Imprecise Data Computation
for High Performance Asynchronous Processors

Jeong-Gun Lee  Eui-seok Kim  Dong-Ik Lee

Dept. of Information and Communications, K-JIST
1 Oryong-Dong, Buk-Gu, Kwangju, 500-712, Republic of Korea
Tel: +82-62-970-2267
Fax: +82-62-970-2204
E-mail: euil@egurl.kjist.ac.kr, uskim@egurl.kjist.ac.kr, dilee@kjist.ac.kr

Abstract — Instruction level parallelism (ILP) is strictly limited by various dependencies. In particular, data dependency is the major performance bottleneck of data intensive applications. To accelerate the execution of sequential code serialized due to data dependencies, this paper proposes an imprecise computation as a fast data computing technique for a high-performance asynchronous processor. To show the performance benefits of the suggested computing model, simulation results are presented. The imprecise computation can be used effectively in data intensive processing with a microprocessor, a Digital Signal Processor or a multimedia processor.

I. Introduction

In its short lifetime of 26 years, the microprocessor has achieved a total performance growth of greater than 10,000 fold, in particular, through improvements in technology and microprocessor innovations. Moreover, the industry plans to achieve 100 BIPS by 2010 through the integration of one billion transistors on a chip. Though the rapid advances in semiconductor and computer architecture technologies support a high degree of parallelism, the concurrency of a code is seriously limited by various dependencies[1].

In this paper, we propose an imprecise computation (IC) as a new architectural computing model to accelerate data computations strictly serialized due to data dependencies with utilizing the high availability of hardware resources. This is an orthogonal approach to the previous approaches, which tried to resolve the data dependencies by exploiting more concurrency. The basic concept of an imprecise computation has been introduced by Lin et al.[2] in the area of real-time database and is a flexible technique for the design of decentralized real-time systems that are subject to transient overload: when a real-time system is overloaded, it does not process tasks completely but partially executes tasks in some tolerable degree. This concept is modified and applied to data processing at the instruction level for a high-performance computation.

In the imprecise computation, each variable corresponds to a pair of data as its own representation. Each functional unit performs its function with a given input data set and produces a pair of data as an intermediate result rather than producing an exact value completely. The computation is partially processed until the pair of the result data have been produced, if the result is not critically demanded in the form of an exact value. A pair of data representing a certain value is called an imprecise value. Producing imprecise values can be considered as partial execution of tasks in tolerable degree in real time database systems. However, in order to implement the imprecise computation, processor architectures need various functional units having various processing delays according to the input/output combinations of exact and imprecise values. To implement those functional units with locally optimized processing delay, asynchronous design methodologies are useful. Incorporating the imprecise computation to current high-performance asynchronous architectures leads to significant performance improvement.

This paper is organized as follows; Section 2 presents preliminaries necessary for better understanding of this paper. In Section 3, the imprecise computation method is explained in detail in various aspects. In Section 4, experimental results are given to show the effectiveness of the proposed method, and finally the conclusions are presented in Section 5.

II. Preliminaries

A. Data Dependency Relations

Data dependency relations may be classified as one of following three types: read after write (RAW), write after write (WAW) and write after read (WAR), according to the execution order of read and write instructions. Three types of data dependencies may cause performance degradation, because of pipeline stalls in a certain pipelined micro-architecture. In most contemporary high-performance computer architectures, WAR and WAW can be avoided by register renaming but RAW is still difficult.
to solve. Therefore, RAW is considered as a true data dependency. To resolve RAW dependency, several techniques have been proposed.

In particular, special emphasis is given to repetitive structures, such as for or while structures. In those repetitive structures, RAW dependency can create very long instruction sequences being composed of repetitive dependent instruction chains. This long instruction sequence seriously limits the performance of a system. Therefore, efficient data processing in loop structures is an important issue in improving the system performance.

