associated with other pipeline stages. There are mainly two types of instructions that could affect the delay time of the PC fetch logic. One type is the *branch* instructions located at EX stage, which evaluate the conditional flags and determine whether the branch will be taken or not. The other case is caused by some complex instructions, e.g. *shift*, *push*/*pop*, which require two or more clock cycles to complete based on the ISA. Those instructions introduce PC "stall" to prevent PC from incrementing, as shown in Fig. 3.

Category 4: Miscellaneous operations

There are still small portion instructions consume longer delay which has not been included in the categories above. For example, some specific instruction sequences could occasionally trigger some long operations paths by special register value transitions. The instructions in this category are highly related to specific architecture design and can be scrutinized based on the particular processor design rather than the ISA.

The delay time distribution of the categorized instructions is shown in the histogram of Fig. 3. In the distribution, instructions with longest delay time mostly come from Category 1 at EX stage. However, only 1.2% instructions really execute the worst case critical path close to T_{clk}. This observation points out the significant pessimism in conventional worst-case based design approach where clock is fixed at worst-case delay leading to significant redundancy in operating speed. The proposed dynamic clocking scheme exploits such redundancy margin to achieve significant enhancement on processor speed.

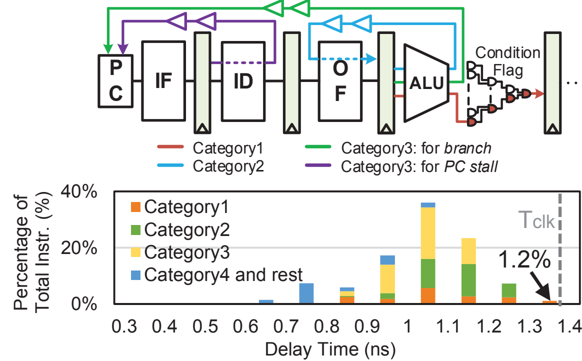

Fig. 3. Typical long delay instruction paths and the instruction category in delay time distribution view.

2.3 Instruction-based Clock Management

The instruction and hardware association above is completed for every instruction to find their corresponding dynamic delay bound T_{bound}. In the proposed scheme, the T_{bound} control bits are encoded into instruction codes, which will be sent to the control unit to dynamically adjust the processor clock period. The details are introduced in Section 4.

Fig. 4 shows the distribution of pipeline dynamic timing $T_{d,pipe}$ and the implemented clock based on bound timing T_{bound}. In an ideal situation, the clock period seamlessly tracks the real dynamic timing $T_{d,pipe}$. As a result, the effective clock frequency determined by $mean(T_{d,pipe})$ and the speedup is expressed as (3), which shows 34% frequency speedup benefit as listed in Table 1.

$$speedup, ideal = \frac{T_{clk}}{mean(T_{d,pipe})} - 1 \qquad (3)$$

However, in reality, the instruction level dynamic clocking can only follow T_{bound} instead of $T_{d,pipe}$ in order to pessimistically cover all possible instruction delay scenarios because it is not easy to predict the operand dependency of the instruction. Hence the practical frequency speedup is determined by $mean(T_{bound})$ as (4), in which smaller T_{bound} will bring more clock speedup.

$$speedup = \frac{T_{clk}}{mean(T_{bound})} - 1 = \frac{T_{clk} - mean(T_{bound})}{mean(T_{bound})} \qquad (4)$$

In the experiment for program *403.gcc*, the proposed clocking scheme following T_{bound} achieves effective frequency 906MHz, which obtains 20.76% clock frequency speedup comparing with conventional design frequency, as listed in Table 1.

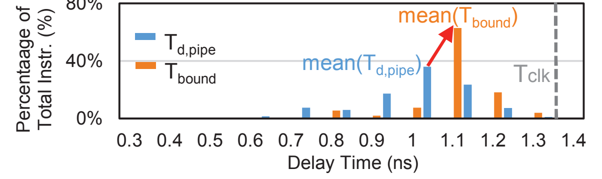

Fig. 4. Distribution of the pipeline dynamic clock period $T_{d,pipe}$ and the implemented dynamic clocking based on T_{bound}.

