**Regular Paper**

# Dependence Graph Model for Accurate Critical Path Analysis on Out-of-Order Processors

Teruo Tanimoto[1,a] Takatsugu Ono[2,b] Koji Inoue[2,c]

Received: April 21, 2017, Accepted: August 27, 2017

***Abstract:*** The dependence graph model of out-of-order (OoO) instruction execution is a powerful representation used for the critical path analysis. However, most, if not all, of the previous models are out-of-date and lack enough detail to model modern OoO processors, or are too specific and complicated which limit their generality and applicability. In this paper, we propose an enhanced dependence graph model which remains simple but greatly improves the accuracy over prior models. The evaluation results using the gem5 simulator with configurations similar to Intel's Haswell and Silvermont architecture show that the proposed enhanced model achieves CPI errors of 2.1% and 4.4% which are 90.3% and 77.1% improvements from the state-of-the-art model.

***Keywords:*** dependence graph model, critical path analysis

## 1. Introduction

Out-of-order (OoO) instruction execution is one of the key paradigms that has improved processor performance over the past decades. To improve the performance or energy efficiency, it is important to understand dominant instructions that determine the execution time of the program, which are called *critical instructions*. Generally, identifying critical instructions and their criticality accurately is difficult because multiple operations are reordered and executed in parallel on OoO processors.

Critical path analysis of OoO processors is a useful tool to solve the problem and enable a number of performance and energy optimizations. For example, critical path analysis can be used to turn on/off costly speculative executions [10], adaptively schedule non-critical instructions [23], establish metrics such as *slack* [9] or *interaction cost* [11] for deep understanding of microarchitectural bottlenecks, or for processor design space explorations [17]. These applications are all based on dependence graph representations of OoO instruction execution which model the dependencies between specific events of each dynamic instruction.

Although existing dependence graph models are shown to work well on relatively simple OoO processors (e.g., the *sim-outorder* model of the SimpleScalar simulator [5]), they are too simple and lack the details which are critical to performance for modern OoO processors. An attempt to heavily customize the model for specific microarchitectures has been made, but it lacks generality and cannot be widely applied to other microarchitectures [17]. In this paper, we propose an enhanced dependence graph model which achieves the higher accuracy than existing models and also maintains its generality at the same time. To be specific, we implement and perform accuracy comparisons against existing models using the O3CPU model of the gem5 simulator [4].

Our study makes the following contributions [*1]:

- **Enhanced dependence graph model for modern OoO processors.** We improve the dependence graph model based on two key microarchitectural details that have not been taken into account in previous models: dynamic variation of branch misprediction penalty and a modern SQ (store queue) design which combines the SQ and the writeback buffer (WBB).
- **A gem5-specific modification to the enhanced dependence graph model.** We further modify the dependence graph model to incorporate gem5's implementation-specific frontend delays which occur when the frontend stages are (blocked and then) unblocked.
- **Accuracy evaluations.** We evaluated the CPI (cycles per instruction) error of the enhanced dependence graph model with two processor configurations. With configurations similar to Intel's Haswell and Silvermont processor, the CPI (cycles per instruction) errors are 5.3% and 12.2% without the gem5-specific modification, which are 76.0% and 49.1% improvements over the existing model, respectively; CPI errors further reduce to 2.1% and 4.4% with the gem5-specific modification, respectively.

In Section 2, we discuss the reason why the dependency graph

[1] Graduate School of Information Science and Electrical Engineering, Kyushu University, Fukuoka 819–0395, Japan
[2] Faculty of Information Science and Electrical Engineering, Kyushu University, Fukuoka 819–0395, Japan
[a] teruo.tanimoto@cpc.ait.kyushu-u.ac.jp
[b] takatsugu.ono@cpc.ait.kyushu-u.ac.jp
[c] inoue@ait.kyushu-u.ac.jp

[*1] This paper is an extended version of the paper [25]. In this paper, the modifications of the dependence model are described in more detail, and accuracy evaluations with multiple configurations are conducted. It is important to show that the proposed model can apply to not only the specific configuration but also others.

model needs to be enhanced. In Section 3, we describe our proposal, the enhanced dependency graph model, and in Section 4 gem5-specific modification to the model is explained. Section 5 evaluates the accuracy of our models. Section 6 summarizes related work. Section 7 draws conclusions.

## 2. Background and Motivation

The dependence graph model of OoO instruction execution is a well-known representation for enabling critical path analysis. The graph is a weighted directed acyclic graph where each node represents an event (e.g., dispatch) of each dynamic instruction, and each edge which is weighted by the latency in terms of clock cycles models various data and resource dependencies between these events during the actual execution. The weight of each edge is the number of cycles to resolve the constraint that the edge represents.

We choose the dependence graph model proposed in the literature [11], [12] as the baseline model, which we call the Fields's model, since it is simple (five node types and 12 edge types) yet powerful enough to capture the critical microarchitectural bottlenecks of OoO processors. **Table 1** shows the definition of the Fields's model, which represents each dynamic instruction with five nodes of *(D)ispatch*, *(R)eady*, *(E)xecute*, *com(P)lete* and *(C)ommit*. The execution trace is necessary to construct the dependence graph because whether *PD*, *PR* and *PP* edges connect nodes or not is conditional on the dynamic events (such as branch misprediction and data dependency) on the processor and the weights of *DD*, *RE*, *EP* and *CC* edges are determined according to the trace.

**Figure 1** shows an example of the dependence graph that is generated from Fields's model. It assumes a processor, whose pipeline width is two and the number of ROB (reorder buffer) entries is four, executes ten instructions on the left. Solid edges are lastly arriving edges to a node, which are candidates of the critical path; thick edges consist of the critical path; dashed edges are non-critical edges. $i_0$ conducts a division whose execution takes a long latency of five cycles, which is expressed by the edge from $E_0$ to $P_0$. $i_2$, $i_6$ and $i_9$ are load instructions, whose *EP* edges' weights include the cycles calculating load addresses and accessing caches and main memory. $i_4$ and $i_8$ are branch instructions. $i_8$ is a mispredicted branch and has a *PD* edge, which correctly predicted $i_4$ doesn't have. The thick edge from $C_3$ to $D_7$ is *CD* edge.

