Proceeding of Asia South Pacific Design Automation Conference(ASPDAC2003), pp.129~134, January, 2003, Kitakyushu, Japan

Trace-driven Rapid Pipeline Architecture Evaluation Scheme for ASIP Design

Jun Kyoung Kim
Systems Modeling Simulation Lab
Dept of EE/CS, KAIST
373-1 Kusong-dong, Yusong-gu, Taejon
305-701, Korea
+82.42.869.5454
jkkim@smslab.kaist.ac.kr

Tag Gon Kim
Systems Modeling Simulation Lab
Dept of EE/CS, KAIST
373-1 Kusong-dong, Yusong-gu, Taejon
305-701, Korea
+82.42.869.5454
tkim@ee.kaist.ac.kr

ABSTRACT

This paper proposes a rapid evaluation scheme of pipeline architecture using phase-accurate simulation with only delay model and trace. With latency information for every stage, we can decide if an instruction in one stage can proceed to the next stage or if an instruction can be issued for each cycle without evaluating the value for registers. Branch target becomes available with trace generated by fast instruction set simulation. Fast verification time becomes possible because instruction set simulation is performed only once.

I. INTRODUCTION

The appearance of totally new applications accelerates the development of new embedded systems for them. In addition, the fact that the specification of such an application evolves over time makes programmability of embedded system more and more important. Therefore, the processor component should be developed in such a way that enables us to get both the performance satisfying the requirement and fast time-to-market. To design an optimal processor for an application is an intensive task because the architecture of a processor is very diverse and complex.

To address this problem, a few methods that improve evaluation performance are proposed such as hardware emulation or compiled simulation. The evaluation performance using hardware emulation is high, but development takes very much time and is inflexible. In addition, we can't make an observation on the internal state of the models being developed, which result in a long design time due to difficult debugging. With compiled simulation can we get a significant improvement in evaluation time by making use of a priori knowledge to accelerate simulation, with the highest efficiency achieved by employing static scheduling techniques [1]. We can also apply the same technique to our framework, the trace-based token-level pipeline simulation proposed in this paper. This is reserved as a further work and we expect significant improvement on simulation performance.

The scheme proposed in this paper is based on a language-based hierarchical design methodology where pipeline architecture design is performed after instruction set architecture is already developed. Our ADL(architecture description language), XR^2(eXtensible, Reusable and Reconfigurable), which is an extension of READ(Reusable Architecture Description)[2] language, supports such characteristics as retargetable compiler/simulator generator. To get a high-performance pipeline simulator, we introduced a trace-driven token-level simulation technique. Pipeline architecture can be viewed as a sequence of stages. With issue latency and result latency information defined for each stage, we can decide if an instruction can proceed from one stage to the next stage considering various factors such as data dependency and control dependency without evaluating the real value for registers or memory addresses. In addition, if resources such as functional unit, data bus, register ports are defined for combinations of stages and instructions, we can get the utilization for every resource. This can be done in cycle-by-cycle manner, thus we can apply the cycle-based simulation also. The branch target would be unavailable without real values which can only be determined at run-time. With trace which can be acquired at instruction set architecture level, we can solve this problem by fixing the target address for all the branching instruction. In addition, the separating an instruction set architecture and pipeline architecture hierarchically makes the trace equivalent to different pipeline architectures. In other words, we can evaluate different pipeline architectures with only one trace. Trace-based performance evaluation has often been used for rapid cache simulation[3] or designing on-chip communication architecture[4].

Chapter 2 shows the background and the overall structure of our framework. In chapter 3, we will explain the B-PASS formalism, a basic formalism on which our pipeline description, $LowXR^2$, is based and syntax briefly. The simulation algorithm that enables us to evaluate the performance indexes in a short time will be given in chapter 4. We will show the effectiveness of our framework by exemplifying the CalmRISC processor with the proposed ADL in chapter 5. Finally, we will conclude this methodology and talk about future work in chapter 6.