Table 1. Effective frequency speedup in program *403.gcc*.

	T_{clk}	$mean(T_{d,pipe})$	$mean(T_{bound})$
Effective T	1.33 ns	0.995 ns	1.104 ns
Effective f	750 MHz	1005 MHz	906 MHz
Speedup	0	34.0%	21.43%

3. PROGRAM-DRIVEN HARDWARE DESIGN OPTIMIZATION

As there is only small number of instructions execute the longest critical paths in the pipeline, it is in fact not efficient to only improve the longest critical path delay as the conventional design optimization. On the other hand, much larger benefits can be obtained if we optimize the hardware design based on the usage of instructions from software program. For instance, improving more frequently used instructions will lead to higher frequency speedup even if the instruction is not on the critical path. In fact, the area cost of optimizing such less critical instruction is much less than the cost of instructions on critical path. The proposed program-driven design optimization flow is shown in right bottom (B) of Fig. 1. First, we perform analysis on software programs, with the frequency of each instruction's usage extracted and ranked. Second, a sophisticated software instruction and hardware gate level netlist mapping is conducted, with a mathematical model of the weighting function developed to provide guideline on the benefits and costs of optimization for each instruction. Finally, the ranked instructions are optimized through backend optimization flow to obtain maximize speedup with minimum hardware costs.

3.1 Instruction Usage Mapping with Hardware

As shown in the example in Fig. 5(a), our cross-layer simulations which is explained earlier captures the instructions with the dynamic execution time and the critical path endpoint registers for each pipeline stage in one clock cycle. The pipeline stage with longest execution time determines the pipeline clock period $T_{d,pipe}$. At the same time, its corresponding path endpoint is denoted as *critical endpoint* (CEP), e.g. the *OF_Reg_64(ldr)* in Fig. 5(a). The number of times of each CEP register observed in simulations over the total instruction numbers is defined as CEP *usage percentage* p_{CEP}, which represents the individual CEP usage frequency during the program operations. The mapping of registers to a particular instruction allows us to associate software instruction usage with hardware gate level netlist. Fig. 5(b) shows the top 10 high usage percentage CEPs mapped with instruction type in one program. In practical, all CEP usage percentage are calculated and ranked.

To create finer mapping between instructions and gate level logic paths, corresponding instruction start points (SP) in gate level netlist are also obtained. In the proposed flow, we identify all the

possible SP register candidates based on the RTL design. Their register value transitions are stored during the simulations. As the example in Fig. 6, register SP1 and SP2 value transitions between 0 and 1, which could possible trigger the logic paths ended at CEP. Register SP3 maintains a constant register value and is not a valid path start point. As a result, SP1, SP2 and CEP construct a path group which maps into instruction "*ldr*". We formulate *path group* as shown in (5), where the instruction is mapped with *N* numbers of SP, CEP, dynamic delay bound, and the frequent usage of the instruction. All instructions are mapped in this format into gate level netlist. As shown in Fig. 7, all the gate level logics on the instruction paths are shown in the backend layout. Several high usage percentage instruction path examples are also highlighted.

$$Path_{instr(stage)} = \{SP_1, SP_2, \dots, SP_N, CEP, T_{bound}, p_{CEP}\} \quad (5)$$

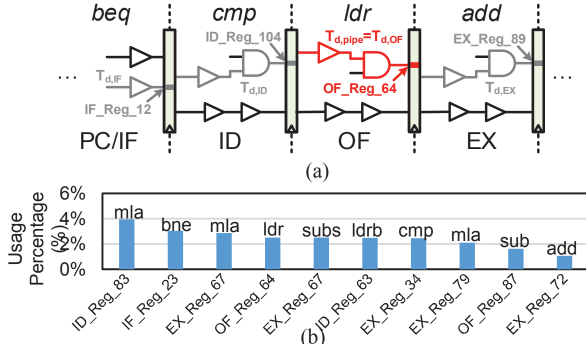

Fig. 5. (a) Simulation of the critical endpoint; (b) Top 10 usage percentage CEP mapped with instruction in one benchmark.