Each node has at least one last arriving edge which determines the node's time from the start node. The critical path is defined as the longest path from the start node to the end node and is iden-

**Table 1** The definition of Fields's model [11], [12].

| Name | Constraint | Edge | Definition | Latency |
|---|---|---|---|---|
| *DD* | In-order dispatch | $D_{i-1} \rightarrow D_i$ | | icache misses, itlb misses |
| *FBW* | Finite fetch bandwidth | $D_{i-fbw} \rightarrow D_i$ | where $fbw$ is the maximum no. of instructions fetched in a cycle | constant latency (1 cycle) |
| *CD* | Finite re-order buffer | $C_{i-w} \rightarrow D_i$ | $w$ = size of the re-order buffer | constant latency (0 cycle) |
| *PD* | Control dependence | $P_{i-1} \rightarrow D_i$ | inserted if $i-1$ is a mispredicted branch | constant branch recovery latency |
| *DR* | Execution follows dispatch | $D_i \rightarrow R_i$ | | constant pipeline latency |
| *PR* | Data dependences | $P_j \rightarrow R_i$ | inserted if instruction j produces an operand of i | constant latency (0 cycle) |
| *RE* | Execute after ready | $R_i \rightarrow E_i$ | | functional unit contention |
| *EP* | Complete after execute | $E_i \rightarrow P_i$ | | execution latency |
| *PP* | Cache-line sharing | $P_j \rightarrow P_i$ | inserted if instruction j produces cache miss to block loaded by i | constant latency (0 cycle) |
| *PC* | Commit follows completion | $P_i \rightarrow C_i$ | | constant pipeline latency |
| *CC* | In-order commit | $C_{i-1} \rightarrow C_i$ | | store BW contention |
| *CBW* | Finite commit bandwidth | $C_{i-cbw} \rightarrow C_i$ | where $cbw$ is the maximum no. of instructions committed in a cycle | constant latency (1 cycle) |

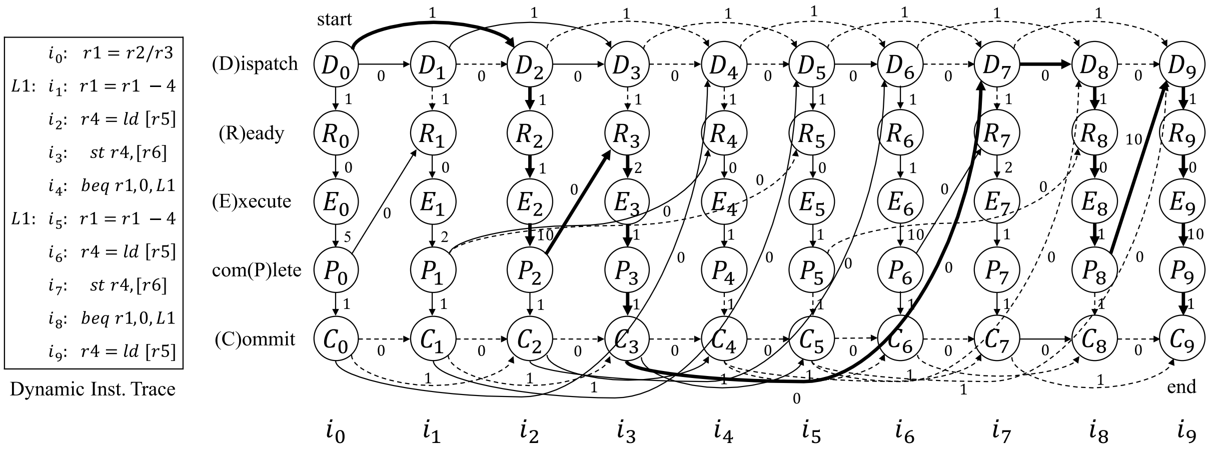

```
    i0:  r1 = r2/r3
L1: i1:  r1 = r1 - 4
    i2:  r4 = ld [r5]
    i3:  st r4,[r6]
    i4:  beq r1,0,L1
L1: i5:  r1 = r1 - 4
    i6:  r4 = ld [r5]
    i7:  st r4,[r6]
    i8:  beq r1,0,L1
    i9:  r4 = ld [r5]
```

**Fig. 1** An example of the dependence graph generated from Fields's model – Ten instructions on the left side are executed on the processor whose pipline width is two.

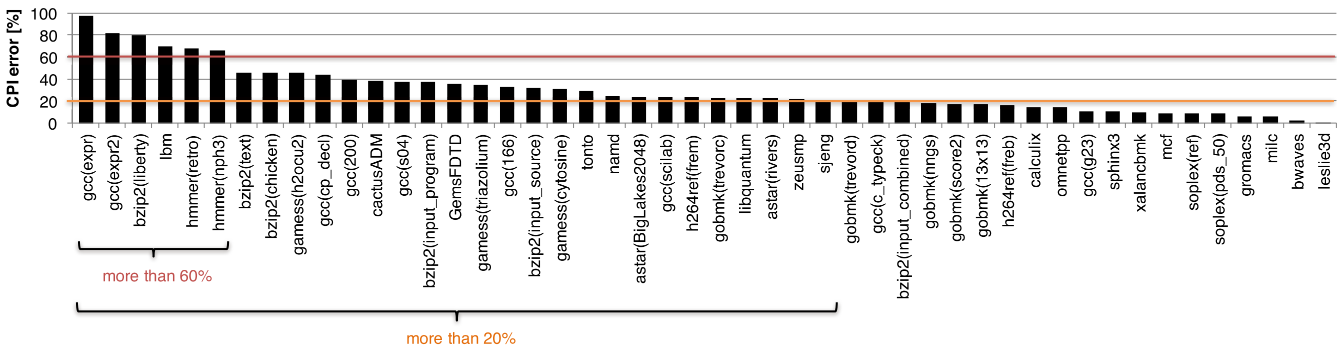


**Fig. 2** CPI error of the estimated critical path cycles using the Fields's dependence graph model.

**Table 2** Modifications applied to the Fields's model. The newly added constraints are denoted with $^+$ with their names.

| Name | Constraint | Edge | Definition | Latency |
|---|---|---|---|---|
| **Dynamic variation of branch misprediction penalty (described in §3.2)** | | | | |
| *PD* | Control flow | $P_{i-1} \rightarrow D_i$ | *i*-1 is a mispredicted branch | branch misprediction penalty (**dynamically decided** by Equation (4) for gem5) |
| **SQ conditions (described in §3.3, the latencies for the processors with SSQ such as O3CPU of gem5)** | | | | |
| $CS^+$ | Writeback after commit | $C_i \rightarrow S_i$ | | writeback latency |
| $SS^+$ | In-order writeback | $S_j \rightarrow S_i$ | *j* and *i* are consecutive stores | constant latency (0 cycle) |
| $SD^+$ | Finite SQ entries | $S_j \rightarrow D_i$ | *j* and *i* are stores; *i* is the *N*th store after *j* where *N* is number of SQ entries | constant latency (0 cycle) |
| *CC* | In-order commit | $C_{i-1} \rightarrow C_i$ | | constant latency (0 cycle) |
| **gem5 specific modification (described in §4)** | | | | |
| *DD* | In-order dispatch | $D_{i-1} \rightarrow D_i$ | | dynamically decided by Equation (5) |

tified by backtracking last arriving edges from the end node to the start node. The sum of the weights of the critical path is the estimated execution time calculated from the dependence graph model.

