九州大学学術情報リポジトリ
Kyushu University Institutional Repository

# Challenges in Evaluations for a Typical-Case Design Methodology

Kunitake, Yuji
Graguate School of ComputerScience and System Engineering, Kyushu Institute of Technology, Kyushu Institute of Technology

Chiyonobu, Akihiro
Graguate School of ComputerScience and System Engineering, Kyushu Institute of Technology, Kyushu Institute of Technology

Tanaka, Koichiro
Center for Microelectronic Systems, Kyushu Institute of Technology

Sato, Toshinori
System LSI Research Center, Kyushu University

https://hdl.handle.net/2324/6365

出版情報：Proc. of 8th International Symposium on Quality Electronic Design, pp.374-379, 2007-03-27. International Symposium on Computer Quality Electronic Design
バージョン：
権利関係：

# Challenges in Evaluations for a Typical-Case Design Methodology [§]

**Yuji Kunitake[+] Akihiro Chiyonobu[+] Koichiro Tanaka[+] Toshinori Sato***

**[+]Kyushu Institute of Technology *Kyushu University**

## Abstract

*According to the current trend of increasing variations in process technologies and thus in performance, the conservative worst-case design will not work since design margins can not be provided. We are investigating a typical-case design methodology, where designers focus on typical cases rather than on rarely-occurring worst cases. On evaluating the typical-case design, accurate circuit delay has to be considered, which is ignored in the current architectural-level simulations. While gate-level simulations consider circuit delay, they require huge amount of simulation time and hence are inappropriate for system designs, where designers examine a wide variety of design choices. In this paper, we show the challenges in evaluating designs that are based on the typical-case design methodology, and build a prototype architectural-level simulator, which can estimate circuit delay within tolerable simulation time.*

## 1. Introduction

As the complexity of the semiconductor manufacturing process increases, it is likely that process variations will be more difficult to control. The demand for low power leads supply voltage reduction and hence makes voltage variations a serious problem. Higher and higher clock frequency increases temperature variation in a chip. Under these situations, the deep submicron (DSM) semiconductor technologies will make the traditional worst-case design impossible, since they can not provide design margins that it requires. In order to realize robust designs, we have to design LSIs by considering typical cases rather than worst cases.

We are investigating such a typical-case design methodology, which we call Constructive Timing Violation (CTV) [10]. The CTV enables LSI designers to be free from very rare worst cases and to focus on typical cases. As its name explains, the CTV exploits variations in circuit delay. It is commonly observed that the delay for an individual operation of every logic circuit is generally much shorter than its critical path delay [6]. Input signals activating the critical path are limited to a few variations. Based on these observations, we expect that timing errors rarely occur even if timing constraints on the critical path are not satisfied. We can further optimize performance and power as the timing constraints are relaxed. This means that the CTV is based on a circuit-level speculation, and hence misspeculations (timing errors in this case) might occur, resulting in logic errors. The CTV provides a safety net for the rare situations.

We investigated to adopt the CTV to microprocessors design, made several designs of carry select adders (CSLA) [8, 11, 13], and evaluated the adders both at architectural-level and at gate-level. However, until now, our architectural-level simulations, which evaluate an entire microprocessor, do not consider actual circuit delay. On the other hand, our gate-level simulations considering circuit delay are performed only on a part of processor, the CTV-based CSLA. Moreover, the architectural- and gate-level simulations are conducted independently. We only estimated the usefulness of the CTV based on the two kinds of evaluations. In order to really understand how the CTV works, we have to evaluate the entire microprocessor with considering circuit delay. Since gate-level simulations on entire processors take huge amount of time, they are not appropriate for architectural-level design and hence a breakthrough in simulations is required. In this paper, we investigate an architectural-level simulator that considers circuit delay.

This paper is organized as follows. Section 2 introduces the CTV. Section 3 details an architectural-level simulator, which takes circuit delay into consideration. Section 4 presents experimental results. Section 5 reviews the related works. Finally, Section 6 concludes.

## 2. Typical-case design methodology

The DSM technologies increase variations, and hence design margins that the conventional worst-case design methodology requires, are reduced. The conservative approach will not work. Considering the situation, we have to change design methodology. Typical-case design methodology is a promising one. It exploits an observation that worst cases are rare. We should focus on typical cases rather than worst cases. Since we do not have to consider worst cases, design constraints are relaxed, resulting in easy designs.

