Exploiting Dynamic Timing Margins in Microprocessors for Frequency-Over-Scaling with Instruction-Based Clock Adjustment

Jeremy Constantin*, Lai Wang†, Georgios Karakonstantis*, Anupam Chattopadhyay‡, and Andreas Burg*
* Telecommunications Circuits Laboratory, Institute of Electrical Engineering, EPFL, Switzerland
Email: {jeremy.constantin,georgios.karakonstantis,andreas.burg}@epfl.ch
† MPSoC Architectures Research Group, UMIC, RWTH Aachen University, Germany
‡ School of Computer Engineering, NTU, Singapore

***Abstract*—Static timing analysis provides the basis for setting the clock period of a microprocessor core, based on its worst-case critical path. However, depending on the design, this critical path is not always excited and therefore dynamic timing margins exist that can theoretically be exploited for the benefit of better speed or lower power consumption (through voltage scaling). This paper introduces predictive instruction-based dynamic clock adjustment as a technique to trim dynamic timing margins in pipelined microprocessors. To this end, we exploit the different timing requirements for individual instructions during the dynamically varying program execution flow without the need for complex circuit-level measures to detect and correct timing violations. We provide a design flow to extract the dynamic timing information for the design using post-layout dynamic timing analysis and we integrate the results into a custom cycle-accurate simulator. This simulator allows annotation of individual instructions with their impact on timing (in each pipeline stage) and rapidly derives the overall code execution time for complex benchmarks. The design methodology is illustrated at the microarchitecture level, demonstrating the performance and power gains possible on a 6-stage OpenRISC in-order general purpose processor core in a 28 nm CMOS technology. We show that employing instruction-dependent dynamic clock adjustment leads on average to an increase in operating speed by 38% or to a reduction in power consumption by 24%, compared to traditional synchronous clocking, which at all times has to respect the worst-case timing identified through static timing analysis.**

I. Introduction

Aggressive technology scaling has allowed the continuous increase of processor performance while maintaining the same power consumption across generations. However, the worsening of variations and the enormous design margins enforced by conventional techniques for avoiding timing failures limit the performance returns from technology scaling and lead to power overheads especially in sub-40 nm nodes. Particularly one type of highly pessimistic margin has already burdened the traditional design flow even long before the recent rise of variability issues: the traditional synchronous design paradigm determines the operating frequency according to the worst critical path that is identified through static timing analysis under assumed worst case conditions. However, this critical path is not always excited, thus leading to processors that perform (much) slower than what they can theoretically achieve.

A. Related Work

The pessimistic limitation of the clock frequency by the worst-case path in synchronous designs led to a short research trend that promoted the idea of asynchronous circuits [1]. However, even though the success of pioneering early work [2] in that field showed noticeable speedup and even power savings, the lack of a consistent methodology and reliable tools, as well as considerable design risks led industry to effectively abandon the asynchronous design approach.

More lately, related, but quite different adaptive design techniques have become popular for limiting the timing margins (due to delay uncertainty from process-variations) in pipelined processors. As opposed to asynchronous design, these techniques focus on in-situ error detection and correction mechanisms in (near) timing-critical paths [3]–[5]. A popular approach for operating beyond the 'always correct' critical timing/voltage margin is for example to stall a pipeline or replay instructions, when a timing violation is detected. However, the enforced timing constraints for timing-error detection and the overheads for the applied recovery methods (e.g., multi-cycle replay penalties), especially in case of high activation probability of critical paths can offset the gains achieved by removing the static or dynamic safety margins. Alternative methods include the use of occasional two cycle operations, when operands are detected that activate the critical path in arithmetic units through delay prediction circuits [6]. Unfortunately, such a method can also lead to performance overheads in case of high excitation probability of the long latency paths and its application has so far been limited to arithmetic circuits. In any case, both methods still do not consume all margins, due to the constraints imposed by the used flip-flops and the widespread coarse grained application of clock stretching from one to two (or multiple) cycles, leaving a large potential for unexploited timing slacks on the table.

Recently another method was proposed [7] that enforces time borrowing within a fixed window between pipeline stages and that stretches the clock in the next clock cycle for avoiding any timing failure. Although such a method can reduce the incurred penalties of the aforementioned techniques, it still leaves unexploited potential timing margins due to the fixed time borrowing windows and entails considerable design complexity due to the special flip-flops that need to be integrated. Only last year a less intrusive design method was proposed that tried to reduce static timing margins by applying application-adaptive guard-banding [8]. Although this method is very attractive, it mainly focused on tackling temperature and process variations, while the dynamic clock adjustment in the used LEON3 core is applied on a rather coarse granularity, discerning only two instruction classes based on their error tolerance capability. In addition, a systematic approach for characterizing the required execution time per instruction is required to apply the idea on a finer granularity.