II. Background and Overall Framework

A system designer is faced with the tasks of rapidly exploring and evaluating different architectures. Evaluating processor architecture requires a simulator and a compiler. Developing a simulator and a compiler whenever processor architecture changes delays the development time very much, resulting in a very long time-to-market. As a result, language-based design methodology appears in a design paradigm and architecture description language (ADL) is developed to drive automatic compiler/simulator toolkit generation. A top-down design framework is useful because information acquired at the high-level design can be exploited at the low-level design. Measuring some performance indexes as high-level as possible is very fast because unnecessary processing is not performed. Top-down design framework is well suited for processor design because there are many and complicated aspects in processor component.

Table 1 Required information for simulation objective

Objectives/Feature	*Required information*	
Instruction correctness	Per instruction behavior	High
Addr mode correctness	Per addr mode behavior	Level
Code size	Instruction set architecture	
Total cycle count	Instruction – stage - delay relation	
Resource usage	Instruction – stage – resource relation	
Cycle-true snapshot	Cycle-accurate model	
Area estimation	Register transfer level info.	Low
Power estimation	Register transfer level info	Level

Table 1 shows the relationship between objectives of processor simulation and required information for them. Based on this, our design framework is devised according to the following grounds.

- A design stage should be related with explicit objectives of modeling/simulation
- A design stage should be as high as possible for the objectives of modeling/simulation : simulation speed issue
- A design stage should not be too far from both higher/lower design stages : seamless design environment
- Information exchange between design stages should be well defined

Table 2 Design Stages of XR^2 Framework

Required Information	Design Stage	Architecture
Instruction Set Architecture	HiXR^2	Instruction Set Architecture
Per Instruction Behavior		
Per Addressing Mode Behavior		
Instruction - stage-delay relation	Token-level LowXR^2	Pipeline, VLIW, Superscalar
Instruction-stage-resource relation		
Cycle-accurate Model	Cycle-accurate LowXR^2	
Register Transfer Level Model	MicroXR^2	Micro-Architecture

We propose a processor design framework by dividing the required information in Table 1 into 4 design stages as in Table 2 and Fig.1. Three of the design stages, HiXR^2, cycle-accurate LowXR^2 and MicroXR^2, are observed in many other frameworks. First, HiXR^2 is a design stage for instruction set architecture, which consists of instruction set and addressing modes. This design stage enables us to determine what instructions and addressing modes are useful for a specific application. The processor architecture is instruction set architecture and processor behavior is defined on instructions and addressing modes. Cycle-accurate LowXR^2 is a design stage for cycle-accurate model that renders us a cycle-true behavior including all the snapshot of storage elements. This model helps us determine a pipeline architecture including data path and control path. In this case,

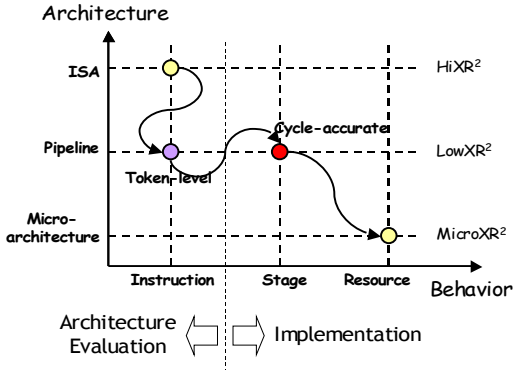


Fig.1 Taxonomy for Dividing the Design Stage

processor architecture is pipeline architecture and the processor behavior is defined on stages or resources. A disadvantage of this model is a simulation speed. The last one, MicroXR^2, is an implementation model. Generally, this is a HDL model.

