Vol.2009-ARC-184 No.3
2009/8/4

A Light Bypass Network Design for Cascading ALU Executions

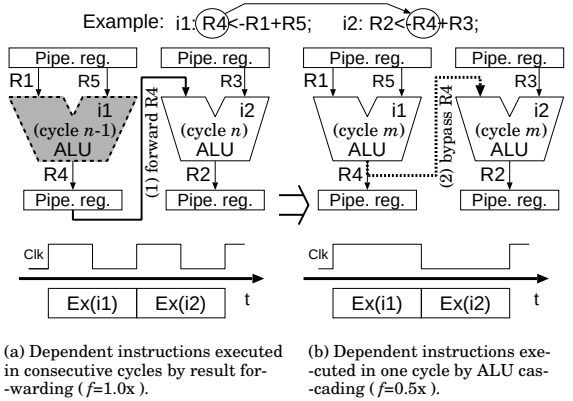
Jun Yao,[†1] Hajime Shimada,[†1] Takashi Nakada[†1] and Yasuhiko Nakashima[†1]

ALU cascading is a possible solution to reduce the processor energy consumption under low workload and low clock frequency executions. However, to sufficiently use all detected cascadable pairs for a better performance, a specific bypass network which provide internal results between simultaneously issued producer/consumer pairs is required. This added cascading bypass network complicates the designs of cascading enabled processors, especially when the delay and area of wires can not be neglected. In this paper, we present a light bypass network design which multiplexes the usage of the original forwarding bypasses in a superscalar processor. The arbitration scheme and possible performance penalties are studied in detail with our employed workloads. The results indicate that after applying several simple additional policies on the instruction issue, ALU cascading can still achieve a comparable performance increase with the low cost bypass design.

1. Introduction

ALU cascading[1),2)], also known as data collapsing[3)], is originally nominated as a performance enhancing method. Its basic idea is to collapse the execution of dependent instructions into a single cycle, so as to save the total execution cycles. However, as the cascaded execution of dependent instruction shall be kept in sequence and thus requires a longer execution cycle period, the application of ALU cascading is generally limited in media and vector processors or a globally-asynchronous locally-synchronous processor which can use different clock frequencies in individual pipeline stages[1)].

Recently, many of current power saving methods try to apply a lowered frequency to reduce power under a light workload. Among them, several mechanisms



(a) Dependent instructions executed in consecutive cycles by result for-warding (f=1.0x).

(b) Dependent instructions exe-cuted in one cycle by ALU cas-cading (f=0.5x).

Fig. 1 ALU Cascading.

such as pipeline stage unification (PSU) [4),5)] and dynamic pipeline scaling (DPS)[6)] are designed to be of voltage scaling free ones. Under these methods, it is possible to use the lowered frequency for cascading purpose, as described in paper 7). Since energy is the product of both power and execution time, the increased performance from cascading can help achieve further energy reduction when the hardware extension for the cascading can be controlled under a negligible level.

Figure 1 gives an illustration of ALU cascading by executing two dependent instructions ($i1$ and $i2$ in Fig. 1) under a halved frequency. Generally, as shown in Fig. 1(a), a normal superscalar execution will require two consecutive cycles to finish the two arithmetic operations as $i1$ and $i2$, by using the result forwarding path ("(1)" in Fig. 1(a)) to bypass value $R5$ from $i1$'s execution which was generated in cycle n-1. After the frequency is scaled to half of the maximum value in Fig. 1(b), a clock cycle period can hold two ALU operations if voltage keeps unchanged. The cascading enabled processor connects an output of an ALU to an input of another ALU and additionally bypasses $i1$'s result to $i2$ ("(2)" in Fig. 1(b)). By this means, ALU cascading can utilize the latter half of the clock cycle time for a second operation as depicted in the timeline in Fig. 1(b). It is expected to increase the Instructions Per Cycle (IPC) for the program execution and the regained performance under a lowered frequency can benefit the final energy reduction. If the performance portion is more weighted in metrics

†1 Nara Institute of Science and Technology

like energy-delay-product (EDP), the efficiency of cascading augmentation can be further amplified as compared to the original low frequency execution.