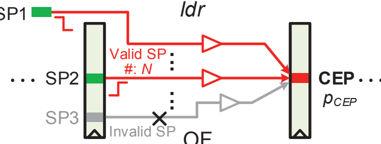

Fig. 6. Instruction path groups identified by SP and CEP.

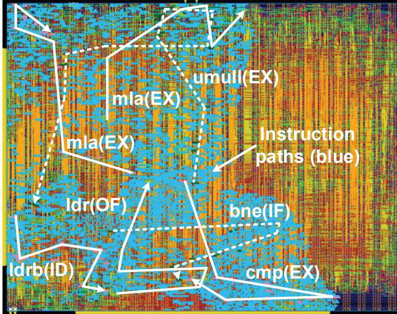

Fig. 7. Examples of instruction and logic path group mapping in the complete layout view of the ARM pipeline design.

3.2 Program-driven Optimization Method

In order to obtain more clock speedup benefit, a program-driven hardware optimization flow is developed. In general, specific optimization constraints are added to the instruction paths to constrain the path execution time, as (6).

$$T_{path} = T_{bound} - t_{opt} \quad (6)$$

in which t_{opt} represents *optimization strength*. Larger t_{opt} brings more clock speedup benefit while generates more area penalty. The optimization may become ineffective if t_{opt} is too large to cause timing violation. In our experiment, t_{opt} ranges from 0.1 to 0.3ns.

A mathematical model is developed to quantify the area cost as (7) in which *m* is the total number of path groups under optimization, t_{opt} is the target improvement of speed, and *N* is the path numbers in each path group. α is an empirical constant factor, which varies within 1.2e^{-3}~2e^{-3} in experiment. This cost equation relatively represents the difficulty of improvement, e.g. the number of logic path under optimization and the optimization timing target.

$$Cost = \alpha \sum_{i=1}^{m} t_{opt,i} \times N_i \quad (7)$$

All the instruction path groups are ranked by the *path weight* function considering their usage percentage p_{CEP}, area cost and associated T_{bound} for the instruction, as shown in (8). Here, larger T_{bound} is given higher optimization priority because it provides more speedup improvement space.

$$w_i = \frac{T_{bound,i} \times p_{CEP,i}}{Cost_i} = \frac{T_{bound,i} p_{CEP,i}}{\alpha t_{opt,i} N_i} \quad (8)$$

With the total optimization path groups number of *m*, the *optimization weighted sum* is expressed as (9). If all the instruction path groups are optimized, then $weight_{opt}$ is 100%. Fig. 8 shows the instruction weight distribution of all the path groups. It is interesting to observe that only small portion (<20%) of path group contribute to the weighted sum of 70%. This means we can selectively optimize small portion of instruction paths to obtain majority of speed up, which aligns with our earlier observation.

$$weight_{opt} = \frac{\sum_i^m w_i}{\sum w_i} \times 100\% \quad (9)$$

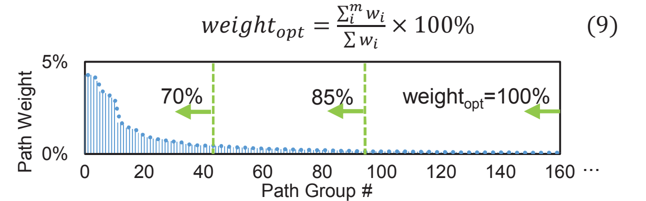

Fig. 8. Path weight distribution in benchmark *403.gcc.*

After the instruction path weights calculated, instruction paths are optimized by utilizing "*set_max_delay*" command to constrain the max execution delay of logic path to $T_{bound} - t_{opt}$. We also compared the optimization efficiency at either front-end synthesis stage or back-end place & route stage. Our experiment shows that optimizing design at back-end is more effective as it considers the practical place and route effects. As a result, we utilized our proposed optimization flow at back-end stage.