B. Contributions and Outline

In this paper, we depart from the conventional static timing analysis based clocking and from existing design intrusive techniques based on a-posteriori timing-error detection and correction. Instead, we introduce a predictive dynamic clock adjustment technique to trim dynamic timing margins by exploiting the different timing requirements of individual instructions in a processor. Our approach not only limits the design complexity of the aforementioned approaches but its application at the instruction level combined with its fine grained clock adjustment also ensures the utilization of a large part of the timing slacks that are available during the dynamically varying program execution flow. More specifically, the paper makes the following contributions:

- We propose to dynamically adjust the clock period of a pipelined microprocessor on a cycle-by-cycle basis to achieve an average clock frequency beyond the one given by the static timing analysis limit (*frequency-over-scaling* without introducing timing errors).
- We propose and show how to dynamically set the clock period based on the instructions in the pipeline.
- We develop a generic framework that characterizes the delay required per instruction and per pipeline stage, through post-layout dynamic timing analysis.
- We apply the proposed approach to a popular open source RISC processor core after optimizing it to smoothen the timing wall that would otherwise reduce the gains from our technique.
- To allow rapid evaluation of the proposed approach for any complex benchmark, we develop a fast custom cycle accurate simulator, which accurately tracks the dynamic clock adjustment and reveals the overall code execution time (average/effective clock frequency).
- Finally, we show the potential improvements in speed and power consumption (by utilizing the speed gains for voltage-frequency scaling) for various benchmarks with an advanced technology node.

The rest of the paper is organized as follows: Section II presents the proposed design technique and the steps of the proposed design flow. Section III describes the processor core used as a case study for applying our method, briefly presents the microarchitecture modifications and introduces our custom instruction set simulator, used for performance evaluation. The performance gains for several benchmarks and the obtained energy efficiency improvements are presented in Section IV. Conclusions are drawn in Section V.

II. Proposed Approach and Design Flow

As almost any digital circuit, a pipelined processor typically consists of a set of N unique combinational paths $\mathcal{P} = \{p_1, ..., p_N\}$, which are characterized by their delays $D(p_n)$ for $n = 1, ..., N$ (each including the setup time of the end-point). In a pipeline with S stages, each path can be attributed to exactly one stage $s = 1, ..., S$, based on its end-point, and we can define corresponding exclusive per-stage path groups $\mathcal{P}^s$ such that $\bigcup_{s=1}^{S} \mathcal{P}^s = \mathcal{P}$ and $\mathcal{P}^s \cap \mathcal{P}^{s'} = \emptyset$ for $s \neq s'$. In a synchronous design, the longest path (in terms of delay) across all pipeline stages determines the clock period such that

$$T_{\mathrm{CLK}}^{static} = \max_{s=1,...,S} \left\{ \max_{p \in \mathcal{P}^s} \{D(p)\} \right\} = \max_{p \in \mathcal{P}} \{D(p)\}. \quad (1)$$

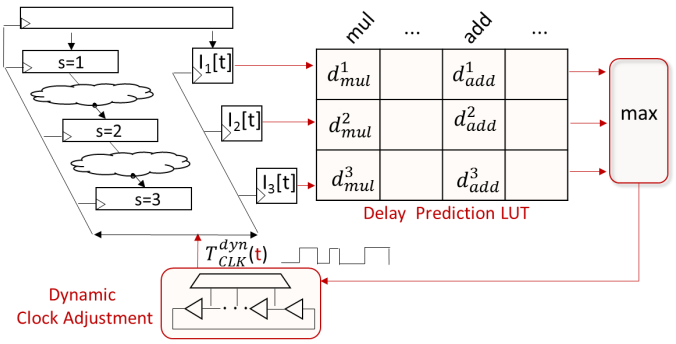

Fig. 1. Proposed instruction-based dynamic-clock adjustment technique in a S=3-stage pipelined processor