Our main concern in this paper is fast evaluation of pipeline processor model by introducing token-level LowXR^2. The token-level LowXR^2 model is not cycle-accurate in a strict sense because it does not inform us of a cycle-accurate snapshot of storage elements. In other words, this model does not process such activities as evaluating register values or arithmetic operations defined on instructions cycle by cycle. Instead, this model manages an abstract pipeline model whose stages have delay parameters such as issue/result latency. This model gives us an accurate total cycle count, resource utilization, rough area and power estimation.

Fig.2 shows a design flow of HiXR^2 and token-level LowXR^2.

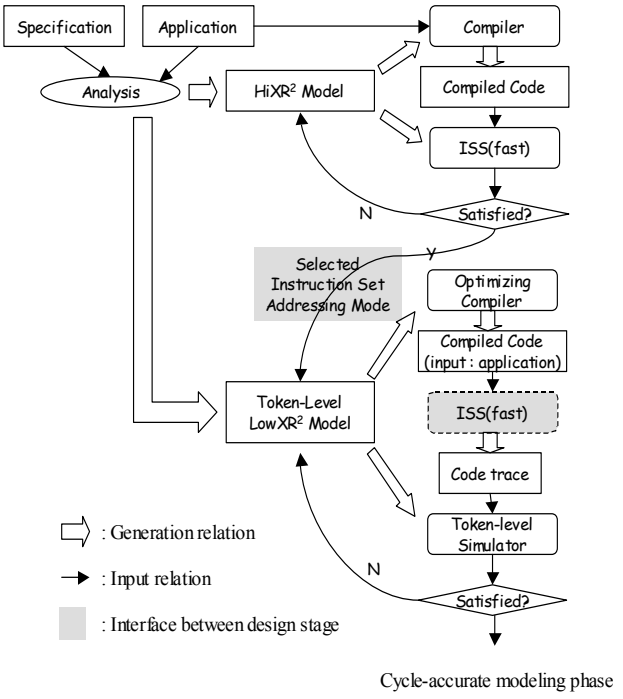


Fig.2 Design Framework

Token-level LowXR^2 simulator uses traces that are free of control dependency to make it possible that token-level simulator does not process any value evaluation. The traces are generated by fast instruction set simulator.

III. Semantics and Syntax

A. Semantics : B-PASS Formalism

With the instruction set and addressing mode selected as a best solution for the target application at HiXR^2, we can design the target processor's pipeline architecture. Efficient design of a processor at this abstraction level requires formal definition of a processor, thus we introduced *B-PASS*(Basic Pipeline Architecture System Specification) formalism that defines a processor's pipeline architecture in a mathematically correct way. B-PASS formalism is defined based on the HiISA, semantics for HiXR^2, thus both formalisms are shown below.

HiISA = <IS, AM, ST, R_{IA}, R_{AS}>, where

- *IS* : A structured set for the instructions of target architecture.
- *AM* : A set of addressing modes of target architecture.
- *ST* : A set of the storages of target architecture.
- $R_{IA} \subseteq IS \times AM$: A relation between the instruction set and addressing modes..
- $R_{AS} \subseteq AM \times \{ST \cup IMM\}$: A relation between the addressing modes and associated storages (or immediate value).

B-PASS = < HiISA, STAGES, RES, pipe, resource, lat> *where*

- *STAGES* : set of pipeline stages
- *RES* : set of resources
- $pipe : IS \times AM \rightarrow STAGES^n$ *for positive integer n*
  - The pipeline architecture is a function of instruction set and addressing modes.
- $resource : IS \times AM \times STAGES \rightarrow RES$
- $lat : STAGES \times IS \times AM \rightarrow I \times I$
  - Stage has two latency information, issue/result latency

B-PASS formalism defines pipeline architecture with HiISA in a hierarchical way. *STAGES* is a set of stages which constitute a pipeline architecture. *RES* is a set of resources. *pipe* is a function that maps instruction set and addressing mode to the pipeline architecture which is a sequence of stages. We can model what resources are used at what stage, depending on an instruction and addressing mode with *resource* function. Latency information is specified with *lat* function. The first integer *n* is an issue latency, which implies that the stage can receive a new instruction every *n*th clock cycle. The second integer *n* is a result latency, which implies that *n* clock cycles are necessary for an instruction to finish its job at the stage.