3.3 Optimization Performance and Cost

To verify the program-driven optimization performance and the cost, three cases with different optimization weighted sum are conducted, as shown in Fig. 9 and Table 2. Results show that more than half of the speedup benefits (4.98%) have been obtained in Case A (weighted sum of 70%) with only 1.8% area overhead. In Case B (weighted sum of 85%), speedup of 7.82% is achieved with area cost only 3.4%. If all the path groups are optimized, as in Case C, 9.64% speedup improvement is achieved with the area cost of 7.2%. This observation highlights the strength of our optimization method, i.e. majority of benefits are obtained with very little area overhead, which also demonstrates the effectiveness of our weighting functions. The proposed program-driven optimization is also compared with a "blind" optimization without the knowledge of instruction usage, listed by the dash curve in the Fig. 9. The comparison shows the proposed optimization achieves almost twice of speedup benefits compared with the "blind" optimization where the weight of instruction path is not considered.

Fig. 10 shows the dynamic delay distribution for baseline design and optimized cases simulated with complete backend P&R design. With the proposed optimization flow, instructions with dynamic timing ranges 1~1.2ns are constrained down to less than 1ns. When more path groups are optimized, more instructions are optimized to be shorter delay, as in Case C compared with Case B. Comparing the baseline design and Case C, a speedup of 9.64% is achieved with the proposed program-driven optimization method.

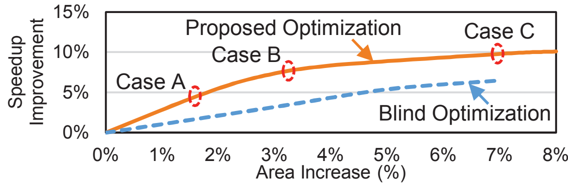

Fig. 9. Clock speedup improvement by proposed optimization.

Table 2. Optimization cases with different $weight_{opt}$.

	Baseline	Case A	Case B	Case C
$weight_{opt}$	0%	70%	85%	100%
Path Group #	0	40	91	218
Area Cost	0%	1.8%	3.4%	7.2%
Speedup Improve	0%	4.98%	7.82%	9.64%

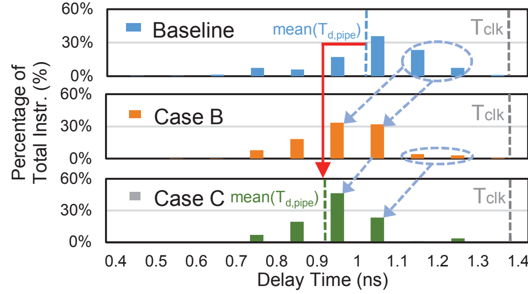

Fig. 10. Optimization effects on dynamic delay distribution.

4. SYSTEM SCHEME IMPLEMENTATION

4.1 Overall System Design

The overall diagram of proposed ultra-dynamic clock system is shown in Fig. 11, which includes ARM pipeline, control units, PLL, etc. The dynamic clock period management is determined by 3-bit control values which are encoded into each individual instruction code. This strategy is similar to the previous study with benign binary modification techniques for encoding information directly into the instruction stream [12]. The instruction is sent to both IF stage and control unit. The dynamic clock period control values are decoded by the control unit and sent to the PLL for glitch-less phase selection. Considering the delay time of controller unit, the dynamic clock period control value is encoded one cycle early than its actual execution cycle. Comparing with the conventional clocking, the hardware overhead of the proposed scheme is the phase selection multiplexer and control unit, which is negligible compared with overall processor area.