Here, we evaluate the accuracy of the Fields's model against the O3CPU model of the gem5 simulator with the SPEC CPU2006 benchmark suite [14]. The simulator is modified to generate the dynamic instruction trace required to build the dependence graph model. The configuration of the simulator is Config-A described in Section 5.1. **Figure 2** shows the CPI errors of the Fields's model. CPI error is the percent error of the estimated CPI using the model, which is based on the longest path (i.e., critical path) of the dependence graph, against the actual CPI obtained from cycle-accurate event-driven simulations. It is calculated by Eq. (1).

$$CPI\ error\ [\%] = \frac{|Simulated\ CPI - Estimated\ CPI|}{Simulated\ CPI} \times 100 \quad (1)$$

The CPI errors are greater than 20% in 29 out of 48 benchmarks and greater than 60% in six benchmarks. This clearly suggests that the Fields's model lacks some critical dependencies which happen to be the microarchitectural bottleneck of the execution. Our goal is to close this gap between the simulator and the dependence graph model, while at the same time without losing generality of the original Fields's model so that it can apply to a large set of OoO processors.

## 3. Enhanced Dependence Graph Model

### 3.1 Overview

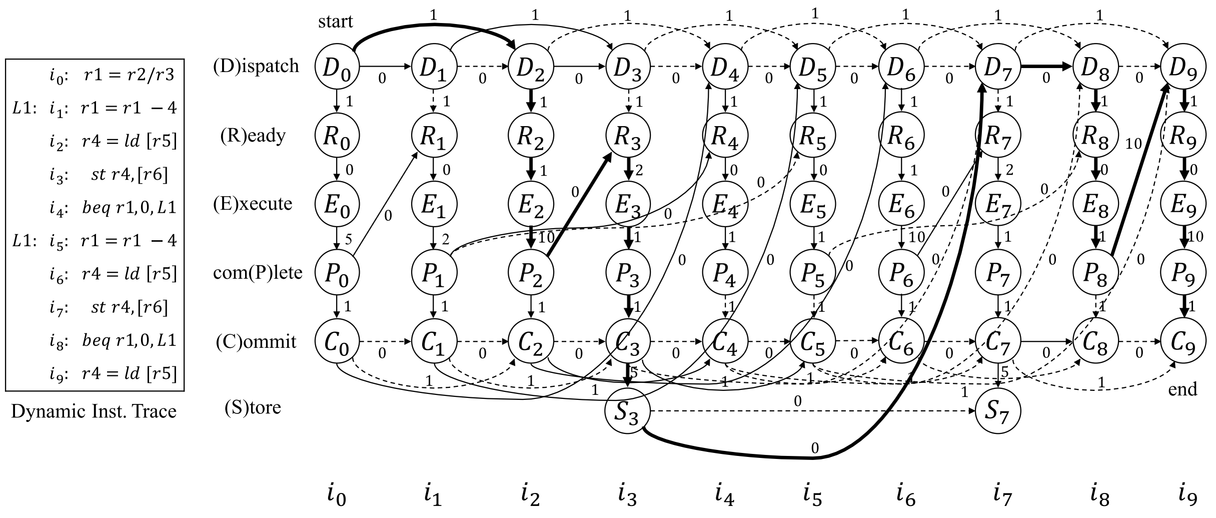
We performed a detailed analysis of the results shown in Fig. 2. The analysis required comparing the actual cycle of the simulator against the estimated cycle from the model for each stage of each dynamic instruction to identify in what circumstance the two diverge, like debugging using single-stepping in GDB. We identified two major events which are sources of CPI errors that are common to the majority of the programs: branch misprediction and SQ (store queue) full. We discuss how to incorporate the two events into the model in detail in the following subsections.

The gap between the processor and the dependence graph model in relation to these events comes from microarchitectural design choices. The aim of enhancements in this section is not just to adjust the dependence graph model to a specific processor model but to enlarge processor designs which the dependence model matches accurately. Therefore, the enhancements are applicable to processor models that adapt the same/similar design choices.

The original Fields's model defines five nodes per dynamic instruction and 12 edges between the nodes. The proposed model adds only one node, $S$, and three new edges, $CS$, $SS$ and $SD$, (and modifies the weights of three edges) to the Fields's model whose simplicity we believe is still maintained. **Table 2** summarizes the differences between the proposed model and the Fields's model. **Figure 3** shows an example of the dependence graph generated by the proposed model. $i_3$ and $i_7$ are store instructions and have $S$ node. Compared with Fig. 1, Fig. 3 has a different critical path.

### 3.2 Dynamic Variation of Branch Misprediction Penalty

In the Fields's model, the weight of the *PD* edge is supposed to be the branch misprediction penalty which is defined as the frontend latency (the frontend pipeline stage length including the fetch, decode and rename) [11]. This definition matches designs where mispredictions cause the fixed pipeline stall latency



```
     i0:  r1 = r2/r3
L1:  i1:  r1 = r1 - 4
     i2:  r4 = ld [r5]
     i3:  st r4,[r6]
     i4:  beq r1,0,L1
L1:  i5:  r1 = r1 - 4
     i6:  r4 = ld [r5]
     i7:  st r4,[r6]
     i8:  beq r1,0,L1
     i9:  r4 = ld [r5]
```

**Fig. 3** An example of the dependence graph generated from the proposed model.

such as checkpointing, which is adopted in simulators such as Multi2Sim [26], Sniper [7] and ZSim [22].

The problem with this approach is that the branch misprediction penalty can be substantially larger than the frontend pipeline length [8]. Upon a branch misprediction, there are several operations that happen in parallel: (**a**) fetch the correct path instructions and re-fill the pipeline and (**b**) recover the rename map table state (e.g., some recovery mechanisms require that all the instructions prior to the mispredicted branch have to be retired before ranaming the next instruction in the correct path). There are two reasons why the penalty can vary. First, until operation (b) completes and the rename map table is recovered, the correct path instructions cannot be renamed. Therefore, when the operation (b) does not finish by the time when the instructions fetched by the operation (a) reaches the rename stage, the branch misprediction latency becomes longer than the frontend pipeline length. Second, in the first case we assume that the instructions on the correct path can be fetched without additional delays, but in reality, an instruction cache/TLB miss can occur.

Since the Fields's model considers the branch misprediction latency to be always the frontend pipeline length, it assumes that the instruction cache/TLB miss never occurs, and the rename map recovery always finishes on time. We modify the weight of the *PD* edge to be determined dynamically to account for the variation of the actual penalty. We believe the rename map recovery mechanism used in gem5 is explained as "using the frontend map table and a history buffer (HBMAP+WALK)" elsewhere [2]. This scheme incrementally restores the rename map table by using a history buffer that keeps the speculative table updates. It starts with the current rename map table and walks from the tail of the ROB towards the mispredicted branch, recovering the rename map table state of each instruction. Therefore, the enhanced weight definition *Weight* of *PD* edge is as follows (additional terms in **bold**).

$$\begin{aligned}Weight_1 &= \boldsymbol{ICache\ and/or\ ITLB\ miss\ latency} \\ &\quad + Fetch\ latency + Decode\ latency \qquad (2)\\ Weight_2 &= \boldsymbol{ROB\ walk\ latency} \qquad (3)\\ Weight &= max(Weight_1, Weight_2) + Rename\ latency \qquad (4)\end{aligned}$$

There are other recovery designs proposed in Ref. [2] where the weight of the *PD* edge can be defined in a similar manner. For example, the detailed processor model of PTLsim [28], which is derived to MARSSx86 [19], also implements ROB walk to reconstruct the speculative rename map table.