However, the bound put on the clock period in (1) is overly pessimistic since it ignores the fact that in each pipeline stage, the respective longest path $\max_{p \in \mathcal{P}^s} \{D(p)\}$ is not always excited. Instead, at time t, both operands and especially the instruction $I_s[t]$ that is currently in-flight in stage s determine a subset of relevant paths $\mathcal{P}^s_{I_s[t]} \subset \mathcal{P}^s$ that is sufficient to determine the position of the next clock-edge for that pipeline stage. Since all registers are connected to the same clock, we can now determine a safe, but less pessimistic clock period for time instant t as follows

$$T_{\mathrm{CLK}}^{dyn}[t] = \max_{s=1,...,S} \left\{ \max_{p \in \mathcal{P}^s_{I_s[t]}} \{D(p)\} \right\} \leq \max_{p \in \mathcal{P}} \{D(p)\}. \quad (2)$$

The corresponding relevant (active) instructions $I_s[t] = L[t + 1 - s], s = 1, ..., S$ are thereby determined by the S subsequent instructions in the program trace $L[t], t = 1, ..., M$ which therefore influence the average cycle time taken over the entire program execution [1] $T_{\mathrm{CLK}}^{avg} = \frac{1}{M} \sum_{t=1}^{M} T_{\mathrm{CLK}}^{dyn}[t]$.

In less formal words, we can say that in each cycle the *relevant path* for each pipeline stage depends on the instruction [2] of the program that is currently in that stage. The longest relevant path across all stages then determines the clock period for that cycle that is necessary to avoid timing violations.

A. Instruction Based Clock Adjustment

The main idea behind our approach is to exploit different timing requirements of the various instruction sequences in the program trace to opportunistically over-scale the average frequency (i.e., to operate effectively at $T_{\mathrm{CLK}}^{avg} < T_{\mathrm{CLK}}^{static}$). To this end, we propose to adjust the clock period on a cycle-by-cycle basis, ideally according to (2). The corresponding architecture, as shown in Fig.1 monitors the instructions $I_s[t]$ that are in-flight in all processor pipeline stages $s = 1, ..., S$ and uses a lookup table (LUT) for each stage that contains for each instruction type the maximum delay of all relevant paths $d_I^s = \max_{p \in \mathcal{P}_I^s} \{D(p)\}$ in that stage. These per-stage maximum delays (for the respective current instructions $I^s_{I_s[t]}$) are then combined to yield $T_{\mathrm{CLK}}^{dyn}[t]$ and to adjust a tunable

[1] For simplicity, we assume long programs with limited impact of initialization effects.

[2] The delay of the relevant path also depends on other conditions, such as the operand values, state of the forwarding logic (influenced by the instructions in other stages), as well as the temperature, which are all accounted for by considering their worst case effects on the path delay.

clock generator (CG) accordingly in each cycle. Such a CG can for example be realized in form of a tunable ring oscillator with a muxed clock output [9], [10] or via a multi-PLL clocking unit such as the one proposed in [11], which is thus beyond the scope of this paper. We note that the design of an appropriate CG can have a significant influence on the system power consumption, and requires special care.

B. Design Flow

The design flow for realizing and evaluating the proposed cycle-by-cycle adaptive clocking is shown in Fig. 2.

1) Implementation

The first step in the process is the RTL implementation, optimization, synthesis, and place & route of a suitable microprocessor. While we keep the objective of achieving a high clock frequency, as determined by static-timing-analysis, we also note that this worst-case timing is not the only objective to obtain a high average clock frequency. Thus, it is important to not only optimize the critical path of the design, but also to reduce the number of near-critical paths and to keep the remaining paths short to minimize $\max_{p \in \mathcal{P}_I^s}\{D(p)\}$, $s = 1, ..., S$ for all (or at least all frequent) instructions I. Unfortunately conventional implementation strategies and well balanced pipelines tend to produce a so-called timing wall as shown in Fig. 3(a), since they focus on the critical path only and allow other paths to become near-critical to recover area or power [12]. This timing wall typically has no negative impact on the static timing limit, but on the gains available from the dynamic clock adjustment proposed in this paper. We therefore propose to use suitable synthesis and implementation constraints (and possibly dedicated tools as in [12]) that also optimize sub-critical paths to keep them short, as illustrated in Fig. 3(b). While this objective usually involves overhead (in area, speed, and power), we will later show that this overhead can be kept small. In addition, to avoid the timing wall in the synthesis and place & route step, optimizations at the register-transfer-level can be used to remove long, but functionally irrelevant paths from $\mathcal{P}_I^s$, for example through shielding.