B. Syntax

Syntax of token-level LowXR^2 is a textual form reflecting the components of the B-PASS formalism. There are five sections as follows.

(i) General description section

General information about the processor is specified. We can specify the name of the processor, addressing modes, input port and output ports in this section.

(ii) instruction alias section

For modeling convenience, we can group the instructions to an alias. This is helpful when many instructions share pipeline architecture. We need not specify for all the instructions, but for this instruction aliases. This is used at the stage description body. The syntax has the following form.

$Alias\_name : inst_1, inst_2, inst_3, \ldots, inst_n;$

(iii) Resource declaration section

The resources used at the data path of target processor are declared here. This section specifies the name and the number of resources.

(iv) pipeline section

This section has the pipeline architecture by sequencing stages. This section has the following form

$Pipeline1\{ (stage_1, stage_2, stage_3, \ldots, stage_n); \}$

(v) stage description section

This section describes various feature of stages used at the pipeline section. The following is an example of a stage description.

```
m_cycle_ex{
    issue latency = 4; result latency = 8;
    phase(7:1){ bypass_to s_cycle_ex, m_cycle_ex; }
}
mem{
    issue latency = 1; result latency = 1;
    switch(inst){
        case(LD_INST): phase(0){ lock DABUS, DBUS;}
    }
}
```

m_cycle_ex stage has four as its issue latency and eight as its result latency. The body of syntax $phase(n_1:n_2)$ should be performed at the $(n_1+1)^{th}$ cycle and $(n_2+1)^{th}$ phase. For example, *phase(7:1)* shown in the example specifies that the bypassing the result to the *s_cycle_ex* and *m_cycle_ex* stages should be performed at the 2nd phase of the 8th(last) cycle. The phase(n) is used only for the stages with result latency one.

In addition, some syntactic sugars specialized to processor architecture are supported. For example, with keywords *fetch_opds, lock_dest, unlock_dest* and *bypass_to* can we model the data hazard of a processor. These keywords are used at the stage description body. When keyword *lock_dest* is specified at a stage, it implies that the destination register should be locked at the stage to declare this register is unavailable. Keyword *unlock_dest* performs the unlocking of the destination register. Keyword *fetch_opds* implies that operand fetching is done at the stage. If the registers which contain the source operands are locked by other instruction, it should wait till the registers become available. Keyword *bypass_to* specifies the forwarding logic. This keyword has an argument which is a stage. This implies that the result is directly forwarded to the stage specified as an argument regardless of the register locking. With these keywords and simple scoreboarding algorithm can we model and simulate the data hazard very easily

The behavior of an instruction is not specified because the simulation at this abstraction level does not perform the evaluation of register value at all. In other words, only information about control-path is specified. This is helpful in simplifying not only the modeling but also retargeting a processor.

IV. Simulation Algorithm

Simulation algorithm has three inputs, APG, code trace and decoding table.

First input is **APG**, an abstract pipeline graph, that has a close relationship with B-PASS formalism. With APG, we can manage the pipeline architecture as a graph form. Node represents a stage and edge implies the stage sequence defined for an instruction. The mapping from B-PASS formalism to APG is as follows.

APG = <v, e> *is a directed acyclic graph and related with correct* ***B-PASS formalism*** *model as follows*

v = *STAGES*

$e = v \times v$ *where* $v^n \times e \times v^m$ *is an element of valid pipeline architecture set.*

This can be obtained with pipe(is, am) for $is \in IS^{\Sigma}$, $am \in AM$, *any value n and m that are nonnegative values.*

The second input is a code trace, **CT**, which is free of control dependency. This can be shared by different pipeline simulation which shares the instruction set architecture.