### 3.3 Modern Store Queue Design

Another disparity we found between the Fields's model and the simulator is the lack of the modeling of stalls due to the SQ full event. This originates from the inability of the Fields's model to express the modern SQ design. An advanced implementation employs a combined SQ and a WBB (writeback buffer), which is sometimes called the senior store queue (SSQ) organization, and has been implemented in not only cycle-accurate processor simulators like Zesto [18] and gem5 but also in commercial processors such as the Alpha 21264 [16] and Intel's P6 microarchitecture [24].

The problem of applying the Fields's model to the SSQ design boils down to the fact that it cannot express the important behavior of store instructions. In an SSQ design, when a store "commits" it leaves the ROB but still stays in the physical SQ, and when it "retires" it gets written back to the cache and leaves the SQ. If the *C* node models "commit," the graph will miss all the cycles spent in the memory system which are actual bottlenecks of the execution when the processor stalls due to SQ full. On the other hand, if the *C* node expresses "retire," the effective ROB size in the graph will be smaller than the actual size since the instructions that have already left the ROB are not properly modeled. We chose the former for the evaluation in Fig. 2 where the high CPI error benchmarks suffer from stalls due to SQ full.

In order to handle this problem, we introduce a new node only for store instructions, *(S)tore-retirement*, to represent the retire event of store instructions. Also, we add three new edges *CS*, *SS* and *SD* that connect with this node. *CS* is an intra instruction edge which ensures the store retirement happens at the same time or after the commit. *SS* is an edge from the *S* node of a *store* instruction to the *S* node of its next *store* instruction, which models the in-order writeback to the cache. *SD* models the finite SQ size. It is an edge from the *S* node of a *store* instruction *s* to the

*D* node of the *N*th *store* instruction from *s*, where *N* is the number of SQ entries. In addition, the weights of all the *CC* edges are changed to zero since *SS* edges represent the additional latency of the writeback. This extended model is not only capable of representing the SSQ organization but it can also express an SQ with a separate WBB by changing the weights of *CS*, *SS* and *SD* to zero.

## 4. Gem5-Specific Modification

We believe that the main causes of the inaccuracies of the Fields's model have been ruled out by the enhancements proposed in the previous section, however, we have found that there still remains behavioral mismatch between the enhanced model and the cycle-accurate event-driven simulation. Unlike the enhancements in the previous section this disparity is specifically due to gem5's implementation where we describe the details below.

The frontend stages of the gem5 consist of fetch, decode and rename, where each stage is pipelined and its stage length can be given as a parameter. Ideally one might expect to have queues in between stages to absorb instructions inflight in case of a blocking event, where the implementation of gem5 is conservative in terms of performance [20].

For example, when the rename stage (blocks and then) unblocks, there will be pipeline bubbles equal to `decodeToRenameDelay` + `fetchToDecodeDelay` in total in the worst case. These are parameters in gem5 where the former defines cycles between when rename stage asks the decode stage to begin sending instructions again and when the first instruction arrives the rename stage, and the latter defines the equivalent for decode and fetch stages. **Figure 4** shows an example of how additional bubbles are inserted where the latencies of the fetch stage and decode stage are 3 and 2, respectively. The enhanced dependence graph model assumes perfect queuing where these bubbles do not exist, whereas 5 (3+2) cycles [*2] of bubbles are inserted in Fig. 4. The fundamental reason of the additional bubbles is that each frontend stage is implemented to be blocked until all the inflight instructions reach the latter stage once it is blocked.

In order to account for this implementation detail, we modify the weight of the *DD* edge which models the delay of frontend stages. Although modeling of frontend stages explicitly is one possible approach, we avoid it to keep our model simple. The *DD* edge originally models the frontend delays including instruction cache and/or TLB misses, where we add the additional bubbles (which can be either `decodeToRenameDelay` or `fetchToDecodeDelay`, depending on the instruction and which stage was blocked) inserted to its weight. This modification is effective only for modeling of instruction execution on gem5. Our new definition is as follows (the additional term in **bold**).

$$\begin{aligned} \textit{Weight} = \textit{ICache or ITLB miss latency} \\ + \boldsymbol{\textit{Number of additional bubbles}} \end{aligned} \quad (5)$$

In the case of the example of Fig. 4, the weight of $D_1D_2$ edge (the edge from Dispatch node of $i_1$ to that of $i_2$) is 2 and that of $D_4D_5$ edge (the edge from Dispatch of $i_4$ to that of $i_5$) is 3.

[*2] Our gem5 simulation adds 14 (7+7) cycles of bubbles.

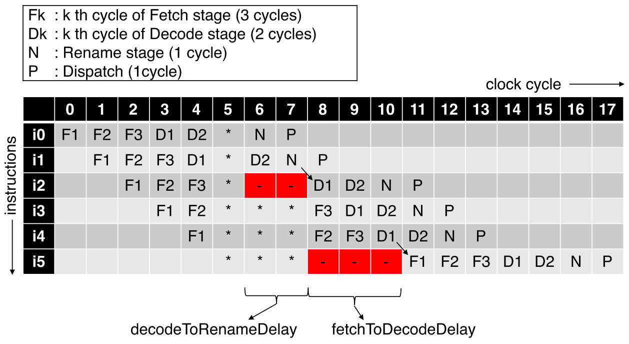

Fk : k th cycle of Fetch stage (3 cycles)
Dk : k th cycle of Decode stage (2 cycles)
N : Rename stage (1 cycle)
P : Dispatch (1cycle)
clock cycle
instructions
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
i0 F1 F2 F3 D1 D2 * N P
i1 F1 F2 F3 D1 * D2 N P
i2 F1 F2 F3 * - - D1 D2 N P
i3 F1 F2 * * * F3 D1 D2 N P
i4 F1 * * * F2 F3 D1 D2 N P
i5 * * * - - - F1 F2 F3 D1 D2 N P
decodeToRenameDelay
fetchToDecodeDelay

**Fig. 4** An example of the additional bubble in frontend of gem5.

**Table 3** Details of the target microarchitecture models.

| Parameters | Config-A | Config-B |
|---|---|---|
| CPU model | O3CPU | O3CPU |
| Frequency | 2 GHz | 2 GHz |
| ROB/Issue queue | 192/60 | 32/32 |
| LQ/SQ entries | 72/48 | 22/16 |
| Pipeline width | 8 (issue width: 6) | 2 |
| Branch predictor | Tournament predictor | |
| Choice/global/local predictor size | 8 K/8 K/2 K entries | |
| Local history table size | 2 K entries | |
| L1 ICache/Dcache size | 32 KB/32 KB | |
| L1 Dcache access latency | 4 cycles | |
| L2 cache size | 256 KB | |
| L2 cache access latency | 12 cycles | |
| DRAM | DDR3-1600 11-11-11 | |

## 5. Evaluation

### 5.1 Experimental Setup

We use the gem5 cycle-accurate simulator with x86 ISA and SE (syscall emulation) mode for our evaluations. We evaluated dependence models with two processor models with different parameters to confirm that the proposed model works with multiple configurations. Validation efforts have been conducted for gem5. A. Butko et al. [6] and A. Gutierrez et al. [13] proved the simulator matched real processors for ARM ISA. A. Akram and L. Sawalha [3] examined x86 simulators including gem5, Multi2sim, PTLsim and Sniper. They conclude gem5 is relatively accurate among the simulators that have the flexible design and implementation of microarchitecture.