2) Characterization

The second step in the process is the characterization of the dynamic timing margins per instruction and per stage. This information is used for performance evaluation of the approach and eventually to populate the lookup-table for the clock period adjustment of the design. To this end, we start from the final netlist and the standard delay format (SDF) post-layout timing information of the design and perform gate-level simulations. We run different test programs, which include small hand-written kernels as well as semi-random test-cases that are generated by a code generation tool. The simulation outputs value change dumps (VCDs) that are used for power analysis based on the switching activity of the core. Moreover, the evolution of the program counter is recorded to be able to generate a program trace $L[t]$ from the disassembled binaries. The next step is the *dynamic timing analysis*. In contrast to conventional static timing analysis, we are interested in the dynamic behaviour of the path endpoints during execution of actual programs. Dynamic timing analysis aims to uncover the unused timing margins of the processor that are available at run-time, which can not accurately be characterized through static timing analysis, due to the missing notion of path activation probabilities. To obtain this information, the gate-level simulation (at a "low" clock frequency) also monitors the inputs (data and clock) of all flip-flops and memory macros in the design and outputs a corresponding event log. This log is then provided to a custom *dynamic timing analysis* tool. For each sequential element (end-point), this tool identifies the time of the last event on its data-input pin in each clock cycle and relates it to the arrival time of the next active clock edge on the clock input of the same memory element. This individual comparison is important to also account for clock skew, which is often even introduced artificially to improve the post-layout timing. The difference between the clock and data activity time stamps (minus the setup time) is then recorded as the available dynamic slack for this particular element for each clock cycle. The time-average over the worst-case slack among all end-points in each cycle provides an optimistic lower-bound on the average clock period T_{CLK}^{avg} of the processor, including all data and instruction dependencies.

In a next step, the analysis tool partitions path end-points into path-groups $\mathcal{P}^s$ according to a provided pipeline specification of the processor and determines the longest delay per pipeline stage and per cycle $\max_{p \in \mathcal{P}_{L[t]}^s}\{D(p)[t]\}$, $s = 1, ..., S$, where $D(p)[t]$ is the dynamic delay observed on path p as measured in cycle t. The corresponding dynamic timing limits at pipeline stage granularity are then combined with the program trace to produce the dynamic slack distributions and the minimum cycle times $d^s[t] = \max_{p \in \mathcal{P}_{L[t]}^s}\{D(p)[t+s-1]\}$, $s = 1, ..., S$ for all executed instructions $L[t]$ in the program trace. The maximum delays per instruction and pipeline stage are then extracted by taking the maximum across all occurrences of that instruction in the program trace as $d_I^s = \max_{t:L[t]=I}\{d^s[t]\}$ and are then exported and are used to fill-up the delay prediction LUT (cf. Fig. 1), characterizing the instruction-dependent clock adjustment potential.

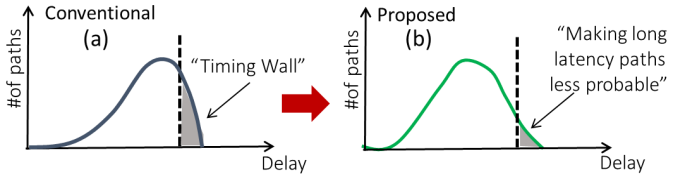

Fig. 3. Timing profile in a conventional processor vs the proposed

III. Case Study and Evaluation Environment

The proposed approach is applied to a general purpose open-source processing core, the OpenRISC [13]. This section discusses the microarchitectural modifications, as well as the applied constraints for reducing the impact of a timing wall and making the core suitable for our technique. Additionally, we discuss the performance evaluation environment, based on a custom instruction set simulator.

A. OpenRISC Microarchitecture

The mor1kx cappuccino microarchitecture implementation of the OR1000 OpenRISC architecture, which is an in-order 32-bit RISC pipeline consisting of six stages, is used as our case study. The schematic of the adapted core is shown in Fig. 4. The fetch unit and load and store units, as well as parts of the arithmetic and logic unit (e.g., the multiplier) have been optimized to achieve close to one instruction per cycle execution throughput. The ALU is comprised of an adder and a single-cycle multiplier (with 32-bit output), along with a