Besides the regular PC fetch, the pipeline could experience flush scenarios by instructions like *branch* or *ldr pc*, which trigger the PC recover signal. As the pipeline is blank after flush, the first few instructions entering the pipeline always complete within short time in the experiment. Thus the pipeline PC recover signal is also used to notice the controller to provide short clock period after each pipeline flush. The experiments show this PC recover function introduces additional 2% clock speedup benefit.

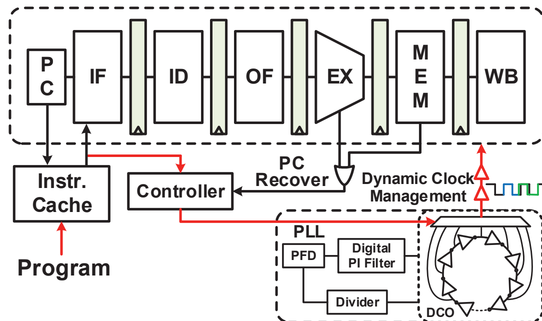

Fig. 11. Overall diagram of the proposed system scheme.

4.2 ADPLL Design and Phase Noise Margin

In order to generate cycle-by-cycle dynamic clocking, an all-digital phase locked loop (ADPLL) with clock phase selector is designed in 65nm technology with full transistor implementation. The ADPLL loop consists of time-to-digital converter (TDC), digital proportional-integral (PI) filter, digital controlled oscillator (DCO) and frequency divider, as shown in the Fig. 12. The DCO frequency is proportional to the drain current of the ring array and inversely proportional to the loading capacitance, which is similar to the design reported in [13]. There are 6bit coarse tuning and 7bit fine tuning to control active rings and loading capacitance, which achieve the coarse and fine resolution 30MHz and 0.3MHz. The ADPLL output frequency covers 30MHz to 2GHz.

For the DCO design, 11 stages are designed in each ring element, which provide total 22 phases with the constant delay $t_{delay} = T_{out}/22$. To maintain same delay between adjacent phase, identical fine capacitance loads area distributed at each phase. All these 22 phases are connected to a glitch-less multiplexer, which is selected by 5bit signal from the controller. Whenever the pipeline requires shorter/longer clock period, the mux selection is accordingly changed by n and generates $T_{shrink} = T_{out} - n \times t_{delay}$ or $T_{stretch} = T_{out} + n \times t_{delay}$.

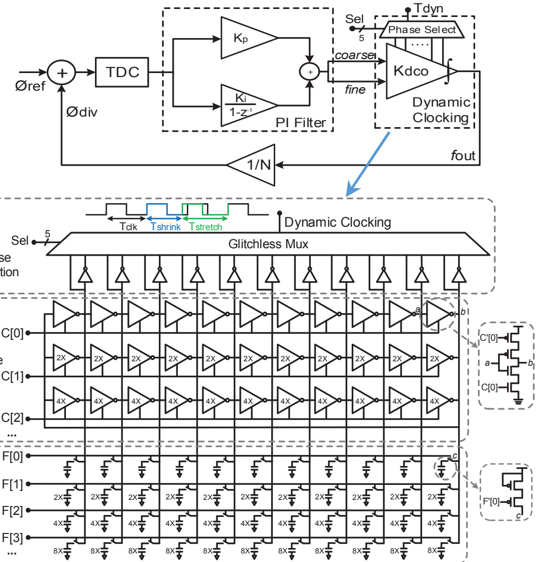

Fig. 12. Block diagram of the ADPLL design and the dynamic clocking generation.