### B. Carry Save Adder Structure

When the addition of three or more operands is performed using a two-operand adder, the carry-propagation, which is time-consuming, is repeated several times. If the number of operands is $k$, the carry-propagation occurs $(k-1)$ times. Several techniques for addition of multiple operands that attempt to lower the carry-propagation penalty have been proposed and implemented. The most well-known one is carry-save addition [3]. In the carry-save addition, the carry is propagated only in the last step, while in all other steps a partial sum and a sequence of carries are generated separately. A carry-save adder (CSA) for $n$-bit three operands, A, B, and C, with carry-in, $c_0$, is shown in Figure 1. Note that a CSA is called a (3,2) counter because it receives three operands and generates two partial results. Therefore, CSAs constitute a datapath to add multiple operands without carry-propagation. The structure of a counter can be extended to accept more inputs with only small increased delay.

### C. Asynchronous System Architecture

The asynchronous circuit avoids the use of a global clock which presents several serious limitations, such as performance and power consumption [4]. Asynchronous circuits, which are operating under the localized handshaking protocol, may improve system performance by exploiting the locally optimized timing for each functional unit.

### III. Imprecise Computation

In this section, an imprecise computation method is explained in detail. In current, an addition, a subtraction, a multiplication, and a shift are considered as functional units supporting the imprecise computation. Here, for the lack of space, we skip explaining a subtraction and a shift operations. Please refer to reference [5] for the details.

### A. Basics of an Imprecise Computation: Idea and Processor Architecture

As the name implies, for an imprecise computation, the operations are not processed completely. Instead of completing each operation, partial results are yielded in the form of a data-pair, for example in the case of an addition ($\text{sum, carry}$), and these pairs are used as inputs in the following operation. A complete representation of the results, in the form of data-pairs, can be obtained at each point of the computation flow. For example, consider an addition. Assume five operands are added sequentially using a carry save adder. The intermediate result ($\text{sum, carry}$) pair is not a complete representation. However, the exact value for each intermediate addition is always obtainable by adding the intermediate $\text{sum}$ and $\text{carry}$ pair. This computation is called imprecise computation. From this point, the concept of the imprecise computation is extended and applied to a processor architecture. In the remainder of this paper, we use the term imprecise value for a partially processed pair of data and use the term exact value for a completely processed single datum. Figure 2 shows the benefits of an imprecise computation in the aspect of performance. As shown, it is worthy to note that an imprecise computation is not a dependency resolution method, but the method of suppressing the computation time of an operation itself.

Figure 3 shows a superscalar architecture that supports the imprecise computation, and Table 1 shows the additional instruction set extended for the imprecise computation. As an example, the instruction 32C performs an addition of three exact values and generates a pair of data ($\text{sum, carry}$). It can perform an addition of one exact value and one imprecise value and produce an imprecise value as the result. Similarly, the instruction 42C
Figure 3: Computer Architecture for an Imprecise Computation

Table 1: Additional Instruction Set for an Imprecise Computation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instruction</th>
<th>Delay</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>32C</td>
<td>1 FAD</td>
<td>EV+IV–IV</td>
</tr>
<tr>
<td>Addition</td>
<td>42C</td>
<td>2 FADs</td>
<td>IV+IV–IV</td>
</tr>
<tr>
<td>Addition</td>
<td>52C</td>
<td>3 FADs</td>
<td>EV+IV–IV</td>
</tr>
<tr>
<td>Multiplication</td>
<td>2MUL</td>
<td></td>
<td>EV×EV–IV</td>
</tr>
<tr>
<td>Multiplication</td>
<td>3MUL</td>
<td></td>
<td>IV×IV–IV</td>
</tr>
<tr>
<td>Multiplication</td>
<td>4MUL</td>
<td></td>
<td>IV×IV–IV</td>
</tr>
</tbody>
</table>

- EV and IV mean Exact Value and Imprecise Value respectively
- FAD is 1-bit Full Adder Delay
- Delay marked by “-” will be presented in the next section

takes two imprecise values as inputs and executes addition of the given two imprecise values and finally stores the result in the form of an imprecise value. This instruction is a typical addition instruction for two imprecise values. 32C instruction is useful for addition between an imprecise value and an exact value through a program execution. 32C and 42C instructions are called *imprecise additions*. The imprecise addition can be executed without carry propagation and is executed independent of the bit width of the data. A subtraction for an imprecise computation is implemented in a similar way to imprecise addition [5].