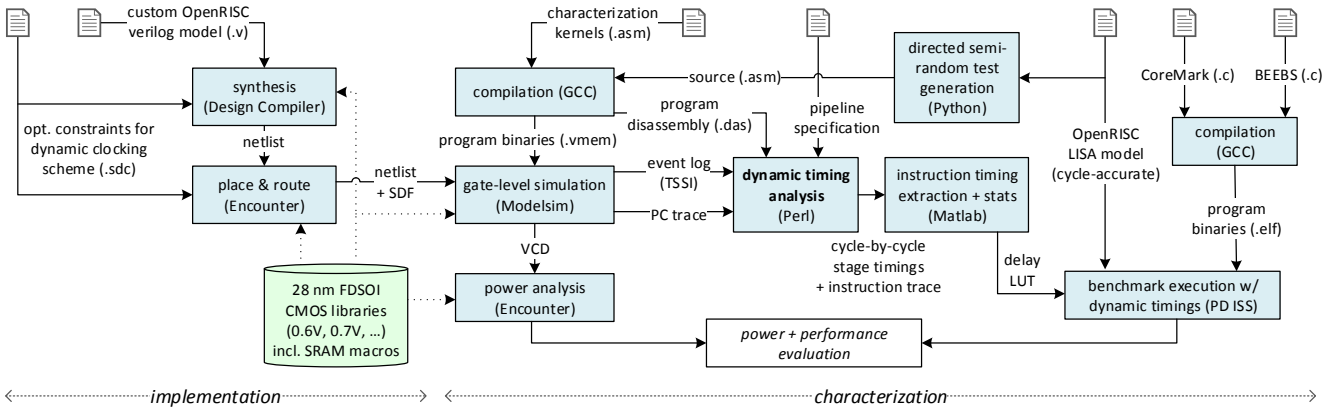

Fig. 2. Proposed design flow including dynamic timing analysis, instruction timing extraction and evaluation

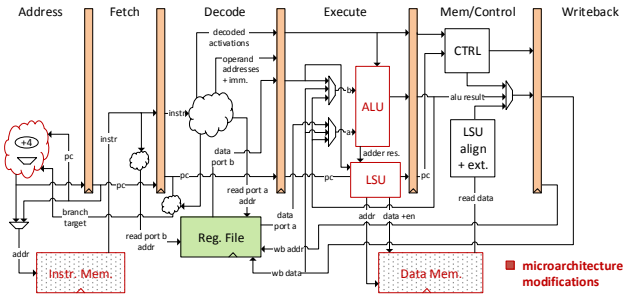

Fig. 4. Customized mor1kx microarchitecture of OpenRISC

TABLE I. EFFECTS OF CRITICAL RANGE OPTIMIZATION ON DYNAMIC INSTRUCTION DELAY WORST-CASES

Instruction	Max. delay factor
l.add(i)	0.92
l.bf	0.78
l.j	0.74
l.lwz	0.85
l.mul	1.10
l.nop	0.78
l.sw	0.85
...	...

shifter. A single 32-entry register file with two read ports and one write port is used. Furthermore, to make the core suitable for integration in an increasingly popular low-power many-core accelerator [14], the microarchitecture of the pipeline has been modified in order to support a tightly coupled memory interface, for the instruction as well as the data memory. Since both memories are implemented as fast SRAMs, access latency for instructions and data are both single-cycle. Other modifications include also the integration of the data memory requests and parts of the load store unit in the execute stage.

To increase the opportunities for shorter clock cycles, the multiplier has been shielded from the inputs of the other units of the ALU by separate, parallel registers that are only loaded for multiplications. This measure avoids unnecessary ”parasitic” activity for operations other than multiplications, which not only reduces power but also avoids excitation of the long paths through the multiplier during other operations.

To reduce the issue of a timing wall and increase the number of short paths as described in Sec. II-B, we utilized the ”critical-range” optimization feature of Synopsys Design Compiler along with path over-constraining during synthesis.

Clearly, this optimization does not come for free in terms of area and power, however a comparison to a core without the critical-range optimization shows that area and power penalties can be limited to 5-13% in a 28 nm fully depleted silicon on insulator (FDSOI) CMOS technology, depending on the target library and operating voltage. The positive effects of the applied design step for our dynamic clocking technique are shown in Table I, which reports the factors for the dynamic maximum instruction delays, when applying the critical-range optimization, compared to a standard implementation of the same design. Interestingly, the overall minimum clock period derived from static timing analysis increases by 9% with the critical-range constraints, due to unwanted side effects of the synthesis tool. However, the worst case delay excited by most instructions reduces significantly compared to the conventional design, and the slight increase in delay is only visible for the multiplication instruction.