In the typical-case design methodology, we adopt two methods to a design at a time. One is performance-oriented design, where only typical cases are considered. Since worst cases are not considered, design constraints are relaxed, resulting in easy designs. The other is function-guaranteed design. While it requires worst case considerations, designers doesn't have to consider performance but have to guarantee functions. Hence designs must be simple, resulting in easy verifications.

The CTV paradigm is such a design methodology, where designers are focusing on typical cases rather than are worrying about very rare worst cases. We will explain its details in the rest of this section.

### 2.1. Constructive timing violation

The CTV exploits an observation that the longest path for an individual operation of every logic circuit is generally much shorter than its critical path [6]. The CTV also utilizes the fact that input signals activating the critical path are limited to a few variations. In other words, timing errors rarely occur even if the timing constraints on the critical path are not satisfied. For example, it has been reported that nearly 80% of paths have delays of half the critical time [12]. The CTV relies on circuit-level speculation, and thus sometimes timing errors occur, resulting in possible logic errors. Some fault tolerance mechanisms are provided to recover circuits from timing errors.

The concept of the CTV is as follows. We design every timing critical function in a chip by two methods. The design consists of two components as shown in Figure 1. One is called main part, and the other is called checker part. While two parts share the single function, their roles and implementations are mutually different. The main part is designed with performance considerations, but might cause timing errors. That is, it is implemented by the performance-oriented design. The checker

[§] This work is partially supported by Grants-in-Aid for Scientific Research #16300019 and #176549 from Japan Society for the Promotion of Science.

part is a safety net for the main part. It detects timing errors that occur in the main part, and thus it has to satisfy all timing constrains in the chip. However, designers do not have to optimize performance nor power but only have to guarantee the function. That is, it is implemented by the function-guaranteed design. If a timing error is detected by the checker part, the circuit state has to be recovered to a safe point by any means.

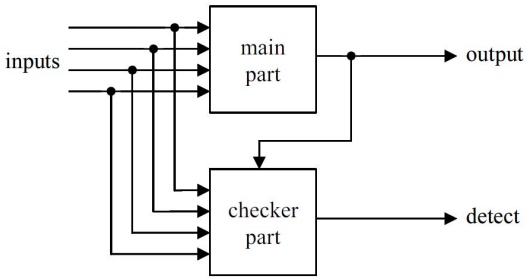


Figure 1 : Constructive timing violation

## 2.2. An example

We show an example using an ALU in microprocessors. Figure 2 depicts a CTV-based ALU. It is assumed that the clock frequency $f_L$ is considerably slow and thus safe, that means timing errors do not occur. Now we would like to increase the frequency up to $f_H$, where $f_L <= f_H < 2 * f_L$. First, the ALU is duplicated by three times. One ALU, which we call main ALU, works at $f_H$ and the remaining two ALUs, which we call checker ALUs, work at $f_L$. Thus, the checker ALUs do not cause timing errors and they keep up with the throughput of the original ALU to verify operations of the main ALU. The clock signals of the checker ALUs are provided complementary with each other, and hence they work alternatively to verify the main ALU. The verification is based on comparing two execution results from the main ALU and a corresponding one of the checker ALUs. If they do not match, a timing error is detected. In such cases, any recovery action should be initiated. Since the comparators should be error free, they will work at slower clock frequency of $f_L$.

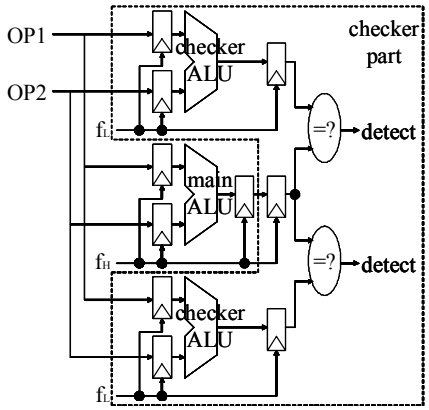


Figure 2: CTV-based ALU

In order to recover processor state to a safe point where the error is detected, we propose to utilize the recovery mechanism implemented in modern processors for speculative execution. In other words, timing errors are managed as if they were misspredicted branches. Thus, there is no hardware overhead in the recovery mechanism. After flushed, its pipeline executes instructions immediately following the erroneous instruction.