The detailed microarchitectural parameters of the processors we modeled in our study are on par with Intel's microprocessor. **Table 3** summarizes the detailed parameters. Config-A and B model processors similar to Haswell and Silvermont microprocessor, respectively. Haswell, which is used for desktop computers and servers, is a relatively rich OoO processor. On the other hand, Silvermont, which is used for embedded systems is leaner than Haswell.

We use 25 applications from the SPEC CPU2006 benchmark suite with `ref` inputs to evaluate the dependence graph models. Benchmarks are fast-forwarded 1 billion instructions, another 100 million instructions were used to warmup the processor state, and the next 1 million instructions were simulated in detail.

### 5.2 Accuracy Evaluation Results with Config-A

**Figure 5** shows the results of the accuracy evaluation of the proposed enhanced dependence graph model (without gem5-

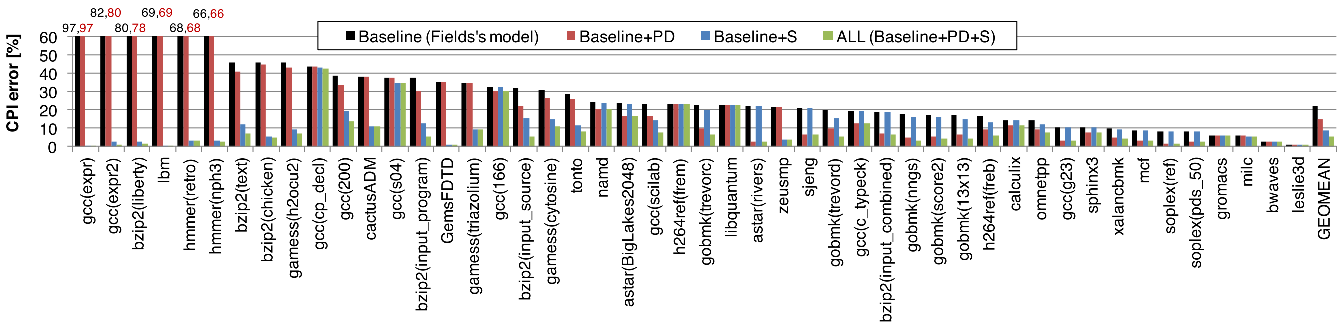


**Fig. 5** CPI estimation error of the enhanced dependence graph model with Config-A (rich processor).

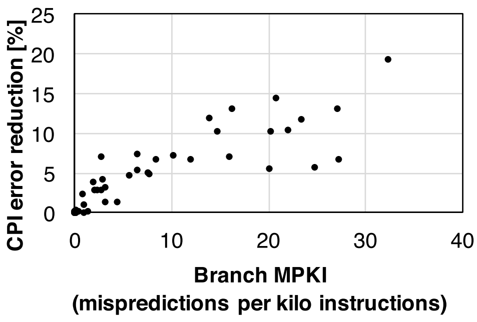


**Fig. 6** Correlation between the branch misprediction rate and the CPI error reduction of *Baseline+PD* with Config-A.

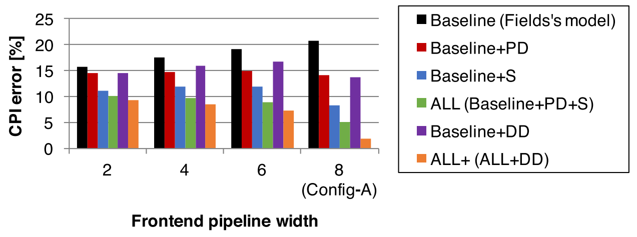


**Fig. 7** Sensitivity analysis with different frontend pipline widths.

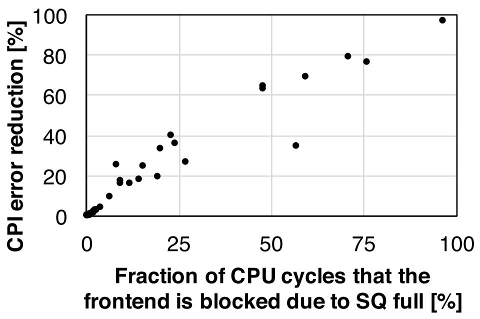


**Fig. 8** Correlation between the fraction of cycles that the frontend is blocked due to SQ full and the CPI error reduction of *Baseline+S* with Config-A.

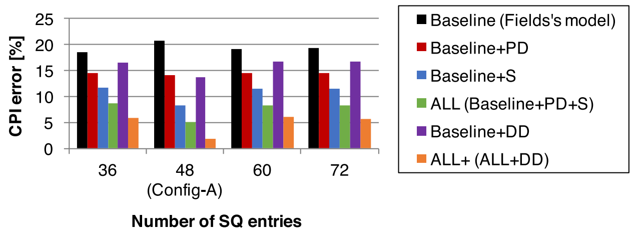


**Fig. 9** Sensitivity analysis with different numbers of SQ entries.

specific modifications). The baseline is the Fields's model, whose results (Fig. 2) are shown in the chart for reference. *Baseline+PD* is the model that takes the dynamic variation of branch misprediction penalties described in Section 3.2 into account. *Baseline+S* is the model with the *S* node and edges *CS*, *SS* and *SD* which models the modern SQ organization described in Section 3.3. *ALL* is the model with both enhancements. On average, the CPI errors of the *Baseline*, *Baseline+PD*, *Baseline+S* and *ALL* are 22.0%, 14.9%, 8.9% and 5.3%, respectively. The *Baseline+PD*, *Baseline+S* and *ALL* reduce the CPI errors by 32.4%, 59.6% and 76.0%, respectively, compared to the *Baseline*.

It is interesting to see that some programs such as `gcc` and `bzip2` with different inputs result in different amounts of CPI estimation accuracy. This is not surprising since different inputs sometimes exercise different control/data paths and end up in totally different behaviors. For example, `gcc(expr)` executes 1.5 times more store instructions compared to `gcc(g23)`, causing more SQ full events which directly affects the CPI error. The effect of each modification is discussed in the following paragraphs.

**PD:** The error reduction of *Baseline+PD* comes from accurately modeling the branch mispredictions. To examine whether our enhancements correctly capture microarchitectural events, we calculate the correlations between CPI reductions by our models and the numbers of events corresponding to the extensions. **Figure 6** shows the correlations between the branch mispredictions per kilo instructions and the CPI errors, where we can see a high correlation coefficient of 0.87. As expected, it is important to consider the dynamic variation of branch misprediction penalties especially for the benchmarks with higher misprediction ratio.