B. Performance Evaluation Environment

We evaluate the performance gains with the help of a cycle-accurate instruction set simulator of the processing core, which is enhanced to be aware of the dynamic clock adjustment technique. While still operating on a cycle-by-cycle basis, it accounts for varying real-time instruction delays according to the delay table generated by our dynamic timing analysis tool flow. We base our instruction set simulator on a custom developed LISA[3] model of our OpenRISC core, and employ the Synopsys Processor Designer tool suite to generate the custom simulator from this model. Performance benchmarks are then run using the popular embedded benchmark suites CoreMark [15] and BEEBS [16]. Their C sources are compiled with the standard OpenRISC GNU toolchain (GCC). The execution time and speedup results of the simulator are finally combined with the extracted power consumption values to evaluate the energy efficiency gains from voltage-frequency scaling[4] for constant execution time.

[3]Language for Instruction Set Architectures

[4]Voltage-frequency scaling is based on fully characterized cell libraries for different operating points.

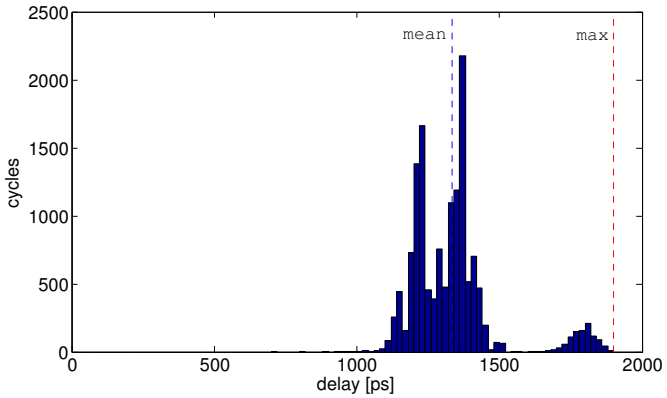

Fig. 5. Histogram of dynamic maximum delays per cycle over all pipeline stages of the OpenRISC core

IV. Results

We present the results of our proposed design method in two parts. The first part focuses on the characterization of the dynamic timing margins per instruction and per stage for the OpenRISC core by applying the developed flow as described in Section II. The second part leverages these margins and presents the achievable performance and power gains, when executing popular benchmark suites on our enhanced core with instruction-based clock adjustment.

A. Dynamic Timing Analysis of OpenRISC

Before characterizing the core timing on an instruction level, we study an upper-bound on speed improvements from dynamic clocking with a genie-aided clock adjustment. To this end, we perform dynamic timing analysis, but initially we do not enforce a worst-case clock period for all occurrences of the same instruction. Instead, we assume that the duration of each cycle can be adjusted according to the a-posteriori measured dynamic timing slack. Figure 5 shows that considering all endpoints of the core (including the SRAMs), on average we only require a delay of 1334 ps for the correct program execution, in contrast to the conservative limit of 2026 ps given by static timing analysis. With correct program execution we mean that all timing requirements of all excited paths in any given cycle are always met. The histogram shows the distribution of the dynamic slack available in the full core on a cycle-by-cycle basis. Exploiting these dynamic slacks, the theoretical average speedup is 50%.

When introducing endpoint groupings according to the pipeline stages, our analysis shows that the execution stage is the dominating stage, regarding worst-case dynamic paths (cf. Figure 6). In particular, for 93% of the cycles the overall maximum delay, defining the length of a cycle in Figure 5 is due to an endpoint that is located between the execute and control stage, which can be a pipeline register or the data memory SRAM macro cell. In 7% of the cases endpoints attributed to the address stage are the limiting factor, which corresponds to the instruction memory endpoints. Rarely the fetch or decode stage can also be the limiting stage, however these cases appear in less than 1% of cycles, and the limiting delays are in these cases very short.

These results indicate that clock adjustment can be performed in the case of our OpenRISC core by considering only the delay of an instruction in the execute stage, as long as

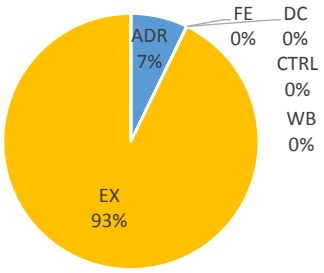

Fig. 6. Percentage of a pipeline stage containing the limiting path, which determines the minimum cycle time when employing dynamic clocking

TABLE II. Dynamic instruction delay worst-cases

Instruction	Max. delay [ps]	Stage
l.add(i)	1467	EX
l.and(i)	1482	EX
l.bf	1470	EX
l.j	1172	ADR
l.lwz	1391	EX
l.mul	1899	EX
l.sll(i)	1270	EX
l.xor	1514	EX
...	...	...

it is guaranteed that the instruction memory address timings (address stage) are always respected (as well as the few cases where other stages are limiting, which however have a short delay in general for these cases). This fact can significantly simplify the clock adjustment control module, since its pipeline monitoring can be simplified.