This is a very easy implementation of the CTV and is used here for easily understanding the concept. We have already tried to reduce its area and power overheads as well as to remove the complex clock distribution [8, 13].

## 2.3. Enhancements of CTV-based design

As we will see later, the CTV-based design severely diminishes processor performance when it combined with the recovery mechanism for handling misspredicted branches. In order to mitigate the performance loss, we propose two enhancements on the design. We guess just a small number of specific instructions are responsible for most timing errors. Every operation that once causes a timing error must cause the same error over and over again when the operation is performed with the same operands. To exploit the characteristic, we use result cache (RC) [9] proposed by Richardson. In addition, every instruction that once causes a timing error might cause the same error over and over again. To exploit the characteristic, we develop error history buffer (EHB). To the best of our knowledge, the use of histories for avoiding useless speculations has not been applied to typical-case based design.

The RC exploits computation redundancy, where some operation repeatedly does the same function because it repeatedly sees the same operands [9]. Thus, if the previous computation is kept in a table, the next same computation can be eliminated by looking up the table. Figure 3 depicts the RC studied in this paper. It resembles caches, and its each entry has a tag field and a data field. It is a direct-mapped table and indexed by a hash function of a combination of two operands. Every tag field contains the most significant bits (MSB) of the index. The associated data field contains the execution result when the operands are provided. Remember that the CTV is used only to adders in the example in the previous subsection. Hence, the RC keeps results of additions and subtracts that once cause timing errors. Later, when an instruction's operands are available, the RC is referred. When a match is found, a possible timing error is avoided by using the previous execution result provided by the RC.

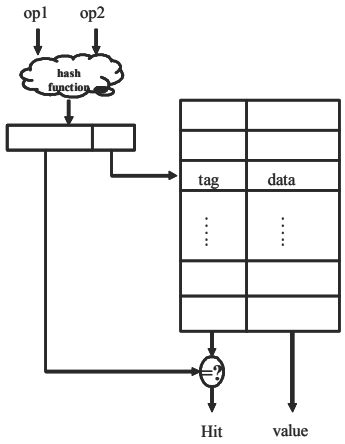


Figure 3: Result cache

Since we use the RC at execution stage, no performance gain is obtained for instructions with short latency. Instead, we exploit redundant computation for error avoidance. Thus, if we determine that the amount of time to access the cache-like structure is included in a single cycle, the processor does not suffer any penalties due to the possible timing error.

As shown in Figure 4, the EHB also has a direct-mapped cache-like structure and only has a tag field. Instruction address is used to index it, and its MSB is kept in the tag field. Instructions that cause a timing error are registered to the EHB. While the RC checks input operands for every instruction, the EHB does not.

Once an instruction is registered in the EHB, it is identified as an error-prone instruction regardless of its input operands. When there is a hit in the EHB, the result provided by the main ALU is dropped but that provided by one of the checker ALUs is used. Hence, in the case of EHB hit, ALU's execution latency is increased to 2 cycles, that is much smaller than the timing error penalty.

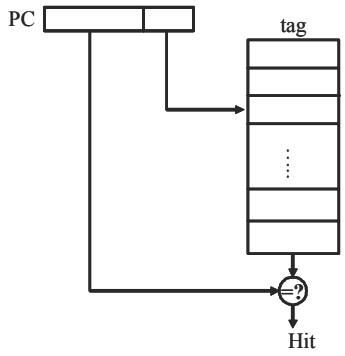


Figure 4: Error history buffer

## 2.4. Preliminary evaluation

We use MASE simulator [1], a derivative of SimpleScalar, to evaluate how dynamic timing errors affect processor performance. The processor we evaluate is a 4-way out-of-order (OoO) superscalar processor. We deliver two times faster clock to the main part than that to the checker part. Hence, if any timing errors are not detected, the processor enjoys two times higher performance. A 2-cycle latency is assumed to verify if a timing error occurs. When timing errors are detected, the pipeline flush mechanism for misspredicted branches is used to recover the processor state. The RC and the EHT have 64 and 512 entries, respectively.