However, as shown in Fig. 1(b), a specific cascading bypass route as "(2)" is required in addition to the normal result forwarding route "(1)" to bypass the intermediate data in a cascading pair like $i1 \rightarrow i2$. As the delay and area of wire are also major concerns in designing modern microprocessors, this specific cascading bypass route may visibly increase the hardware cost since it has a similar complexity as the normal forwarding network. Under this consideration, we give a light bypass network design by multiplexing the usage of normal result forwarding routes. ALU cascading and normal result forwarding are still supported in this architecture while the retirement of the dedicated cascading bypass network can help achieve a similar hardware complexity for the execution stage as in normal superscalar processors. The additional delay and area penalties after supporting ALU cascading are thereby alleviated. Based on the light bypass network design, we studied several schemes to reduce the possible conflicts on the multiplexed single bypass route in detail.

The rest of this paper is organized as follows. Section 2 reviews instruction scheduling methods for superscalar processors that provide additional wakeup/selection features for ALU cascading, which serves as the background technologies of this paper. Based on the scheduling scheme in Section 2, Section 3 describes our proposal of a light bypass network for the execution stage to alleviate the hardware complexity while still supporting cascading execution. Section 4 presents the quantitative study of penalties after decreasing the hardware supports for cascaded executions. Finally, Section 5 concludes the paper.

2. ALU Cascading

2.1 Scheduling Schemes for Cascading

According to the definition of ALU cascading introduced in papers 2), 8)–10) and Section 1, cascadable instructions with dependence between them may be waken up simultaneously. For this purpose, the normal instruction scheduling method in a superscalar processor is required to be extended with cascading supports. The key implementation is either to combine producer and consumer instructions into a single package[9] or to move the data dependence of a consumer instruction from its direct parents to its grandparents[10]. In either method, from the viewpoint of the grandparent instruction, its issue will clear the corresponding dependence relationship in its direct and the nearest indirect consumers, and the cascadable producer/consumer can thus be waken up together if no other unresolved dependence is in presence.

Table 1 A sample program block.

No.	Instructions	No.	Instructions
$i1$:	R1 ← mem(R8+0);	$i2$:	R2 ← mem(R8+4);
$i3$:	R3 ← mem(R9+0);	$i4$:	R4 ← mem(R9+4);
$i5$:	R5 ← R1 + R2;	$i6$:	R6 ← R5 + R3;
$i7$:	R7 ← R5 + R4;	$i8$:	R6 → mem(R9+0);

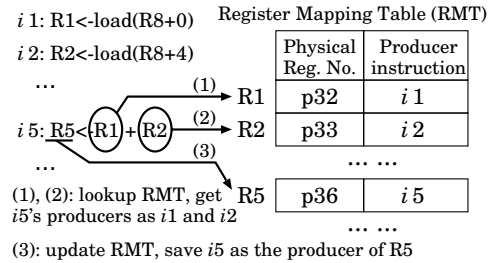


Fig. 2 An RMT implementation to trace the direct producer instruction, as well as the logical/physical register mapping.

We use the scheduling method in paper 10) as the baseline scheduler in this research. In this section, we use an example in **Table 1** to briefly demonstrate the scheduling policies to support ALU cascading.

Normally, there are two procedures performed in a decoding phase to identify data dependences between instructions. Firstly, using the decoding of $i5$ to $i8$ in Table 1 as an example, their direct producers can be back-traced by using the register map table (RMT) in **Fig. 2**. This procedure can detect the dependences between previously decoded instructions in the instruction window and those currently decoding ones, which are $(i1, i2) \rightarrow i5$, $i3 \rightarrow i6$, and $i4 \rightarrow i7$. Secondly, the dependences between $i5$ to $i8$ are checked by comparing their source and destination registers. By this step, dependences $i5 \rightarrow i6$, $i5 \rightarrow i7$, and $i6 \rightarrow i8$ can be identified.

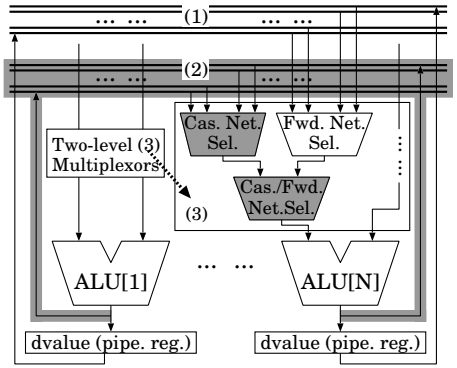


Fig. 3 A dedicated cascading design to support cascaded ALU executions.