Table II presents a selective overview of extracted maximum instruction timings based on a characterization benchmark with a gate-level simulation of 14 k cycles. Such a table (with 6 stage-entries per instruction) is employed in our instruction set simulator to accurately model the instruction delays. Instructions where no accurate maximum delay characterization could be performed (due to limited number of occurrences in the benchmark) are represented in the table with the worst-case clock period timings from static timing analysis.

In Figure 7 we illustrate the dynamic-timing distributions for the 6 pipeline stages in the core for the l.mul instruction (unsigned 32-bit multiplication), as derived from our dynamic timing analysis tool. It can be observed that while the delay of the instruction in the execute stage is in general high (and close to the static maximum), delays for the other stages are significantly lower, which can be exploited when other, faster instructions are currently in the execution stage. The distribution of delays with a spread of about 300 ps for the execute stage stems from the varying data dependent activations of worst case paths. This data dependent timing variation could be further leveraged by approximate computing techniques [17], which would introduce the notion of using shorter clock periods to enhance performance or save energy, while actually allowing a violation of the timing requirements of certain paths in the design. These violations would then produce approximate results, for example for the output of our multiplication instruction, due to paths in the execution stage of the multiplier circuit being violated under certain conditions (e.g. critical operands exciting the worst paths).

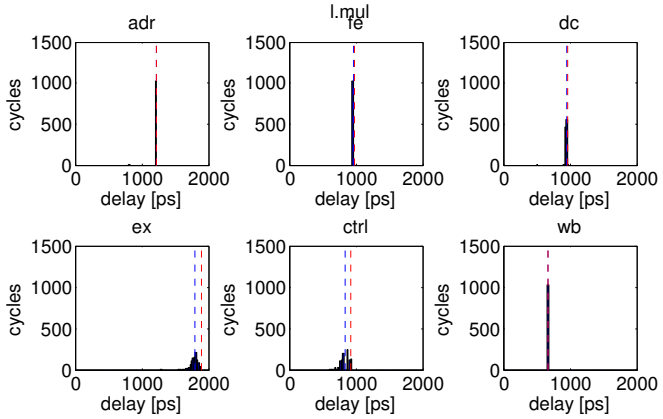

Fig. 7. Histograms of dynamic maximum delays per pipeline stage, for the l.mul instruction (multiplication)

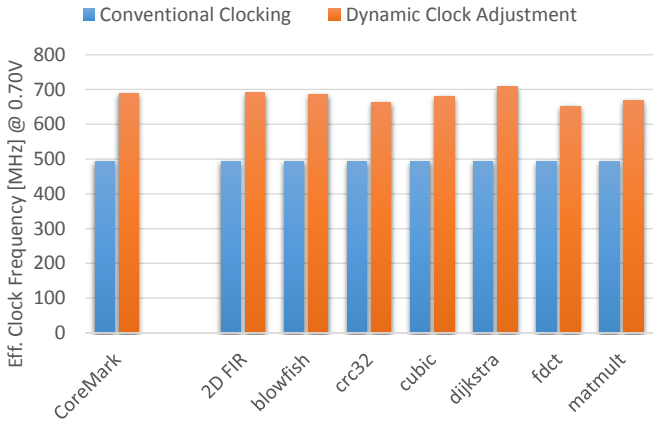

Fig. 8. Performance gains with dynamic clock adjustment for the CoreMark and BEEBS benchmark suites

B. Performance and Power

We evaluate the performance gains of the presented dynamic clock adjustment method by executing the popular CoreMark and BEEBS embedded benchmark suites [15], [16] on our custom instruction set simulator with integrated delay tables. The benchmark-dependent speedups are reported in Figure 8. On average the effective clock frequency can be increased by 38%, from 494 MHz (static timing limit) to 680 MHz at a fixed supply voltage of 0.70 V. We focus our evaluation in 28 nm FDSOI CMOS on a low supply voltage for reasons of higher energy efficiency.

These numbers show that in terms of clock speed on average we only give up 12% by performing clock adjustment based on the instruction types alone. This is compared to the theoretically achievable speedup of 50%, when adjusting the clock period perfectly each cycle, i.e. when considering all data and instruction dependencies, to fully consume all available dynamic timing margins.