We use 164.gzip, 175.vpr, 176.gcc, 197.parser, 255.vortex, and 256.bzip from SPEC2000 benchmark. We skip the first 1 billion instructions to avoid unrepresentative behavior at the beginning of the program's execution. Results are then reported for simulating each program for 100 million instructions.

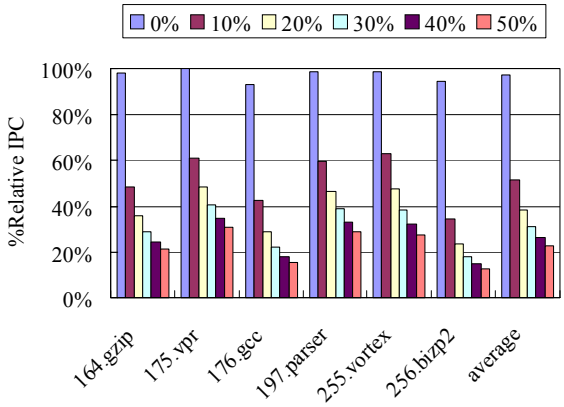


Figure 5: Impact of timing errors on IPC

We assume that dynamic timing errors randomly occur. We vary the error rate between 0% and 50%. Figure 5 presents processor performance in IPC (Instructions per cycle) relative to that of the baseline model. We observe timing errors have serious impact on performance even when the error rate is as small as 10%. The decrease in processor performance at the error rate of 0% is due to the verification latency of 2 cycles. This delays instruction commitment.

Figure 6 shows how the enhancement techniques mitigate performance loss. There are four bars for each program. All bars represent processor performance relative to that of the baseline model. The first one (see from left to right) is for the processor model utilizing the CTV. The next three are for those attached with the RC, the EHB, and the both, respectively. The error rate of 30% is assumed. We can see the followings. In most programs, the RC has little contribution to performance improvement. When we use both techniques, the performance gain mostly comes from the EHB. Only two programs, 197.paerser and 255.vortex, enjoy the synergetic effect of the RC and the EHT.

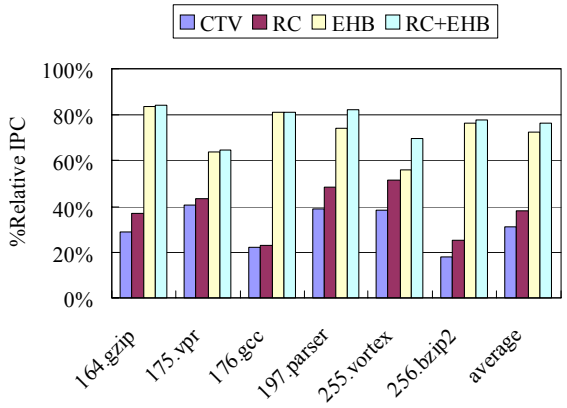


Figure 6: IPC improvements via RC and EHB

The simulation results only show the potential of the enhancement techniques. Furthermore, the performance loss due to the CTV is based on an assumption. These are because we inject dynamic timing errors at random. There is a possibility that this error model does not represent how timing errors actually occur. In order to validate the CTV and its enhancements, we have to consider timing errors considering actual circuit delay.

## 3. Delay-aware architectural simulator

In order to evaluate CTV-based designs, accurate circuit delay has to be considered. On system designs, architectural-level simulators are a must-be tool, which makes designers examine a wide variety of design choices. Unfortunately, current architectural-level simulators do not consider circuit delay and are inappropriate to evaluate CTV-based designs. In contrast, gate-level simulators can estimate accurate circuit delay. However, gate-level simulations require huge amount of simulation time, and hence are inadequate for system design explorations. For example, we saw one order of magnitude speed gap between the gate and architectural levels, when we simulated a CSLA, which we will evaluate in the following section. Considering the current situations, we have to build a fast delay-aware simulator. While it is basically an architectural-level simulator, it can consider delay in timing-critical blocks. We will explain how delay estimation becomes possible without exploding simulation time.

A delay-aware simulation consists of four steps: it requires four simulations at different levels as follows.