To show the performance advantage of an imprecise addition, the sum of each computation time for addition - multiplication - addition instructions is shown in Figure 2 for a 64 bit data width. An optimized conventional adder takes log(bit-width) full adder delay in the worst case. So, in case of 64 bit data, the adder takes six full adder delays to complete an addition. Assume that a multiplication takes a certain delay, denoted by $D_\times$. Since the adder for an imprecise computation, a (4,2) counter, takes only two full adder delays, the total delay of the instruction flow will be “$2 + D_\times + 2m$” in the case of an imprecise computation while conventional computation requires “$6 + D_\times + 6m$”. In this simple comparison, the achieved by the imprecise computation is clearly observable and it can be said that the gain is coming from the fast imprecise additions. Furthermore, $D_\times$ is also reduced in an imprecise computation, which will be explained in the following subsection.

B. Multiplication for an Imprecise Data

In this section, a multiplier supporting the imprecise computation, an *imprecise multiplier*, is presented. A tree multiplier is considered as an imprecise multiplier. Imprecise multipliers can be classified as one of the four following types based on combinations of the input and output value types. Note that EV means an *exact value* and IV means an *imprecise value*.

- $EV \times EV = EV$, (Type EEE): This type of a multiplier is a conventional multiplier and it’s worst-case delay is the sum of the tree depth and the final carry propagation delay.
- $EV \times EV = IV$, (Type EEI): In this type of tree multiplier, the worst-case delay is only the delay of the tree logic. Since the result of the EEI type multiplication is an imprecise value, the final carry propagating addition is not needed. See Figure 4.
- $EV \times IV = IV$, (Type EII): Let $a$ be an exact value and $b$ an imprecise value represented by a pair of data, $(b_1, b_2)$. The result of $a \times (b_1, b_2)$ can be expressed as $(a \times b_1, a \times b_2)$ by the *distributive* law. To make a EII type multiplier, two EEI type tree multipliers are allocated for $a \times b_1$ and $a \times b_2$. Since these two EEI type multipliers produce four values using a (4,2) counter, the final result can be generated in the form of an imprecise value. Compared to the EEI type multiplier, the EII type multiplier additionally consumes 2 FADs in the final (4,2) counter.
- $IV \times IV = IV$, (Type III): The structure of the III type multiplier is constructed by doubling the EEI type multiplier and reducing the four values produced to two values in an imprecise value, using a (4,2) counter as shown in Figure 5. Thus, the delay in the III type multiplier is two full adder delays longer than that of the EII type multiplier.

Until now, the four types of tree multipliers are introduced and analyzed in the viewpoint of their delay. A
Figure 5: Multiplier Structure for Computing $EV \times IV = IV$ and $IV \times IV = IV$

Table 2: Performance of Four Tree Multipliers

<table>
<thead>
<tr>
<th>Multiplier Type</th>
<th>Worst Case Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEE</td>
<td>$8^4 + 6^4 = 14$</td>
</tr>
<tr>
<td>EEI</td>
<td>$8^4 + 7^2 = 17$</td>
</tr>
<tr>
<td>EII</td>
<td>$8^4 + 2^4 = 10$</td>
</tr>
<tr>
<td>III</td>
<td>$8^4 + 4^4 = 12$</td>
</tr>
</tbody>
</table>

1) : Tree Reduction Depth  
2) : Final Carry Propagation Delay  
3) : Counter Delay for generating an Imprecise Data

summary of the four type tree multipliers is shown in Table 2. The area of the EII type multiplier is about two times greater than that of the EEI type multiplier while the area of the III type multiplier is also about two times greater than that of the EII multiplier.

C. Loop Extension

Efficient computation of loop structure is another major issue in performance improvement. Introduction of imprecise computation to loop structure is addressed in this subsection.

Figure 6 shows computation in a loop containing very complex and mutually dependent computation between $u$ and $y$. For proper code scheduling, the instruction code generation consists of three parts: head computation, body computation, tail computation as shown in Figure 7.