The third input is a decoding table, **DT**, which determines the pipeline path with instruction. This is synthesized from the LowXR^2 description.

Actually, we perform the phase-accurate simulation in our framework, but the explanation is shown on the cycle-by-cycle base for brevity.

A. Initialization

Initial configuration is shown in Fig3.(a). Initially, every stage has its latency information. For example, MULTI_ALU stage has one as its issue latency and two as its result latency. Next, there are dynamic values for each latency, issue latency count and result latency count. Issue latency count acts as a condition with which we can decide if a new instruction can be issued to that stage. With result latency count, we can know if there is an instruction finished in that stage. During initialization, issue latency count and result latency count for all the stages are initialized to zero.

Topological sort is performed on original APG because we will decide whether an instruction in a stage can proceed to the next stage or not in reverse topological sorted order. An instruction in one stage can proceed to the next stage if possible because the processing of the next stage is already performed.

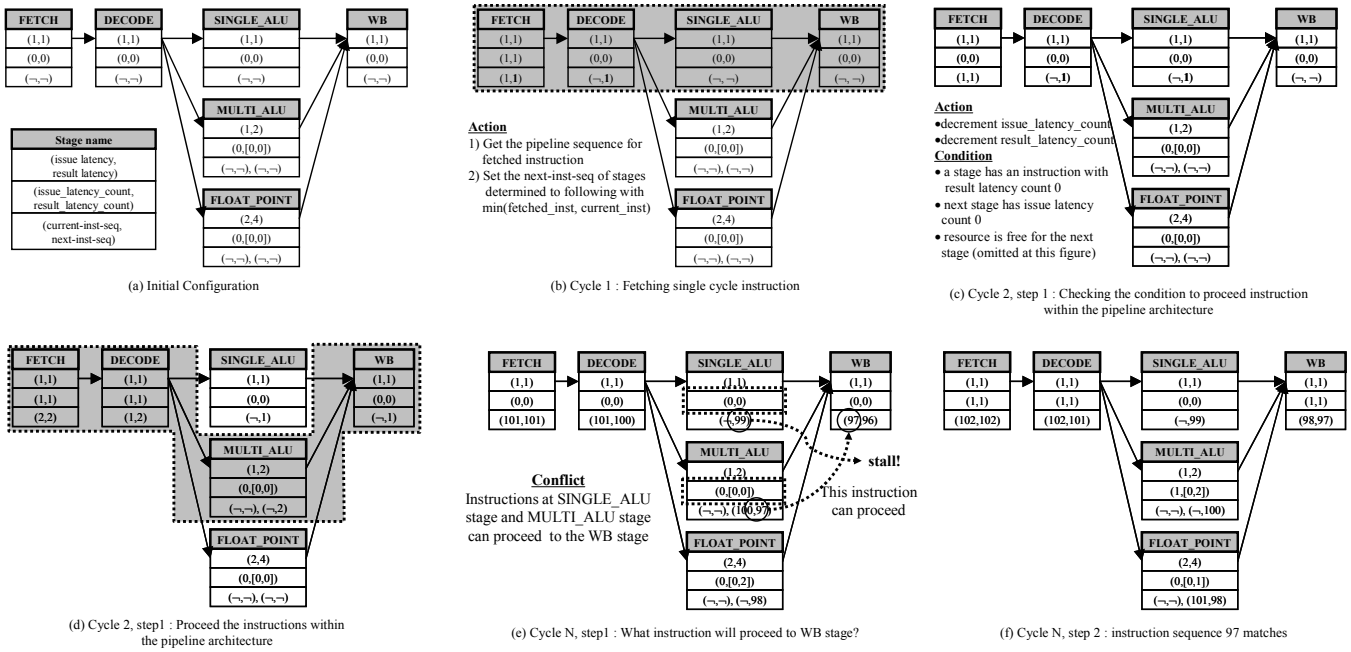
Last, there is instruction sequence information. This is used to keep the in-order completion causality of instruction processing.