The increase in performance can also be traded for reduced power consumption through voltage scaling of the core. The available speedup allows on average to operate the core with a supply voltage that is 70 mV lower, which translates into an improvement in energy efficiency by 24%. Under scaled voltage, but with dynamic clock adjustment, the core consumes only 11.0 μW/MHz while providing the same throughput, compared to 13.7 μW/MHz for the conventional clocking scheme.

V. Conclusion

In this paper, we presented a fine grained dynamic-clock adjustment technique to trim dynamic timing margins by exploiting the different timing requirements of individual instructions in a microprocessor. The proposed approach does not require any error detection and recovery mechanisms to exploit dynamic timing margins, thus eliminating the associated power and performance overheads, while also limiting the increase in design complexity and the associated risks. To ensure considerable performance gains, the proposed approach starts with the design of the processor with a suitable timing profile, characterized by many short paths. The developed integrated design and dynamic timing analysis flow used to characterize the improved timing upper bounds of each instruction illustrates the steps that should be followed for the application of our approach to any other design. The application of the proposed approach to a RISC processor (6-stage OpenRISC) demonstrates for popular embedded benchmarks that on average the speed of the design can be increased by 38%, which can be translated to power savings of 24%. The proposed approach is orthogonal to existing timing error detection and recovery mechanisms, and could be effective in accounting for other static and dynamic timing variations, for example due to process, temperature and voltage fluctuations, by (online-)updating of the used delay prediction table.

References

[1] J. Spars *et al.*, *Principles Asynchronous Circuit Design*. Springer, 2002.

[2] J. Woods *et al.*, "AMULET1: an asynchronous ARM microprocessor," *Computers, IEEE Trans. on*, vol. 46, no. 4, pp. 385–398, Apr 1997.

[3] D. Ernst *et al.*, "Razor: a low-power pipeline based on circuit-level timing speculation," in *IEEE/ACM MICRO*, Dec 2003, pp. 7–18.

[4] S. Das *et al.*, "RazorII: In situ error detection and correction for PVT and SER tolerance," *IEEE JSCC*, vol. 44, no. 1, pp. 32 –48, jan. 2009.

[5] K. Bowman *et al.*, "A 45 nm resilient microprocessor core for dynamic variation tolerance," *IEEE JSCC*, pp. 194 –208, jan. 2011.

[6] S. Ghosh *et al.*, "CRISTA: A new paradigm for low-power, variation-tolerant, and adaptive circuit synthesis using critical path isolation," *IEEE TCAD*, vol. 26, no. 11, pp. 1947–1956, Nov 2007.

[7] K. Chae *et al.*, "A dynamic timing error prevention technique in pipelines with time borrowing and clock stretching," *IEEE TCAS I*, vol. 61, no. 1, pp. 74–83, Jan 2014.

[8] A. Rahimi *et al.*, "Application-adaptive guardbanding to mitigate static and dynamic variability," *Computers, IEEE Trans. on*, vol. 63, no. 9, pp. 2160–2173, Sept 2014.

[9] J.-H. Kim *et al.*, "A 120-MHz-1.8-GHz CMOS DLL-based clock generator for dynamic frequency scaling," *IEEE JSSC*, vol. 41, no. 9, pp. 2077–2082, Sept 2006.

[10] J. Tierno *et al.*, "A DPLL-based per core variable frequency clock generator for an eight-core POWER7 microprocessor," in *IEEE VLSIC*, June 2010, pp. 85–86.

[11] Tschanz *et al.*, "Adaptive frequency and biasing techniques for tolerance to dynamic temperature-voltage variations and aging," in *IEEE ISSCC*, Feb 2007, pp. 292–604.

[12] A. Kahng *et al.*, "Slack redistribution for graceful degradation under voltage overscaling," in *IEEE ASP-DAC*, Jan 2010, pp. 825–831.

[13] OpenRISC Community, "OpenRISC 1000 architecture manual."

[14] L. Benini *et al.*, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *IEEE DATE*, March 2012, pp. 983–987.

[15] The Embedded Microprocessor Benchmark Consortium (EEMBC), "CoreMark," http://www.eembc.org/coremark/.

[16] J. Pallister *et al.*, "BEEBS: open benchmarks for energy measurements on embedded platforms," *CoRR*, vol. abs/1308.5174, 2013.

[17] V. Chippa *et al.*, "Approximate computing: An integrated hardware approach," in *IEEE Asilomar*, Nov 2013, pp. 111–117.