**Head Computation:** Head computation is a preprocessing that transforms the incoming exact values from a loop into imprecise values, which are then used as input data for the body computation. For example, in Figure 7, consider the input data $x$, $dx$, $u$ and $y$. At first, since $dx$ is a constant, it remains as an exact value through the loop processing. Here note that the addition that determines the value of $x1$ is not on the critical path from the DFG in Figure 7 (a). Thus imprecise computation of $x1$ is not needed for the sake of area and performance. Finally, two exact values $u$ and $y$ are transformed into imprecise values, $uu$ and $yy$, and they are used as input values for the body computation, as shown in Figure 7. From the DFG shown in Figure 7, the code generated for the head computation is as follows:

1. Assume that $x$, $y$, $u$, $dx$, have valid data at this point.
2. Loading Phase: $x \rightarrow x1, y \rightarrow y1, dx \rightarrow dx$, $dx \rightarrow dx$.
3. Loop Structure:
4. Head Computation Code:
5. Node Insns.
6. $A00 \ R1, R4, R5 \ ; x = dx - R2$.
7. $22M1 \ R3, R4, (R6, R7) ; x = x + (R6, R7)$.
8. $32M1 \ R4, (R8, R9); y = y + (R8, R9)$.
9. $42M1 \ R5, (R10, R11); y = y + (R10, R11)$.
10. $A00 \ R5, R8, R9; \ ; uu = uu + (R8, R9)$.
11. $A00 \ R6, R7; \ ; yy = yy + (R6, R7)$.
12. $A00 \ R8, R9; \ ; uu = uu + (R8, R9)$.
13. $A00 \ R10, R11; \ ; yy = yy + (R10, R11)$.

In this example, the use of infinite logical registers is assumed. However, an optimized register allocation method can be applied to reduce the number of registers. The performance of this example is seriously limited by the instruction node chain, $\oplus_2 \rightarrow \oplus_5 \rightarrow \oplus_7 \rightarrow \oplus_9$, where each subscript corresponds to the each node number in the DFG. Since the operation $\oplus_7$ depends on the operation $\oplus_9$ in loop cycles, the length of serial instruction sequence significantly increases according to the number of
repetitions. This type of instruction chain is referred to as a “loop cycled critical instruction chain” (LCCIC). The total latency of the LCCIC is 46(17+17+6+6) FADs with a conventional architecture and 25(10+12+1+2) FADs with our imprecise computation based architecture.

**Body Computation:** Using the imprecise values that are obtained during the head computation phase, a repetitive code sequence was constructed. The following code is the body computation code for the example in Figure 7.

```assembly
; Body Computation Code
; x<-R1, dx<-R4, uy->(R18,R19). uy->(R11,R12)
LD R30, #9 ; R30 is used for INDEX "i"
BODYLOOP: ; 99 times repeat

:Node Insns.
1 ADD R5, R4, R1 ; dx = dx - R1
2 32ML (R18,R19), R4, (R6,R7); udx = (R6,R7)
3 32ML (R11,R12), R4, (R6,R7); ydx = (R6,R7)
4 42ML (R18,R19), (R11,R12), (R20,R21); uy = (R20,R21)
5 32ML R1, (R6,R7), (R22,R23); dx:node-2’s result = (R22,R23)
6 42C (R6,R7), (R11,R12), (R14,R15); node-2’s result + y
7 42CSUB (R22,R23), (R18,R19), (R24,R25); node-5’s result - u
8 42CSUB (R14,R15), (R20,R21), (R11,R12); compute "y" in an IC
9 42C (R24,R25), (R6,R7), (R18,R19); compute "u" in an IC
MOV R1, R6
SUBNZ R30, #1, R30, BODYLOOP
```

The differences between a head computation code and a body computation code stem from the availability of an *imprecise value* and the augmented repetition structure, such as a SUBBNZ instruction, where SUBBNZ is an instruction integrating a “subtraction” and “branch if non-zero”. Note that the total number of code repetitions is 99 since the first execution has been done in the head computation.

**Tail Computation:** Tail computation plays the role of synchronizing the imprecise values. It transforms the imprecise values into the exact values. For further optimization, data synchronization can be merged into the head computation of the next code block. The instruction sequence generated for the source code in Figure 7 is as follows:

```assembly
; Tail Computation Code
ADD R11, R12, R26 ; Data Synchronization for y1
ADD R18, R19, R27 ; Data Synchronization for u1
```

**E. Asynchrony on an Imprecise Computation**

Asynchronous designs are considered as an underlying design method for the imprecise computation architecture, shown in Figure 3, to maximize the effectiveness.