**Figure 7** shows the CPI estimation error of Baseline+PD with different frontend widths. Baseline+PD reduces more CPI estimation error with the wider frontend. It is because the number of instructions to be squashed increase with the broader frontend. In the branch misprediction recovery mechanism adopted in gem5, the misprediction penalty gets longer with more instructions on the incorrect path.

**S:** The *Baseline+S* model reduces the CPI errors significantly for about half of the benchmarks. For example, it reduces the errors by 99.7% for `gcc(expr)` and `lbm`. **Figure 8** shows the relationship between the fraction of cycles the frontend was blocked due to SQ full and the CPI errors. The correlation coefficient is 0.97. Our model properly models the SQ full event and its associated stalls and improves upon the Fields's model.

**Figure 9** shows the CPI estimation error of Baseline+S with different numbers of SQ entries. Since SSQ combines SQ and WBB, it tends to be composed of a large number of entries. We

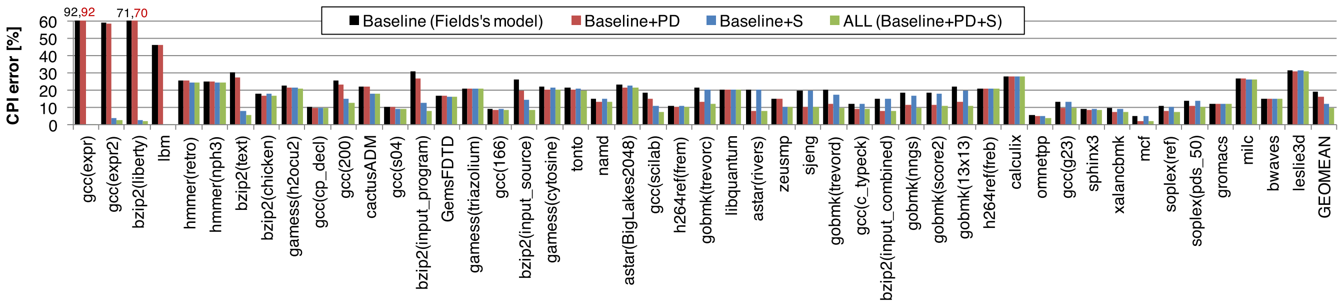


**Fig. 11** CPI estimation error of the enhanced dependence graph model with Config-B (lean processor).

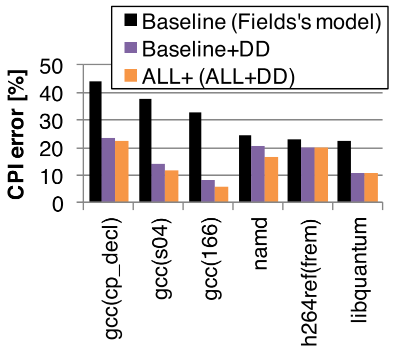


**Fig. 10** CPI estimation error with the gem5-specific enhancement with Config-A.

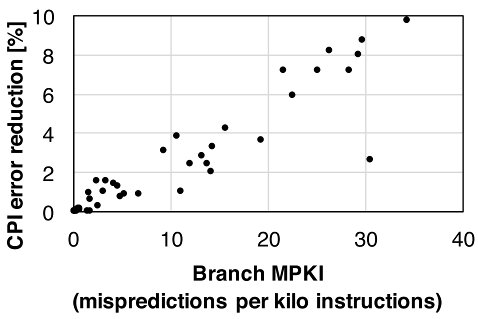


**Fig. 12** Correlation between the branch misprediction rate and the CPI error reduction of *Baseline+PD* with Config-B.

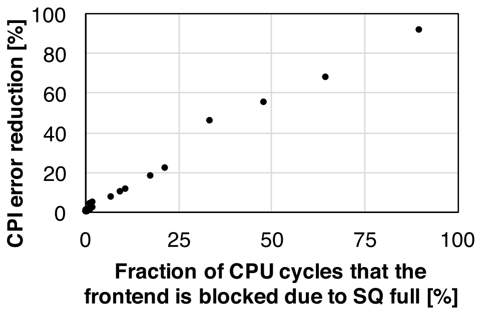


**Fig. 13** Correlation between the fraction of cycles that the frontend is blocked due to SQ full and the CPI error reduction of *Baseline+S* with Config-B.

see less correlation between the CPI errors and the number of SQ entries. Although the SQ capacity affects how long it takes to fill the SQ, the size of 72 entries is not enough to satisfy the required store instruction rate, resulting in pipeline stalls regardless of the SQ size.

**ALL:** With both enhancements considered, *ALL* almost removes the CPI errors in most of the applications. However, there are still some applications that exhibit CPI errors higher than 20%: `gcc(cp_decl)`, `gcc(s04)`, `gcc(166)`, `namd`, `h264ref(frem)`, and `libquantum`.

**DD: Figure 10** shows the CPI errors when gem5-specific modification is applied (only the applications with greater than 20% error with *ALL* are shown). *Baseline+DD* and *ALL+* add the gem5-specific modification to the *Baseline* and *ALL*, respectively. The gap between Baseline and Baseline+DD comes from the conservative implementation of the frontend of gem5, which implies that gem5 reports larger CPI when the frontend is blocked frequently.

On average across all the benchmarks, the average of CPI errors of *ALL+* is reduced to 2.1%, which is a 90.3% reduction from the *Baseline*. We believe that our enhanced dependence graph model is a useful tool which enables detailed microarchitectural bottleneck identification for modern OoO processors. There are still some benchmarks whose CPI error exceeds 10%, where further investigation and model enhancements are required, which are left for future work.

## 5.3 Accuracy Evaluation Results with Config-B

**Figure 11** shows the results of the accuracy evaluation without gem5-specific modifications. On average, the CPI errors of *Baseline*, *Baseline+PD*, *Baseline+S* and *ALL* are 19.1%, 16.0%, 12.2% and 9.7%, respectively. *Baseline+PD*, *Baseline+S* and *ALL* reduce the CPI errors by 16.4%, 36.3% and 49.1%, respectively, compared to the *Baseline*. We discuss the effect of each modification in the following paragraphs.

**PD:** The error reduction on average is smaller than Config-A. One of the possible reasons is branch misprediction penalties vary less than Config-A because of a small number of ROB entries of Config-B. **Figure 12** shows the correlations between the branch mispredictions per kilo instructions and the CPI errors. The correlation coefficient is 0.95, which is higher than 0.87 in the case with Config-A.

**S:** *Baseline+S* model reduces the CPI errors significantly in `gcc(expr)`, `gcc(expr2)`, `bzip2(liberty)`, `lbm` and `bzip2(text)` as well as Config-A. However, the error reduction on average is much smaller than Config-A. It is because applications on Config-B are less store-bandwidth centric than on Config-A. **Figure 13** shows the relationship between the fraction of cycles the frontend was blocked due to SQ full and the CPI errors. The correlation coefficient is higher than 0.99.