1. Function-level simulation: typical phases in the program execution are identified at this level.
2. Architectural-level simulation: a stimulus for the following simulations is generated at this level.
3. Gate-level simulation: delay information in timing-critical blocks is extracted at this level.

4. Architectural-level simulation: a fast delay-aware simulation is performed at this level.

In the following subsections, we will describe these steps one by one. Note that it is assumed that designers have knowledge of timing-critical blocks a priori.

### 3.1. Identifying typical execution phases

First, the typical phase that represents the full execution of the program is identified. System design explorations include numerous evaluations on processor configurations for a variety of workloads. They are usually conducted through architectural-level simulations rather than gate-level simulations, because the latter ones require huge amount of simulation time. In order to reduce simulation time, we only simulate the typical phase that represents the full execution of the program. SimPoint [5] is a tool that dramatically reduces the number of instructions simulated to characterize a program's behavior by identifying phases for the program. By performing a functional-level simulation, the representative execution phase is identified with the help of SimPoint analysis.

### 3.2. Generating simulation stimulus

Second, a stimulus for the gate-level simulation is generated. The stimulus should be compact enough for the gate-level simulation to be performed within tolerable time. After the representative phases are identified, an architectural-level simulation is performed to generate the stimulus for every timing-critical block. The reason why we conduct architectural-level simulations rather than functional-level simulations is as follows. Even if we simulate the identical program phase at architectural- and functional-level simulations respectively, executed instructions are different since speculative execution is performed only at architectural-level simulations. Architectural-level simulations execute more instructions than the functional-level simulations do, and hence the stimulus generated by the former ones may have more erroneous instructions than the latter ones do.

### 3.3. Extracting delay information

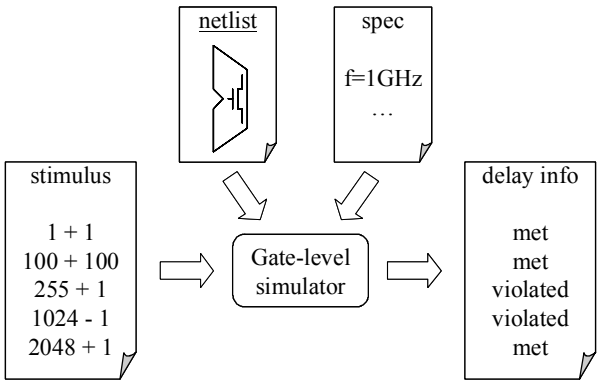


Figure 7: Delay information extraction

Third, circuit delay information is extracted. The information extracted from the gate-level simulation is registered into a table in the architectural-level simulator. The table is created for each timing-critical block, and maps the set of input signals of the block into the circuit delay of the block.

For each timing-critical block, the gate-level simulator uses its netlist and the stimulus generated at the previous step, and reports its delay information. Since just one block is picked up and the stimulus is created compact, the simulation time will be within tolerable range. Figure 7 explains how the gate-level simulator works. In the figure, an adder is used as a representative timing-critical block. This process is repeated according to the number of the timing-critical brocks.

### 3.4. Performing delay-aware simulation

And last, another architectural-level simulation is performed using the delay information extracted from the gate-level simulations. In order to include the delay information, delay information tables are created and embedded into the architectural simulator. For each timing-critical block, its corresponding table maps the set of input signals of the block into the circuit delay of the block. In order to reduce the time required for reference to the tables, only erroneous stimulus is recorded.

The architectural-level simulation should be conducted just for the typical phases identified at the functional-level simulation. This is because we have the delay information only for these phases. They summarize total program behavior and hence we can get enough accuracy for system design exploration even if we consider circuit delay only during the typical phase. Every cycle the architectural simulator refers the tables for every active block. The table informs whether the block satisfies its timing constraints. If a timing error is found, the architectural simulator responds to the error and appropriate events are initiated.