Paper 10) employs the results of above two procedures to detect cascadable instructions. Since the simultaneously decoded instructions like $i5$ and $i6$ has a data dependence relationship as detected by comparing their source and destination registers, a two-hop away dependence chain can be established by using $i5$'s parents, which can also be obtained in this decoding phase, as $i1$ and $i2$. In this example, the two-hop dependence chain for $i6$'s left source operand is $(i1, i2) \rightarrow i5 \rightarrow i6$. According to the basic idea, $i1$ and $i2$—as the grandparents for $i6$—can be used instead of $i5$ to serve as the dependence resolving requirements for $i6$'s left source operand.

Under this situation, if $i3$ which is the producer of $i6$'s right source operand, is also issued at the same time of $i1$ and $i2$, $i6$ can be marked as ready at the same time of $i5$, which may achieve a cascading issue of $i5$ and $i6$ consequently.

This method only detects the cascadable chances among the simultaneously decoded instructions. However, it makes full use of the information obtained in the normal decoding phase so as to maintain a very small hardware cost for detecting cascadable instructions.

2.2 Operating with Dedicated Cascading Bypasses

The execution of cascadable instructions is another concern in a processor that supports cascading. Other than using 3-1 ALUs[11] which are specifically suitable for cascading but may visibly increase the size of ALU cells, additional links to pass cascading intermediate data is required as shown in Fig. 1. Under cascading, the source operands of a consumer instruction may come from either the register file, the bypassed data from previous execution cycles, and the output of the producer in the same execution cycle which represents the cascading intermediate data. For this purpose, a dedicated cascading bypass network is employed in paper 10) to specifically pass the cascadable intermediate values, with a similar topology as the normal bypass network.

Figure 3 shows the design of the execution stage with a dedicated cascading bypass network which is a detailed version of "(2)" in Fig. 1. The shadowed units are added specially for the cascaded execution purpose. In addition to the added network, the cascading bypasses present a third source for each ALU input. The multiplexer to select the correct source before ALU inputs is also require to be extended to a two-level hierarchy, depicted as "(3)" in Fig. 3.

3. A Light Bypass Network Design for EX stage

The design of execution stage introduced in paper 10) and Section 2.2 gives an implementation by using normal processor calculation resources for the cascading purpose, without including specific units like a 3-1 ALU which is specially designed to execute two cascadable instructions in a whole. A dedicated bypass network is used to pass inner data in a cascading pair so that the normal result forwarding will not be affected. However, since the delay and hardware cost of wires are also big concerns in modern processors, the added bypass network may introduce penalties which can not be simply ignored. For this consideration, in this research, we designed a method to multiplex the usage of normal data forwarding routes with cascaded execution supports, in order to prevent adding additional networks. This section gives a detailed introduction of this network design together with several combining schemes for arbitrating the traffics on this designed single network.

3.1 Multiplexing the Usage of Normal Forwarding Bypasses

Figure 4 presents the proposed light network design to handle both cascading and result forwarding. Different to the design in Fig. 3, the additional bypass network dedicated for the transferring of cascading intermediate values has been

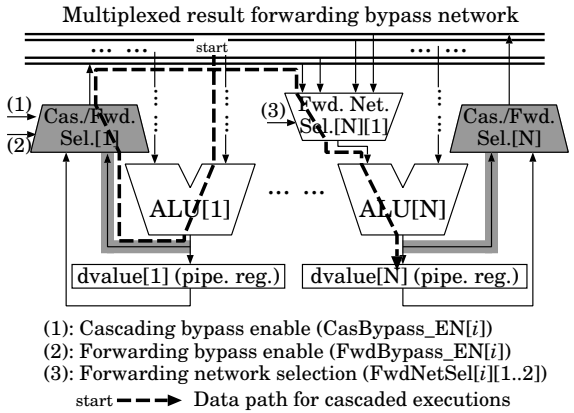


Fig. 4 Multiplex the usage of normal forwarding bypasses.

retired by putting another multiplexor *Cas./Fwd.Sel* before the data is passed onto the bypass network. Assuming that the execution is now working on the instructions in the current clock cycle n, the multiplexor *Cas./Fwd.Sel*[i] works to handle the selection between the output from the pipeline register *dvalue*[i] which stores the i-th ALU output of the cycle n-1, and the output from the i-th ALU of the current cycle n. With the help of the newly added *Cas./Fwd.Sel*, the usage of the original result forwarding network is now multiplexed. Either result forwarding data and cascading intermediate values can be provided from this single network. Without the dedicated cascading bypasses, the hardware extension with ALU cascading augmentation can thereby be largely alleviated, especially when the wire costs can not be neglected. Moreover, as shown in Fig. 4, it has a smaller number of providers for each ALU input after removing the additional cascading bypass network. The selection between the bypass network and ALU inputs can thus refrain from the extended two-level multiplexer hierarchy which was described in Fig. 3.