The ADPLL phase noise is simulated and shown in Fig. 13. It is observed that phase noise is -108.4dBc/Hz at 1MHz offset. The DCO quantization noise is expressed by (10) and dominate the phase noise around the loop bandwidth due to DCO fine frequency tuning resolution, i.e. 0.3MHz, while it can be further suppressed by adding sigma delta module between DCO and PI filter [14].

$$Q(\omega) = T_{ref}\left|\frac{2\pi K_{DCO}}{\omega}\right|^2 \left|1 - e^{j\omega T_{ref}}\right|^2 \frac{1}{12} \tag{10}$$

The simulated phase noise is integrated resulting in cycle-to-cycle jitter $t_{jitter} = 7.8ps$ as defined by (11).

$$\sigma_{jitter}^2 = \int |X(\omega)|^2 4sin^2(\omega/2)d\omega \tag{11}$$

in which $X(\omega)$ represents the power spectrum of the phase noise [15]. The overall jitter performance leads to a conservative 6-sigma jitter of less than 50ps and will not introduce significant constraint on system timing budget. During the system level dynamic clock period adjustment, PLL jitter and other variations, e.g. PVT, should be considered into the safety margin, as (12).

$$T_{dyn} \geq \left(T_{clk} \pm n \times t_{delay}\right) + t_{jitter} + t_{PVT} \tag{12}$$

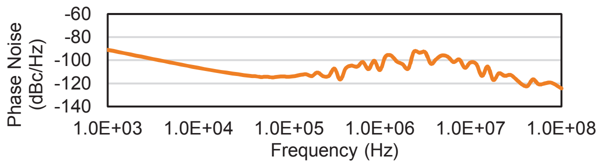

Fig. 13. Simulated phase noise for ADPLL at f_{out}=750MHz.

5. SIMULATION RESULTS

Six benchmark programs from benchmark SPEC CPU2006 [11] are simulated to evaluate the proposed scheme performance. Fig. 14 shows the improvement of the average clock frequency. Based on the conventional design flow, the pipeline is operated with frequency 750MHz. Applying the proposed instruction-based ultra-dynamic clocking strategy, the average clock frequency improved to around 910MHz, which achieve average 21.71% clock speedup. The proposed program-driven hardware optimization based on all these six benchmark programs is conducted to optimize design with area increase of 7.2%. The average clock frequency is further improved to 986MHz, which achieves speedup 31.56% of the average clock frequency. If converting the clock frequency speedup by scaling down the supply voltage, around 25% of power reduction could be obtained from the proposed scheme.

Fig. 15 shows examples of the program-driven optimization effects from the layout view, in which each logic path is mapped with instruction and its execution time. After the optimization, the exercise time of the same instructions are effectively reduced, which are mainly caused by the reduction of the cell numbers on the instructions paths. The proposed system scheme is also simulated in the Cadence Virtuoso AMS mixed-signal environment with full transistor level schematic of ARM pipeline and PLL design. As shown in Fig. 16, the ARM core clock period has been successfully adjusted based on each individual instructions.

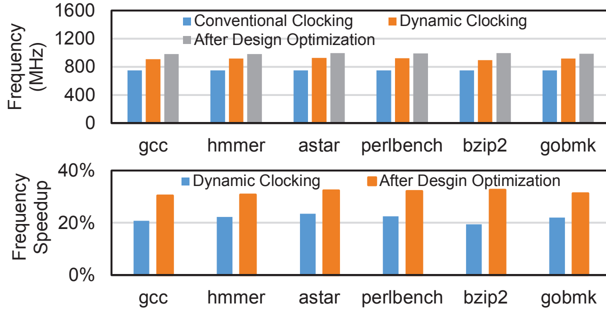

Fig. 14. Comparison of processor effective frequency (upper) and the frequency speedup results (lower).

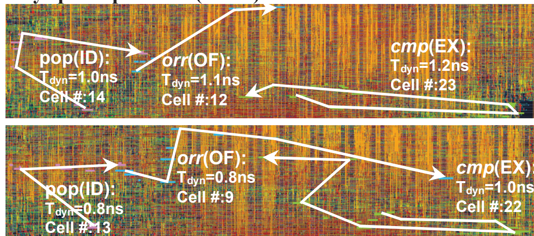

Fig. 15. Instruction path examples before (upper) and after (lower) the proposed optimization.