**ALL:** Compared to Config-A, the CPI errors of Config-B are higher. One of the reasons for the high errors is that the error

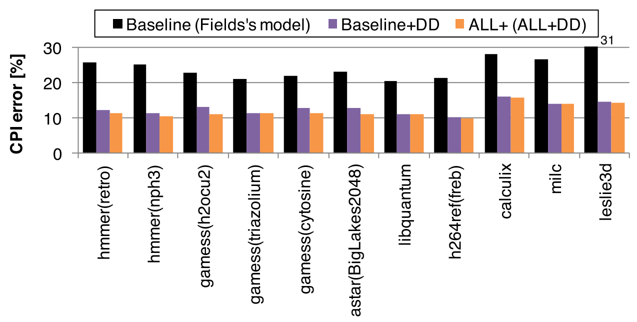


**Fig. 14** CPI estimation error with the gem5-specific enhancement with Config-B.

reduction of *Baseline+S* is smaller than Config-A. The applications with CPI errors higher than 20% are: `hmmer(retro)`, `hmmer(nph3)`, `gamess(h2ocu2)`, `gamess(triazolium)`, `gamess(cytosine)`, `astar(BigLakes2048)`, `libquantum`, `h264ref(freb)`, `calculix`, `milc`, `leslie3d`.

**DD: Figure 14** shows the CPI errors when gem5-specific modification is applied (only the applications with greater than 20% error with *ALL* are shown). The number of applications where *Baseline+DD* reduces CPI errors significantly is larger than Config-A. This means some applications experience more frontend blocks on Config-B because of its restricted parameters.

On average across all the benchmarks, the CPI errors of *ALL+* is reduced to 4.4%, which is a 77.1% reduction from the *Baseline*. These evaluations show that the proposed model can apply to both rich and lean OoO processors.

## 6. Related Work

### 6.1 Execution Time Formulation

Formulating the execution time [1] or the CPI [15] with architectural events' counts as variables is also a well-known method to understand program's performance bottleneck. This approach naively represents the execution time as the sum of the product of the number of occurrences of the architectural events (such as cache misses and branch mispredictions) and the penalty per one event. However, to break down the execution time into architectural events accurately, overlapping of multiple instructions and the dynamic variation of the penalty have to be taken into account [8]. It is because multiple instructions are reordered and executed in parallel.

One of the strong points of this approach is that it is easy to count the number of events with small overheads. The performance monitoring unit, which most of the general purpose processors possess, can be used for this purpose. Though it is useful for the analysis coarser than 1,000 cycles [27], it is not applicable for per instruction analysis which is the major target of the critical path analysis method.

### 6.2 Dependence Graph Models for OoO Processors

The dependence graph models (3 node types and 7 edge types [10], and 5 node types and 12 edge types [9], [11]) were proposed by Fields et al. The latter (5 node types and 12 edge types) is our baseline model in this work. As we point out in Section 2, these models are shown to work well on relatively simple OoO processors but are not applicable to modern OoO processor architectures such as gem5 simulator.

RpStacks [17] enhanced the dependence graph model by a fair amount (13 node types and 30 edge types) in order to realize the accurate processor design space exploration for their specific microarchitectural design. The main objective of our study is to keep the model simple and general which we believe is not accomplished by theirs.

One limitation of the dependency graph representation is that it is impractical to model queues that are not FIFOs. Therefore, the dependencies of the issue width and the number of IQ (Instruction Queue) entries are not modeled explicitly in the dependence graph models [9], [10], [11] and our model, which are considered by *RE* edge whose weight is determined from the execution trace. To take account of changes in issue timing due to various parameters, RpStacks attempts to model the dynamism of IQ by ED edge. We do not adopt this edge because it may be a false dependence when the IQ is large.

### 6.3 Dependence Graph Construction

Critical path analysis is a powerful tool to understand the program's behaviors and detect critical instructions. However, to construct the dependence graph, the large amount of the execution trace of the program on the target architecture is necessary. Criticality predictor [10] can detect critical instructions during the execution. The random sampling method using hardware support [21] enables to construct dependence graphs with small overheads. Our dependence graph model can be utilized with these techniques.

## 7. Conclusion

The dependence graph model of the OoO instruction execution is a useful representation for the critical path analysis. Although there are several existing models proposed in prior studies, most, if not all, of them lack enough detail for modern OoO processors or are too specific/complicated and limit generality and applicability. We propose an enhanced dependence graph model which remains simple but greatly improves the accuracy over prior models. The evaluations using the gem5 simulator with configurations similar to Intel's Haswell and Silvermont processor show that the enhanced model achieves CPI errors of 2.1% and 4.4% which are 90.3% and 77.1% improvements over the state-of-the-art Fields's model, respectively.

**Acknowledgments** This work was supported in part by CREST, JST. We would like to express our thanks to RIIT of Kyushu University for providing us the resource to conduct the experiments in this paper.

### References

[1] Ailamaki, A., DeWitt, D.J., Hill, M.D. and Wood, D.A.: DBMSs on a modern processor: Where does time go?, *Proc. 25th International Conference on Very Large Data Bases*, VLDB '99, pp.266–277, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1999).

[2] Akkary, H., Rajwar, R. and Srinivasan, S.T.: Checkpoint processing and recovery: Towards scalable large instruction window processors, *Proc. 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-36, pp.423–434, Washington, DC, USA, IEEE Computer Society (online), DOI: 10.1109/MICRO.2003.1253246 (2003).

[3] Akram, A. and Sawalha, L.: x86 computer architecture simula-

tors: A comparative study, *2016 IEEE 34th International Conference on Computer Design* (*ICCD*), pp.638–645 (online), DOI: 10.1109/ICCD.2016.7753351 (2016).

[4] Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D. and Wood, D.A.: The Gem5 simulator, *SIGARCH Computer Architecture News*, Vol.39, No.2, pp.1–7 (online), DOI: 10.1145/2024716.2024718 (2011).

[5] Burger, D. and Austin, T.M.: The SimpleScalar tool set, version 2.0, *SIGARCH Computer Architecture News*, Vol.25, No.3, pp.13–25 (online), DOI: 10.1145/268806.268810 (1997).

[6] Butko, A., Garibotti, R., Ost, L. and Sassatelli, G.: Accuracy evaluation of GEM5 simulator system, *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip* (*ReCoSoC*), pp.1–7 (online), DOI: 10.1109/ReCoSoC.2012.6322869 (2012).

[7] Carlson, T.E., Heirman, W. and Eeckhout, L.: Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation, *Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp.52:1–52:12, New York, NY, USA, ACM (online), DOI: 10.1145/2063384.2063454 (2011).

[8] Eyerman, S., Smith, J.E. and Eeckhout, L.: Characterizing the branch misprediction penalty, *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '06, pp.48–58, IEEE Computer Society (online), DOI: 10.1109/ISPASS.2006.1620789 (2006).