To pass correct values of either *dvalue* and ALU to the correct ALU input, the selection signals of *Cas./Fwd.Sel* and *Fwd.Net.Sel* shall be determined after ALU resources have been allocated to the selected ready instructions. **Figure 5** demonstrates the algorithms that determine the correct selection signals

```
(I) After issuing instructions to ALUs:
for (i=1;i≤N;i++) { /* provider index */
  /* Initialization */
  CasFwdSel[i].FwdBypass_EN = false;
  CasFwdSel[i].CasBypass_EN = false;
  /* Comparing and setting flags */
  for (j=1;j≤N;j++) { /* consumer index */
    for (k=1;k≤2;k++) { /* src1, src2 */
      if (dvalue[i].preg == ALU[j].src[k].preg) {
        /* Forwarding from dvalue[i], which was
         * produced by ALU[i] in the last cycle,
         * is required */
        CasFwdSel[i].FwdBypass_EN = true;
        FwdNetSel[j][k] = i;
      }
      if (i==j)
        continue;
      /* Compare only between different ALUs */
      if (ALU[i].dest_preg == ALU[j].src[k].preg) {
        /* Cascading from ALU[i] is required */
        CasFwdSel[i].CasBypass_EN = true;
        FwdNetSel[j][k] = i;
      }
    } /* end k loop */
  } /* end j loop */
} /* end i loop */
(II) After the above setting:
for (i=1;i≤N;i++) {
  if (CasFwdSel[i].FwdBypass_EN
    && CasFwdSel[i].CasBypass_EN)
      /* Confliction on the i-th bypass */
      Stall_Execution();
} /* end i loop */
```

Fig. 5 The determining of multiplexor selection (1), (2), and (3) in Fig. 4.

for all *Cas./Fwd.Sel* and *Fwd.Net.Sel* multiplexors. In Fig. 5, we use variable *CasFwdSel* to represent the multiplexer *Cas./Fwd.Sel* and its two elements of *FwdByass_EN* / *CasBypass_EN* to denote the selection signals. The notation of *FwdNetSel*[i][1..2] is adopted to denote the selection of multiplexor *Fwd.Net.Sel*

between the bypass network and each ALU input. $ALU[i].dest$ is the destination operand of the i-th ALU, while $ALU[i].src[1]$ and $ALU[i].src[2]$ are the two source operands of this ALU. In addition, we use the style of a suffix "*preg*" to represent the corresponding physical register number, for either ALU source and destination operands. Other expressions have the same meanings as in Fig. 4.

In this research, we assume a same superscalar design as in paper 10). A wire-or-style instruction scheduler as Dependence Matrices Table (DMT)[12] is employed as the baseline scheduler. In that architecture, DMT structure which stores data dependence information is used to accelerate the wakeup/selection phase. After the instruction selection phase, no DMT access is performed so that the issue logics in this architecture has no information of data dependences among the ready-to-issue instructions. The issue logic is thus required to detect correct data paths between issued instructions, which is the major task performed in Fig. 5.

With the supports of ALU cascading, there may be normal consumer instructions whose producers are executed in previous cycles, and cascadable pairs of which the producer and consumer will be both executed in this cycle. To detect correct producer/consumer relationships, two kinds of comparisons are performed, as shown in Fig. 5. One kind of comparisons is between the current ALU source operand numbers, and the destination register number in the last cycle by referring to the pipeline register *dvalue* directly after the EX stage. This is for the enabling of corresponding result forwarding route, from the execution cycle n-1 to the current cycle n. The other set of comparisons are made between the ALU outputs and their input register numbers, which is to help enable the bypassing of correct cascading intermediate values. Note that the algorithm in Fig. 5 is given in loop style for clarity. The real implementation of these comparisons can be finished by comparators in parallel so that they can be finished in a single cycle. The *CasBypass_EN* and *FwdBypass_EN* are the wire-or results of the corresponding comparator outputs.