See the following two characteristics of the suggested architecture: (1) *some functional units supporting imprecise computations have a very short execution time* and (2) *the delay variation is very high among the various functional units*. From these two features, the necessities and advantages of using asynchronous design methods in implementing an imprecise computation are summarized as follows.

Firstly, by adopting asynchronous designs, locally optimized processing for various functional units can be realized. In particular, this feature is very beneficial to the imprecise computation because the suggested architecture has various functional units having very different execution delay which should be optimized individually in order to have better overall system performance. Secondly, to fully utilize the fast execution of the imprecise additions, the clock cycle time must be set to about one full adder delay for the case of using 32C instructions in synchronous designs. Consequently, other functional units should be pipelined in depth. For examples, a conventional synchronous 64-bit tree multiplier needs to be pipelined into about 17 stages and this deep pipelining cause a latency increase. In asynchronous designs, the execution of a functional unit can be partitioned into any number of stages without considering a clock cycle time constraint. Therefore, pipeline design difficulties caused by high variation of processing time among the functional units are eliminated. Finally, clock skew is caused by the fast clock cycle time, which is set to about one full adder delay for 32C instructions when synchronous designs are used. In this situation, the clock frequency reaches up to 2 GHz with a 0.35-μm technology or greater with
IV. Experimental Results

We first give analysis results for scalable examples to show the potential performance advantage of the suggested processor architecture. Secondly, to show the real effectiveness, two well-known data intensive applications are used in the simulation.

Figure 8 shows the speedup achieved by the suggested architecture comparing to a conventional architecture. In this analysis, other architectural processing delays (e.g., fetch unit delay, register access time, etc.) are assumed zero and the experiments are focused only on the processing time of functional units. We investigate the speedup under varying rate of arithmetic instructions, such as addition and multiplication, in a repeating LCCIC. In the experiment, we assumed that addition instructions consist of 20%, 40% and 40% of conventional, 32C and 42C additions, respectively. In similar, multiplications are composed of 10%, 20%, 40% and 30% of conventional, EEI, EII and III multiplications, respectively. The speedup is about 1.43 when the occurrence rate of arithmetic instructions in a repeating LCCIC is 70% and the occurrence rate of addition instructions from the arithmetic instructions is 60%.

The simulation results for two well-known codes, a differential equation solver code and a second order IIR filter code, are shown in Table 3. For the simulation, an asynchronous microprocessor simulator has been implemented in RTL-level using C++. Since we assumed that there is no resource limits, the performance is limited by data dependencies and the processing time of other modules (e.g., fetch unit, register file, etc.). As expected, we could observe performance improvement and our approach was proven to be a effective data processing technique. For the detailed information of delay value of each modules, please refer to [5].

Table 3: Performance Comparison for Practical Examples

<table>
<thead>
<tr>
<th>Application</th>
<th>Speedup achieved by our suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiffEq</td>
<td>1.35</td>
</tr>
<tr>
<td>IIR Filter</td>
<td>1.27</td>
</tr>
</tbody>
</table>

V. Conclusions

In this paper, an imprecise computation is newly proposed as a fast data computing technique and we have presented a new computation architecture supporting the imprecise computation. Compared to conventional architecture, the suggested architecture has functional units having faster processing time due to the imprecise computation. Furthermore, to maximally exploit the various processing time of functional units, an asynchronous circuit design is incorporated into the architecture. To show the potential performance benefits of our architecture, simulation results are also presented. An imprecise computation and the proposed architecture are expected to be used effectively in the data intensive processing of a microprocessor, a digital signal processor or a multimedia processor.

Currently, a hardware sharing method is considered for the high utilization of imprecise multipliers and more general performance evaluations have been being investigated.

VI. Acknowledgements

This work has been supported in part by the Korea Science and Engineering Foundation under grant 2006-302-01-2 and by the KAIST/K-JIST IT-21 Initiative in BK21 of Ministry of Education.

References