[9] Fields, B., Bodík, R. and Hill, M.D.: Slack: maximizing performance under technological constraints, *Proc. 29th Annual International Symposium on Computer Architecture*, ISCA '02, pp.47–58, Washington, DC, USA, IEEE Computer Society (online), DOI: 10.1109/ISCA.2002.1003561 (2002).

[10] Fields, B., Rubin, S. and Bodík, R.: Focusing processor policies via critical-path prediction, *Proc. 28th Annual International Symposium on Computer Architecture*, ISCA '01, pp.74–85, New York, NY, USA, ACM (online), DOI: 10.1145/379240.379253 (2001).

[11] Fields, B.A., Bodík, R., Hill, M.D. and Newburn, C.J.: Using interaction costs for microarchitectural bottleneck analysis, *Proc. 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-36, pp.228–239, Washington, DC, USA, IEEE Computer Society (2003).

[12] Fields, B.A., Bodik, R., Hill, M.D. and Newburn, C.J.: Interaction Cost and Shotgun Profiling, *ACM Trans. Archit. Code Optim.*, Vol.1, No.3, pp.272–304 (online), DOI: 10.1145/1022969.1022971 (2004).

[13] Gutierrez, A., Pusdesris, J., Dreslinski, R.G., Mudge, T., Sudanthi, C., Emmons, C.D., Hayenga, M. and Paver, N.: Sources of error in full-system simulation, *2014 IEEE International Symposium on Performance Analysis of Systems and Software* (*ISPASS*), pp.13–22 (online), DOI: 10.1109/ISPASS.2014.6844457 (2014).

[14] Henning, J.L.: SPEC CPU2006 benchmark descriptions, *SIGARCH Computer Architecture News*, Vol.34, No.4, pp.1–17 (online), DOI: 10.1145/1186736.1186737 (2006).

[15] Keeton, K., Patterson, D.A., He, Y.Q., Raphael, R.C. and Baker, W.E.: Performance characterization of a quad pentium pro SMP using OLTP workloads, *Proc. 25th Annual International Symposium on Computer Architecture*, ISCA '98, pp.15–26, Washington, DC, USA, IEEE Computer Society (online), DOI: 10.1145/279358.279364 (1998).

[16] Kessler, R.E., McLellan, E.J. and Webb, D.A.: The Alpha 21264 microprocessor architecture, *Proc. International Conference on Computer Design*, ICCD '98, pp.90–95 (online), DOI: 10.1109/ICCD.1998.727028 (1998).

[17] Lee, J., Jang, H. and Kim, J.: RpStacks: Fast and Accurate Processor Design Space Exploration Using Representative Stall-Event Stacks, *Proc. 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pp.255–267, Washington, DC, USA, IEEE Computer Society (online), DOI: 10.1109/MICRO.2014.26 (2014).

[18] Loh, G.H., Subramaniam, S. and Xie, Y.: Zesto: a cycle-level simulator for highly detailed microarchitecture exploration, *Proc. IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '09, pp.53–64, IEEE Computer Society (online), DOI: 10.1109/ISPASS.2009.4919638 (2009).

[19] Patel, A., Afram, F., Chen, S. and Ghose, K.: MARSSx86: A Full System Simulator for x86 CPUs, *Design Automation Conference 2011* (*DAC'11*) (2011).

[20] Perais, A.: gem5-users mailing list, available from ⟨https://www.mail-archive.com/gem5-users@gem5.org/msg12318.html⟩ (2015).

[21] Salverda, P., Tu ker, C. and Zilles, C.: Accurate critical path prediction via random trace construction, *Proc. 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pp.64–73, New York, NY, USA, ACM (online), DOI: 10.1145/1356058.1356068 (2008).

[22] Sanchez, D. and Kozyrakis, C.: ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems, *Proc. 40th Annual International Symposium on Computer Architecture*, ISCA '13, pp.475–486, New York, NY, USA, ACM (online), DOI: 10.1145/2485922.2485963 (2013).

[23] Semeraro, G., Magklis, G., Balasubramonian, R., Albonesi, D.H., Dwarkadas, S. and Scott, M.L.: Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling, *Proc. 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pp.29–40, IEEE Computer Society (online), DOI: 10.1109/HPCA.2002.995696 (2002).

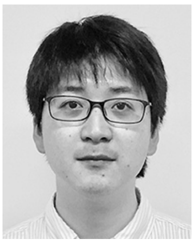
[24] Shen, J.P. and Lipasti, M.H.: *Modern processor design: Fundamentals of superscalar processors*, McGraw-Hill Higher Education, Boston (2005).

[25] Tanimoto, T., Ono, T., Inoue, K. and Sasaki, H.: Enhanced Dependence Graph Model for Critical Path Analysis on Modern Out-of-Order Processors, *IEEE Computer Architecture Letters* (in press) (online), DOI: 10.1109/LCA.2017.2684813 (2017).

[26] Ubal, R., Jang, B., Mistry, P., Schaa, D. and Kaeli, D.: Multi2Sim: A Simulation Framework for CPU-GPU Computing, *Proc. 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pp.335–344, New York, NY, USA, ACM (online), DOI: 10.1145/2370816.2370865 (2012).

[27] Yang, X., Blackburn, S.M. and McKinley, K.S.: Computer performance microscopy with shim, *Proc. 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pp.170–184, New York, NY, USA, ACM (online), DOI: 10.1145/2749469.2750401 (2015).

[28] Yourst, M.T.: PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator, *Proc. IEEE International Symposium on Performance Analysis of Systems Software*, pp.23–34 (online), DOI: 10.1109/ISPASS.2007.363733 (2007).



**Teruo Tanimoto** received his M.S. degree from The University of Tokyo, Japan, in 2012, and was a researcher for Fujitsu Laboratories Ltd., Japan and engaged in developing UNIX server products. He is currently a Ph.D. student in the Graduate School of Information Science and Electrical Engineering at Kyushu University. His research interests include computer system modeling and hardware-software codesign. He is a member of ACM and IPSJ.

**Takatsugu Ono** received a Ph.D. degree from Kyushu University, Japan, in 2009. He was a researcher for Fujitsu Laboratories Ltd., Kawasaki, Japan, and engaged in developing a server for a data center. He is currently an assistant professor in the Faculty of Information Science and Electrical Engineering at Kyushu University. His research interests include the area of memory architecture, secure architecture, and supercomputing. He is a member of IEEE, IPSJ, and IEICE.

**Koji Inoue** received his B.E. and M.E. degrees in computer science from Kyushu Institute of Technology, Japan in 1994 and 1996, respectively. He received his Ph.D. degree in Department of Computer Science and Communication Engineering, Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan in 2001. In 1999, he joined Halo LSI Design & Technology, Inc., NY, as a circuit designer. He is currently a professor of the Department of I&E Visionaries, Kyushu University. His research interests include power-aware computing, high-performance computing, secure computer systems, 3D microprocessor architectures, multi/many-core architectures, nanophotonic computing and quantum computing.