The bypass network conflicts on a single bypass route: Though this design in Fig. 4 can sufficiently reduce the wire extensions for a cascading enabled processor, it may possibly introduce conflicts on the multiplexed single bypass route. The conflict happens when both result forwarding data and cascading intermediate data try to occupy the single bypass. Specifically, as shown in Part (II) of Fig. 5, if both $ALU[i]$ and $dvalue[i]$ are providers to ALUs in this cycle, the selection will fail on the corresponding i-th bypassing route.

Under this situation, a stall of the execution of those just issued instructions in cycle n will be triggered to maintain correct data paths, which is presented as function "*Stall_Execution()*" in Fig. 5. Strictly speaking, the signals *CasFwdSel* and *FwdNetSel* are determined after instruction issue phase and the direct phase following that is Register Read (RR) stage. The "*Stall_Execution()*" is then actually stalling the corresponding RR stage while the EX stage in the same cycle n is not affected. During the stall, instructions issued in cycle n-1 can still be propagated to latter stages, which helps retire the result forwarding from these propagating instructions. When the activity of RR stage is re-enabled in the cycle n+1, only cascading data will occupy those bypass routes and the conflicts are thus eliminated.

3.2 Policies to Reduce Bypass Network Conflicts

The conflicts on the multiplexed single bypass network will cause stalls of execution stage and then impede the performance increase achieved by ALU cascading. In this section, we study several policies for the instruction issue to reduce the possibilities of conflicts on the multiplexed bypass network.

Basically, the pipeline registers after free ALUs in the last cycle n-1 will not occupy the normal data forwarding bypasses in the current cycle n. It is thus safe to exclusively enable the cascading bypass in the corresponding *Cas./Fwd.Sel.* multiplexor. In our implementation, these free ALUs in cycle n-1 will be prioritized to issue ALU instructions in cycle n. However, if the resources of free ALUs in the last cycle can not fulfill all the ready-to-issue instructions, those non-free ALUs in cycle n-1 whose normal bypass routes may be enabled in the current cycle n will be used, which may be the potential sources of conflicts. For this purpose, we applied the following simple policies to the instruction issue to achieve conflict reductions in the multiplexed bypass network with relatively small hardware complexity[⋆1].

(a) **Prioritizing free ALUs in the last cycle (Baseline policy):** As de-

⋆1 Since instructions after selection contain no data dependence information, it is relatively difficult or too costly to apply sophisticated conflict reduction policies.

scribed above, basically we try to issue new instructions prior onto free ALUs in the last cycle. The corresponding *FwdBypass_EN* can always be set to *false* after these ALUs in the current cycle.

(b) **Special treatment for address generation instructions (Augmented policy (1)):** The augmented policies are considering the usage of ALUs which are not free in the previous cycle. Normally, the address generation part of a memory accessing instruction will use ALU resources. By assuming a separate load/store architecture, a memory instruction of $R2 \leftarrow mem(R8+4)$ can be regarded as a sequential execution of $tmp \leftarrow R8+4$ and $R2 \leftarrow mem(tmp)$. Though the first part may be the consumer in a cascadable pair, its ALU output is a hidden value to other instructions and will not be the provider for other cascading. Normal result forwarding can be safely enabled after its ALU output. This kind of instructions can be prioritized to issue onto those ALUs which are not free in cycle n-1.

(c) **Special treatment for the last ALU instruction in a decoding phase (Augmented policy (2)):** When the search range for cascadable instructions is designed to be among those simultaneously decoded ones as introduced in Section 2.1, the cascading opportunity detecting logics will stop at the decoding boundary. Under this situation, the last ALU instruction in the simultaneously decoded block will never become a producer instruction in a cascading pair, since its possible ALU output consumers are in latter decoding phases. Using $i7$ in Table 1 as an example, it is the last ALU instruction if $i5$ to $i8$ are decoded in one cycle. Other than ALU cascading, the wakeup of $i7$'s consumer instructions is guaranteed by the normal dependence resolving method and they will not be marked as ready until $i7$'s issue. The *CasBypass_EN* after $i7$ is always *false*. Similarly to the policy (1), this kind of instructions can also be prioritized to use the non-free ALUs in cycle n-1.

The two augmented policies (1) and (2) can be applied prior to the baseline policy and thereby save these resources of free ALU in cycle n-1. Our experiments show that by applying these two additional policies with a small hardware complexity, conflicts on the multiplexed bypass network can be largely reduced. The corresponding results will be studied in Section 4 in detail. Together with the stall check logics, we can retire the dedicated cascading bypasses to achieve a cost-effective implementation.