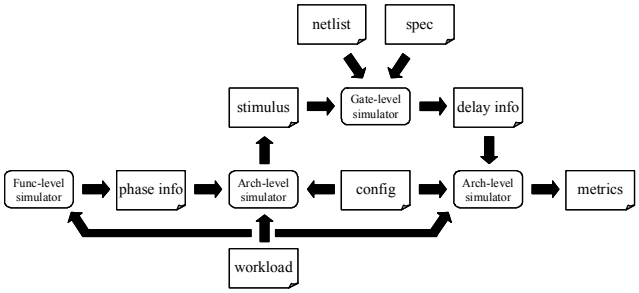


Figure 8: Simulation flow

Figure 8 surveys the whole simulation flow. When a workload is provided, the functional-level simulations identify typical phases of the program execution and the architectural-level simulations generate a stimulus for each phase. The stimulus is loaded into the gate-level simulator, which generates the delay information for every timing-critical block. The information is used by the architectural-level simulator to consider circuit delay, and finally we have some metrics for evaluating the design.

## 4. Experimental results

In order to illustrate the capabilities of the delay-aware architectural-level simulation environment, we evaluate the CTV-based adder used in Section 2.2. That is, we assume that only adders are timing-critical blocks in the microprocessor used in the experiments. We use six programs from SPEC2000 as workloads, which are explained in Section 2.4.

### 4.1. Building architectural-level simulator

First, we use SimPoint [5] to pick representative 10,000 instructions for each program. The SimpleSclar functional-level simulator [1] is used. The reason why we execute such a small number of instructions is that we have to perform gate-level simulations in the next step. Since we use an adder for the experiments, only additions and subtracts are considered.

Second, we generate a stimulus for the gate-level simulations by using the MASE simulator [1].

Third, we perform gate-level simulations. In this step, a netlist of the adder is required. We design a 32b CSLA shown in Figure

9, using Verilog-HDL. It consists of 13 ripple carry adders (RCAs) with four different bit widths. We logic-synthesize it using SYNOPSIS DesignCompiler with Hitachi 0.18u standard cell library. DesignCompiler reports the critical path delay, and we use the clock two times faster than that meets the critical path delay. This is the specification on clock frequency, and is provided to the simulator. We use Cadence Verilog-XL simulator.

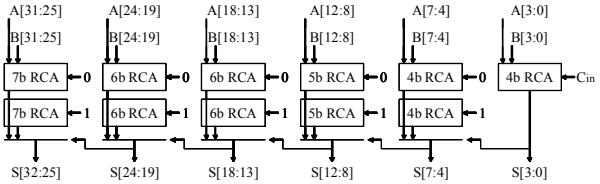

Figure 9: Carry select adder

And last, we built our delay-aware architectural-level simulator. We modify the MASE to include the delay information extracted from the logic-level simulations. The simulator loads the delay information and creates the delay information table for the CSLA. In the experiments, only one table is enough. After that, we conduct architectural-level simulations. The processor model explained in Section 2.4 is used. It is a 4-way OoO superscalar and its clock frequency is boosted by two times faster than that determined by the critical path delay. When timing errors are detected, the pipeline flush mechanism for misspredicted branches is used to recover the processor state. We simulate the same simulation points identified in the first step.

## 4.2. Validating architectural-level simulator

First, we compare timing error rates between at gate level and at architectural level. Figure 10 presents error rates at the both levels. For each group of two bars, the left one indicates the error rate at the gate level and the right one at the architectural level. We observe considerable discrepancies for all programs. Architectural-level simulations show approximately 20% larger error rate.

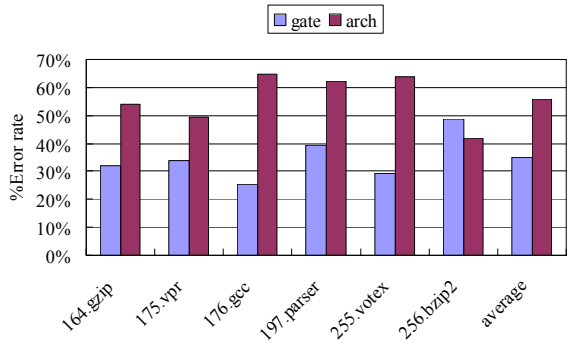

Figure 10: Timing error rates

We compare in details the erroneous instructions at architectural level with those at gate level. We find that a significant number of errorless instructions at gate level cause timing errors at architectural level. Table 1 presents the percentage of instructions that cause timing errors at architectural level but do not at gate level. Almost half of erroneous instructions at architectural level are errorless at gate level. Again we check in details the gate-level simulation results, and find that each erroneous operation does not always cause a timing error every time even if it is executed with the same input operands. Whether the current operation really causes a timing error depends on the preceding operation result. When the preceding and current operation results are similar, the erroneous operation does not always cause an error. This situation never happens at architectural-level simulations, where every erroneous operation always causes an error. We find that this is the reason why there are considerable discrepancies in timing error rates between at gate level and at architectural level.