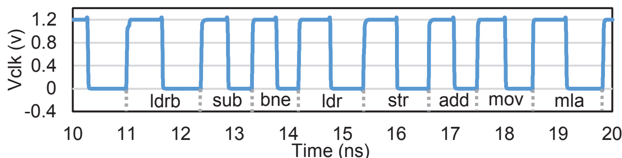

Fig. 16. Full transistor level AMS simulation of the ARM pipeline and ADPLL.

6. CONCLUSION

In this paper, a novel Greybox design methodology is proposed to cross the conventional HW/SW design boundaries to bring new design and co-optimization opportunities. Based on architecture and circuit level co-simulations, the dynamic timing margin of instructions is associated with the hardware pipeline design to provide the basis of ultra-dynamic clock management scheme, leading to a 21.7% frequency speedup. Furthermore, a novel program-driven hardware optimization flow is proposed, in which sophisticated mathematical weighting model is developed to map the frequent usage instructions with the hardware logic paths. Additional 10% frequency speedup is obtained by the proposed program-driven optimization flow. Overall, the proposed Greybox design methodology achieves clock frequency speedup by 31.5% on an ARM ISA based pipeline design, comparing with the conventional design methodology.

7. ACKNOWLEDGMENTS

This work is partially supported by NSF grants CCF-1618065 and CCF-1116610.

8. REFERENCES

[1] J. Howard, et al., "A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling", *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 173-183, Jan. 2011.

[2] Z. Toprak-Deniz, et al., "Distributed system of digitally controlled microregulators enabling per-core DVFS for the POWER8 microprocessor", *International Solid-State Circuits Conference (ISSCC)*, pp. 98-99, Feb. 2014.

[3] S. Kim, et al., "Enabling wide autonomous DVFS in a 22 nm graphics execution core using a digitally controlled fully integrated voltage regulator", *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 18-30, Jan. 2016.

[4] T. Jia, et al., "Exploration of associative power management with instruction governed operation for ultra-low power design", *Design Automation Conference (DAC)*, 2016.

[5] J. Xin, et al., "Identifying and predicting timing-critical instructions to boost timing speculation", *International Symposium on Microarchitecture* (*MICRO)*, pp. 74-85, 2011.

[6] J. Constantin, et al., "Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment", *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015.

[7] H. Cherupalli, et al., "Exploiting dynamic timing slack for energy efficiency in ultra-low-power embedded systems", *International Symp. on Computer Architecture (ISCA)*, 2016.

[8] S. Das, et al., "RazorII: in situ error detection and correction for PVT and SER tolerance", *IEEE Journal of Solid-State Circuits*, vol. 44, no. 1, pp. 32-48, Jan. 2009.

[9] Online resource, ARM, "ARMv5 Architecture Reference Manual", https://silver.arm.com/download/download.tm?pv=1073121

[10] Online resource, http://www.gem5.org/Main_Page

[11] J. Henning, et al., "SPEC CPU2006 benchmark descriptions", *Computer Architecture News*, 34(4), Sep. 2006.

[12] A. Meixner, et al., "Argus: Low-cost, comprehensive error detection in simple cores", *International Symposium on Microarchitecture* (*MICRO)*, pp. 210-222, 2007.

[13] N. August, et al, "A TDC-less ADPLL with 200-to-3200MHz range and 3mW power dissipation for mobile SoC clocking in 22nm CMOS", *International Solid-State Circuits Conference (ISSCC)*, Feb. 2012.

[14] M. Perrott, "Tutorial on digital phase-locked loops", *Custom Integrated Circuits Conference (CICC)*, 2009.

[15] U. Moon, et al., "Spectral analysis of time-domain phase jitter measurement", *IEEE Transactions on Circuits and System II*, vol. 49, no. 5, pp. 321-327, 2002.