Table 2 Baseline processor configuration.

Processor	8-way out-of-order issue, 64-entry RUU, 32-entry LSQ, 8 int ALU, 4 int mult/div, 8 fp ALU, 4 fp mult/div, 8 memory ports
Branch prediction	10K-entry bimode[13)] (4K-entry x2 direction PHT, 2K-entry choice PHT, 12-bit history), 2K-entry BTB, 16-entry RAS, 10-cycle misprediction penalty
L1 I-cache	64KB/32B line/2-way
L1 D-cache	64KB/32B line/2-way
L2 unified cache	2MB/64B line/4-way
Memory	64 cycles first hit, 2 cycles burst interval
TLB	16-entry I-TLB, 32-entry D-TLB, 80 cycles miss latency

4. Simulation Results

The design in Section 3 can be regarded as a supporting implementation for microprocessors where ALU cascading is enabled[10)], in trying to keep the complexity of cascading hardware extensions toward a minimum level. However, the achievements of minimizing hardware cost consequently introduces some performance loss because of the shortage of bypass routes, as discussed in Section 3. Since the main purpose of supporting ALU cascading is to increase per cycle throughput of a superscalar processor, which is represented by Instructions Per Cycle (IPC)[⋆1], we will use IPC as a measure to study the trade-off by retiring the specific cascading bypass network.

4.1 Simulation Methodology

We used a detailed cycle-accurate out-of-order execution simulator—Simple-

⋆1 ALU cascading is enabled together with some energy saving methods which use a lowered frequency but maintain a fixed value of supply voltage. The improved IPC by cascading can help further the energy reduction in these kinds of energy saving architectures.

Table 3 Benchmark programs.

benchmark		baseline IPC	cas. ratio	benchmark	baseline IPC	cas. ratio
SPEC-int2000	bzip2	3.57	17.9%	gcc	2.14	5.7%
	gzip	1.77	9.2%	mcf	0.48	6.6%
	parser	1.56	7.8%	perlbmk	1.72	6.2%
	vortex	3.14	4.2%	vpr	1.50	9.6%
Media-bench	G721 decode	2.30	12.7%	G721 encode	2.00	14.1%
	GSM decode	3.10	17.1%	GSM encode	3.85	19.6%
	MPEG2 decode	2.92	15.1%	MPEG2 encode	1.43	8.4%

Scalar Tool Set[14] to measure the IPC improvements by enabling ALU cascading. Table 2 lists the configuration information of the baseline processor with an assumed 12-stage pipeline. In the simulation, we assumed a separate load/store architecture that divides the operation of memory instruction into address generation and memory access internal instructions. Thus, ALU cascading can be applied on both ALU operation instructions (SHIFTs are also included) and address generation instructions derived from memory accesses.

Table 3 lists the benchmarks which we used for evaluation. We chose eight benchmarks from SPECint2000, compiled with gcc Ver. 2.7.2.3 for SimpleScalar PISA. In addition, we selected six benchmarks from Mediabench[15], after excluding those too short programs which are of less than 50M instructions. The Mediabench programs are executed from beginning to end. For each SPECint2000 benchmark, 1.5 billion instructions are simulated with skipping first 1 billion instructions.

4.2 The Reduction of Bypass Network Conflict Rate by Using Instruction Issue Policies

This section mainly focuses on the effectiveness study of the designed two augmented policies (1) and (2), introduced in Section 3.2 for instruction issue, which are mainly used to alleviate the possibility of conflicts on the single multiplexed bypass network. Since the conflicts on the single bypass network will directly lead to a stall in the EX stage, we use the stall rate to measure these strategies, as in **Fig. 6**.