Resources are assigned to each stage and resource table is managed although now shown in figure. Scoreboarding algorithm is used to manage the register file.

B. Issuing an instruction to a stage

There are two cases in issuing a new instruction to a stage. The first case is fetching instruction from instruction memory and the next case is handover the completed instruction between stages. In case of instruction fetching(Fig3.(b)), a new instruction can always be issued to the first stage as decoded by decoding table if the issue latency count of the stage is zero. Instruction handover between stages is more complex. The following conditions should be met for an instruction to proceed from a stage to the next stage.

- There is a completed instruction in one stage
- The next stage determined by APG and DT has zero as its issue latency count.
- Resources declared to be used at the next stage by the completed instruction are free.
- The register which contains the source operand should be



(a) Initial Configuration

(b) Cycle 1 : Fetching single cycle instruction

(c) Cycle 2, step 1 : Checking the condition to proceed instruction within the pipeline architecture

(d) Cycle 2, step1 : Proceed the instructions within the pipeline architecture

(e) Cycle N, step1 : What instruction will proceed to WB stage?

(f) Cycle N, step 2 : instruction sequence 97 matches

Fig.3 How the Simulation Algorithm Works on the Pipeline Model

available if operand fetching is done at the next stage.

- The sequence number expected to come by the next stage is the sequence number of the completed instruction.

The first condition can be checked if there is an instruction whose result latency is zero at the current stage(Fig.3(c)). Second, the issue latency of the next stage should be zero(Fig.3(c)). Third, the resources the instruction uses at the next stages should be free(Fig.3(c)). With the fourth condition, we can model the true data hazard. Finally, the instruction that the next stage expects to come should be the completed instruction of the current stage.

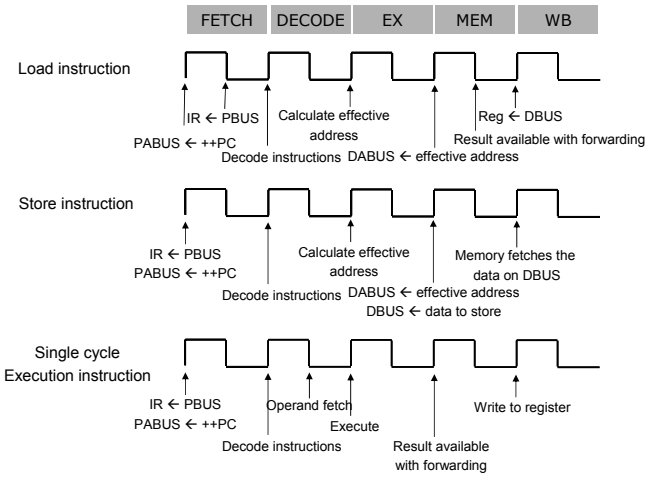
The third and fourth condition explains how it deals with resource conflict and data hazard. The true data hazard is addressed by scoreboard algorithm. Four keywords, *fetch_opds*, *lock_dest*, *unlock_dest* and *bypass_to*, enable us to model this true data dependency. When an instruction goes to a stage where operand fetch should be performed, it first checks if the source operand is available by looking into the scoreboard bit. If available, it can proceed to the next stage when its result latency is zero. If not, it should wait for another instruction that locked the operand to unlock it. Dependency checking of implicitly used registers such as status register is also handled. When an instruction enters a stage where it is specified to lock the destination, it should lock the destination. Forward modeling is possible by locating *unlock_dest* or *bypass_to* at an appropriate stage. These features are modeled in the way that the designer specifies at the token-level LowXR^2 description. We can model the control hazard by considering PC(Program Counter) as a resource. That is, when conditional branch instruction enters the pipeline, it locks the PC at the very first stage. When the conditional branch instruction has reached the stage at which the condition is expected to be resolved, it unlocks the PC. Because every instruction to be fetched uses PC at the very first stage, no other instruction can be issued. For the case of delayed branching, we depend on the compiler. Retargetable compiler generates a retargeted compiler based on the XR^2 machine description language and the syntax of LowXR^2 holds the expression power of delayed branching. Branch prediction scheme is reserved as a further work.