Table 1: Difference in errors

| program | % difference |
|---|---|
| 164.gzip | 46.0 |
| 175.vpr | 46.1 |
| 176.gcc | 60.2 |
| 197.parser | 52.2 |
| 255.vortex | 58.5 |
| 256.bzip2 | 35.0 |

From the observations and the considerations above, we find the current simulation environment has some challenges to be solved, as summarized below.

1. The architectural-level simulator can not consider the dependence of timing errors upon the immediately preceding operation results, and hence will overestimate timing error rate.
2. The gate-level simulator can not consider the change in execution order of instructions, which will happen at architectural-level simulations, and hence might underestimate timing error rate.
3. Wrong execution results due to timing errors are not used as input operands, and hence the gate-level simulator might miss possible chained timing errors.

The last one is not a problem in this case for estimating performance at architectural level. When the first error is detected, the pipeline and hence the succeeding errors are flushed, and therefore the succeeding errors do not have any impact on performance.

Since our current architectural simulator has the limitations above, it overestimates timing errors.

## 4.3. Evaluating CTV with delay considerations

Next, we demonstrate the importance of the actual delay consideration. We compare the delay-aware simulator with the one used in Section 2.4, where timing errors randomly occur. At the random-based simulations, for each program, we use the timing error rate in Figure 10, which we find at the delay-aware architectural-level simulations. Thus, both simulations show the identical error rate.

Figure 11 presents processor performance relative to that of the baseline model, when the CTV is utilized. Surprisingly, we do not see substantial discrepancies between two simulation results. Actual circuit delay does not have impact on the total processor performance.

Figure 12 presents the simulation results when a 64-entry RC is attached. Different from the results shown in Figure 11, there are noticeable discrepancies between the delay-aware and the random-based simulations. In addition, we find an interesting observation. Performance is always higher at the delay-aware simulations than at the random-based one. These observations confirm our guess in Section 2.3 that actually every erroneous operation must cause the same error over and over again, since the RC exploits redundant computation.

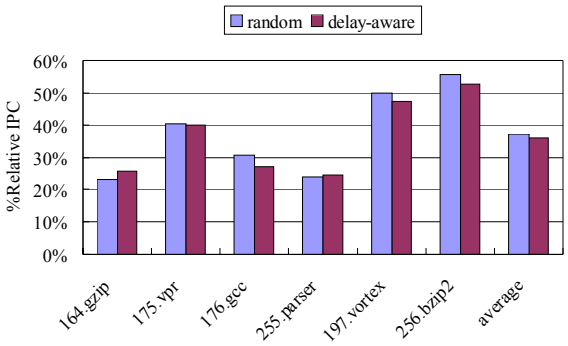


Figure 11: Delay-aware vs. random (CTV)

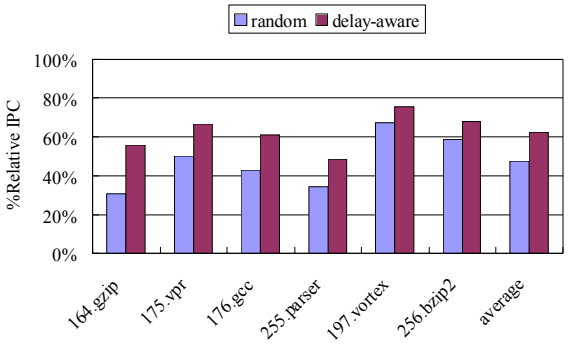


Figure 12: Delay-aware vs. random (CTV+RC)

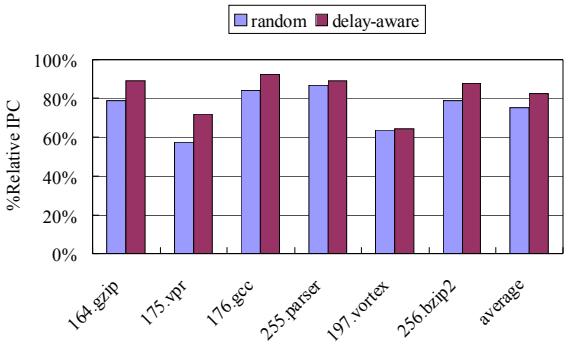


Figure 13: Delay-aware vs. random (CTV+EHT)

Figure 13 presents the results when a 512-entry EHT is utilized. Once again, different results from Figure 11 are observed and the tendency is same to Figure 12.