As shown in Fig. 6, in each benchmark, the horizontal axis gives four bars of different conflict reduction policies introduced in Section 3.2, which are the baseline

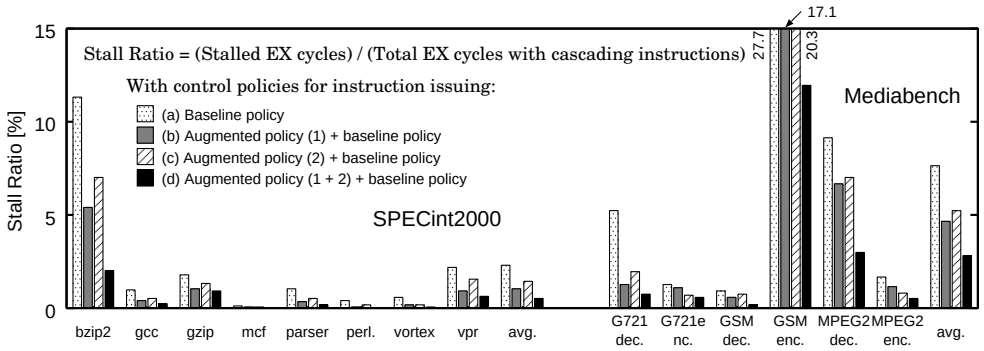


Fig. 6 Stall ratios on the single bypass network after applying different levels of conflict reduction policies in Section 3.2.

policy, augmentations with issue policy (1), (2) and (1 + 2), respectively. The vertical axis depicts the stall ratio, which is measured as the number of stalled cycles divided by the number of all execution cycles which have cascaded executions. The instruction scheduling scheme introduced in paper 10) and Section 2 are used as the baseline instruction scheduler for ALU cascading supports.

It can be observed that if the instruction issue is only with a baseline control, there will be visible stalls after using normal forwarding bypasses for both forwarding and cascading. The average stall rate will respectively be 2.3% for SPECint2000 and 7.7% for Mediabench. Particularly, the stall ratios in benchmark bzip2, GSM enc., and MPEG2 dec. have been over 10%. These stall rates have some correlations to both performance—measured as instructions per cycle (IPC) in Table 3, and the program specific cascading ratio which are shown in the 4th and 7th columns in Table 3. The cascading ratio is measured as the ratio between cascaded consumer instructions and the total instruction number in a cascading detecting among the simultaneously decoded instructions[10]. Benchmark bzip2, GSM enc., and MPEG2 dec. have comparably high IPCs and cascading ratios. These two characteristics may result in heavier traffics of both forwarding and cascading, which may increase the potential for conflicts on the multiplexed bypass network.

The only exception to the above explanation is in GSM dec., which has com-

paratively high IPC and cascading ratio like MPEG2 dec., while the stall ratios in these two benchmarks are quite different. Further tracing data indicate that the occupation rate of ALUs per cycle in GSM dec. is concentrating in the zone near half of the issue width, which are 3, 4, and 5 in our environment. Differently, MPEG2 dec. demonstrates a different trend as it experiences many execution cycles in a full-width ALU occupation which is 8 in our environment and many in a zero-ALU utilization as well. The IPC is an averaged data so that the differences in per cycle ALU occupation rate are concealed. From this viewpoint, ALU utilization is relatively well-balanced in GSM dec. It leads to an averagely larger number of free ALUs in the last cycle which generate no forwarding data to the current cycle. Bypass network conflicts are therefore less likely to occur in GSM dec.

After applying augmentation policies (1), the conflicts will be largely reduced. The average stall rate can be reduced to 1.1% in SPECint2000 and 4.6% in Mediabench. For most benchmarks, this policy (1) is sufficiently effective. Except the three mentioned benchmarks with largest stall rates, conflicts on the single bypass route will be reduced to near 1.0%. As shown in the fourth bar of each benchmark, policy (2) also reduces some stalls but the achievement is less than policy (1) since it only applies to the last ALU instruction in one decoding phase. However, these two policies can be incrementally applied. As shown in the fourth bars, the increment of policy (1 + 2) can reduce the conflicts in bzip2, GSM enc. and MPEG2 dec. to near 1/3 of the maximum values. The new average stall rates are 0.5% and 2.8% in SPECint2000 and Mediabench, respectively. With these two significantly reduced stall ratio, the performance penalty caused by the conflicts may be alleviated.

4.3 IPC Improvements

The final goal of this research is to achieve a smaller hardware cost in implementing ALU cascading while keeping a comparable performance gaining by cascaded executions. In this section, the performance loss caused by the simplified hardware supports will be studied by using IPC measurements.

Figure 7 shows the IPC improvements which are contributed from the cascaded instructions. These IPC improvement data along the vertical axis are normalized by those baseline IPCs shown in Tab. 3, collected from executions without cascading. We conducted five different sets of networks and issue policies according to Section 3.2. The first bar in each benchmark represents the result with a dedicated cascading network design. The following four bars in each benchmark are results with a single multiplex network, which have the same format of Fig. 6.