The fifth condition guarantees the in-order issue and in-order completion strategy. Instructions in the code trace have sequence number in an increasing order from sequence number one. When there is a conflict in a stage between instructions of precedent stages, instruction whose sequence number is identical to the expected sequence number of the next stage can precede. When an instruction enters a stage, it registers its own sequence number to the next stage. The next stage sets its expected sequence with a minimum value of its own expected sequence and the newly entered instruction's sequence number.(Fig.3(e)). The sequence number of an instruction that should finish next time is managed separately, and the simulation algorithm is borrowed from the ROB(Re-Order Buffer) used at the superscalar architecture.

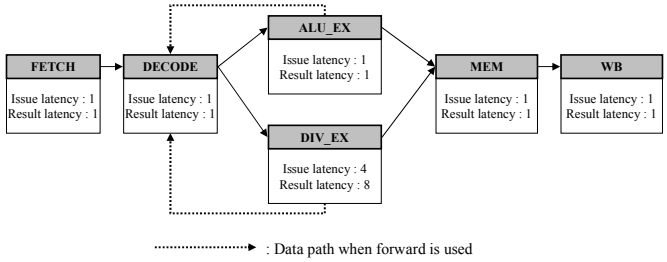
V. Experiment

We have experimented with CalmRISC[5], which is a 32-bit low-power MCU from Samsung Electronics Company. CalmRISC has pipeline architecture that consists of five stages: Fetch-Decode-Excute-Memory-Writeback. We used the ADPCM benchmark program as an application which is a member of MediaBench.

First, we have constructed an instruction set simulation environment for CalmRISC with HiXR^2. By simulating the



(a) Phase-accurate activity of CalmRISC



(b) Delay model of CalmRISC for Simulation

Fig. 4 Experimental Pipeline Model

ADPCM benchmark program with this simulator, we could get a trace whose target addresses of all the branch instructions are fixed. The performance of this instruction set simulator is about 30MIPS because we applied compiled simulation technique. The size of trace file is 874 Kbytes.

Next, with these fixed instructions and addressing modes, we experimented with various pipeline architectures on the proposed token-level simulation environment. The pipeline architecture of CalmRISC is relatively simple, so we made a variation on the original pipeline architecture. Fig.4 shows the pipeline architecture of modified CalmRISC processor. There are two pipeline paths : ALU pipeline and DIV pipeline. It takes longer time for *division* instructions to finish its operation. To improve one cycle period, we assumed that the result latency of DIV_EX stage is 8 cycles and it is internally pipelined into two micro-stages, thus issue latency is 4 cycles.

We tested our framework at the coursework which many graduate students hear. It took about two or three days for a graduate student to model the CalmRISC processor from scratch. The model size is 1716 bytes and 123 lines. We measured simulation performance on AMD Athlon 2100+, whose OS is Windows XP.

Table 3 Experiment Result : ADPCM on CalmRISC model

		Using Forward	Without Forward
Total cycle count		222174	435861
# of instruction executed		219170	
# stall due to data hazard		0	208686
Simulation time(ms)		1105.86	1459.28
CPI		1.01371	1.98869
Simulation performance	Kcycle/sec	200.907	298.682
	Kinst/sec	198.19	150.191