The simulation results shown in Figures 12 and 13 tell us that it is important to consider accurate circuit delay on evaluating the typical-case designs.

## 5. Related works

Architectural-level simulation is very popular in system design. SimpleScalar [1] is one of the most widely-used architectural-level simulators. Most architectural-level simulators including SimpleScalar only report cycle-level performance metrics.

Wattch [2] is an extension of SimpleScalar, and has an ability to evaluate power consumption. While it calculates some circuit delay, it does not consider any interactions between circuit delay and architecture. Wattch and other power simulators can not evaluate microarchitectures that are adaptable to circuit-level phenomena.

Razor [4] has a similarity to the CTV, while they are developed independently of each other. Razor relies on shadow FFs, where a delayed clock is distributed, to hold correct values. An open question is how race conditions are avoided. Lee et al. [7] build a circuit-aware simulation methodology for Razor. The methodology is also similar to ours. We compared results from the gate- and architectural-level simulations in the present paper, while they only show results obtained from their architectural-level simulator.

Liberty-Justice [3] is a simulation infrastructure that allows architects to develop simulators, which models wire delay and area as well as performance and power. It considers only interconnection delay, but does not consider variance in block delay due to its input signals. Adaptation of microarchitecture to circuit-level activity is not modeled in the infrastructure.

## 6. Conclusions

Typical-case design methodology is an attractive design methodology to answer problems emerging in the DSM era. Unfortunately, the traditional architecture-level simulators are not accurate enough to evaluate typical-case designs, since they ignore circuit delay. In this paper, we discussed how to build delay-aware simulators, and demonstrated the importance of circuit delay considerations. We implemented a delay-aware architectural simulator for evaluating the CTV, and found there are considerable discrepancies in performance results between at the delay-aware simulation and at the random-based simulation.

Currently, there are still some challenges in accurately estimating circuit delay. We are investigating these challenges in the future studies.

## References

[1] T. Austin et al.: SimpleScalar: an infrastructure for computer system modeling, IEEE Computer, 35(2), 2002.
[2] D. Brooks et al.: Wattch: a framework for architectural-level power analysis and optimizations, ISCA, 2000.
[3] N. Cater et al.: Modeling wire delay, area, power and performance in a simulation infrastructure, IBM Jour. Res.&Dev., 50(2/3), 2006.
[4] D. Ernst et al.: Razor: a low-power pipeline based on circuit-level timing speculation, MICRO, 2003.
[5] G. Hamerly et al.: SimPoint 3.0: faster and more flexible program analysis, Workshop on MoBS, 2005.
[6] T. Kuroda: 10 Tips for low power CMOS design, DAC, 2003.
[7] S. Lee, et al.: Circuit-aware architectural simulation, DAC, 2004.
[8] K. Mima et al.: Hardware cost reduction in fault detection mechanism for constructive timing violation technique, ISIC, 2004.
[9] S. Richardson: Caching function results: faster arithmetic by avoiding unnecessary computation, Technical Report TR-92-1, Sun Microsystems Lab., 1992.
[10] T. Sato et al.: Constructive timing violation for improving energy efficiency, L. Benini et al. eds. "Compilers and operating systems for low power", Kluwer, 2003.
[11] A. Tanino et al.: Simplifying high-frequency microprocessor design via timing constraint speculation, CAINE, 2003.
[12] K. Usami et al.: Automated low-power technique exploiting multiple supply voltages applied to a media processor, JSSC, 33(3), 1998.
[13] M. Yamahara et al.: A fast fault detection circuit for low-power adders with timing error tolerance, IPSJ Trans., 47(SIG18), 2006.