Simulation result is shown in table 3. All the modeling and simulation is performed in a phase-accurate way. When forward is applied, there is no data hazard. Without forwarding, data hazard happens 208686 times. By applying the forward, 49% improvement of total cycle count has been achieved. The number of instruction executed is same for the two cases. The simulation performance is about 200~300 Kcycle/sec and 150~200 Kinst/sec. Because stall happens many times for the second case, simulation performance of the second case in Kcycle/sec is better than that of the first case. When the simulator detects there to be a hazard, it does not process any other instruction in the pipeline model. This is possible because the evaluation is performed in reverse topological sorted order. On the other hands, the performance of the second case in Kinst/sec is worse than that of the first case. This is because the number of instruction executed is same, but the processing time of the second case is longer due to the stalls. When we experimented with only cycle-accurate model, not phase-accurate model, the simulation performance was observed to be as twice that of phase-accurate case. This is reasonable because the load of simulating a phase-accurate model is almost twice that of simulating a cycle-accurate model. In addition to the total cycle count, the resource utilization has been counted also.

Compared with the simulation performance of instruction set simulator, which is about 30MIPS, the simulation performance of token-level LowXR^2 is low but expected to be higher than that of other interpretive pipeline simulator. This is because unnecessary processing to get total cycle count is not performed. To get the correct total cycle count with conventional approach, we should perform cycle-accurate simulation. This simulation requires a cycle-accurate model, and to build a cycle-accurate model needs much more time and effort. In addition, cycle-accurate simulation takes much longer time because it evaluates all the values cycle-by-cycle, unlike the token-level pipeline simulation.

VI. Conclusion and Future Work

This paper proposes an efficient way of evaluating pipeline architecture by introducing a new abstraction level called token-level LowXR^2. The experiment shows that the new abstraction level, token-level LowXR^2, is valuable in exploring the large design space of pipeline architecture. Two factors enable us to get high simulation performance. First, evaluating the pipeline architecture without value evaluation saves the processing time very much. Without the evaluation, total cycle count and resource utilization can be measured with pipeline architecture with only latency information and resource assignment. Second factor is the hierarchical property of the proposed framework. With one trace that is generated with fast instruction set simulator and free of control dependency, multiple pipeline architectures can be evaluated. This is because pipeline architectures share the instruction set and addressing modes determined to use at the high-level design. This helps explore a large design space of pipeline architectures which share an instruction set architecture.

The ILP processor can also be modeled by token-level LowXR^2 with resource declaration, but some syntactic sugar and related simulation algorithm will be very helpful in designing such an ILP processor. To get correct total cycle count, a few more peripherals should be available such as cache. Currently, cache simulator that uses traces is developed based on the DINERO cache simulator. The trace used by the DINERO can be generated using our HiXR^2 simulator. We are currently working on this integration. Power and area is another important metrics for determining the architecture of a processor, thus high level estimation of power and area is necessary. We will show the effectiveness of our framework by exemplifying more processors and using various benchmark programs.

Other techniques can also be applied at the same time. The application of compiled simulation technique would greatly improve the simulation performance. By introducing the compiled simulation, we expect there to be improvement of simulation up to a few million cycles per second.

VII. REFERENCES

[1] Braun, G., Hoffmann, A., Nohl, A. and Meyr, H.., "Using static scheduling techniques for the retargeting of high speed, compiled simulators for embedded processors from an abstract machine description", Proceeding of the 14th ISSS, pp. 57-62, Montreal, Que. Canada Oct. 2001.

[2] Young Geol Kim and Tag Gon Kim, "A Design and Tool Reuse Methodology for Rapid Prototyping of Application Specific Instruction Set Processors", IEEE RSP-99, pp46-51, 1999, Clearwater, FL, U.S.A

[3] Zhao Wu and Wayne Wolf, "Iterative Cache Simulation of Embedded CPUs with Trace Stripping", Proceeding of the seventh Hardware/Software Codesign, pp. 95-99, 1999

[4] Lahiri, K., Raghunathan, A. and Dey, S., "System-Level Performance Analysis for Designing On-Chip Communication Architectures", IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, pp. 768-783, Vol. 20, No. 6, June 2001

[5] Kyoung-Moon Lim, et al., "CalmRISC™ : a low power microcontroller with efficient coprocessor interface", International Conference on Computer Design, pp. 299-302, 1999