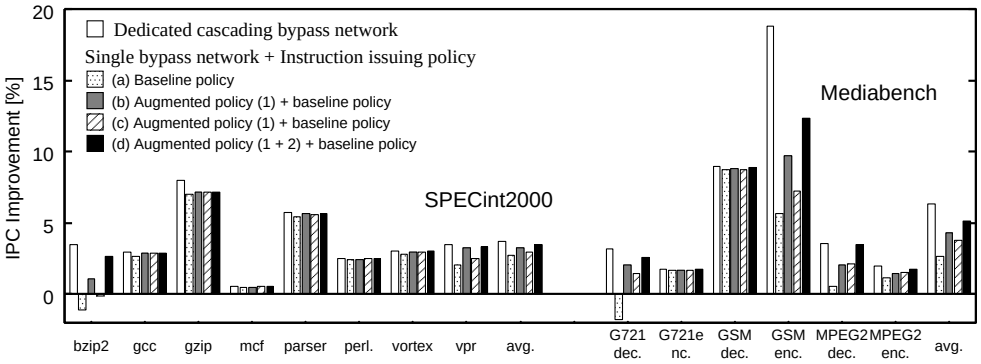


Fig. 7 Normalized IPC improvements based on different cascading bypass implementations.

As shown in Fig. 7, the allowed hardware complexity will have impacts on the final IPC improvements. With a dedicated cascading bypass network, the average IPC improvement under a search range of simultaneously decoded instructions becomes 3.7% in SPECint2000 and 6.4% in Mediabench, respectively. After reducing the hardware complexity by using a single bypass network, the gained IPC improvements will be largely reduced if there is only a baseline policy for the instruction issue logic, as shown in the second bar of each benchmark. The performance loss is correlated to the stall rate in Fig. 6. In some benchmarks like bzip2 and G721 dec., the performance is even smaller than the baseline processor which runs without cascading supports. This is because the stall after detecting a conflict on a single bypass will halt the whole EX stage, which affects up to 8 instructions in our environment.

The performance loss will be alleviated after incrementally applying conflict reduction policies. As shown in Fig. 7, policies (1) and (2) together can effectively add back the performance gainings toward the dedicated bypass design. The large gap is only in benchmark GSM enc., which is caused by its 11.9% stall ratio after

these two issue policies. In average, with both (1) and (2), the IPC improvement will be respectively increased to 3.5% and 5.1% for those two benchmark sets, which are 94.6% and 79.7% of the maximum values. After these two policies, the single network design can finally be regarded as applicable.

5. Conclusions

In this paper, we proposed a simplified bypass network to retire the dedicated cascading bypassing routes for microprocessors which support ALU cascading. The designed network uses the original result forwarding structures to incrementally cover the data bypassing of intermediate values in cascaded instruction pairs. In addition, schemes to reduce the possibility of cascading and normal result forwarding conflicts on the single bypassing route are studied in detail. After applying the conflict reduction schemes, in average, only 0.5% and 2.8% of the execution cycles with cascading in SPECint2000 and Mediabench are respectively required to be stalled due to the insufficiency of bypassing routes. Originally, a processor with dedicated cascading bypasses to support ALU cascading can achieve 3.7% and 6.4% IPC improvements in the two sets of benchmarks, respectively. Using the minimum hardware extensions given in this research, we can still achieve 3.5% and 5.1% IPC improvements.

Acknowledgments This work is partially supported by JST CREST.

References

1) H. Sasaki, M. Kondo, and H. Nakamura. Dynamic instruction cascading on gals microprocessor. In *In Proc. of Int. Workshop on Power and Timing Modeling, Optimization and Simulation 2005, Lecture Notes in Computer Science 3728*, pages 30–39, September 2005.
2) K.Kise, T.Katagiri, H.Honda, and T.Yuba. A super instruction-flow architecture for high performance and low-power processors. In *Proc. of the Int. Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'04)*, pages 10–19, January 2004.
3) Yiannakis Sazeides, Stamatis Vassiliadis, and JamesE. Smith. The performance potential of data dependence speculation & collapsing. In *Proc. of the 29th Annual Int. Symp. on Microarchitecture*, pages 238–247, December 1996.
4) Hajime Shimada, Hideki Ando, and Toshio Shimada. Pipeline stage unification for low-power consumption. In *Int. Symp. on Low-Power and High-Speed Chips (COOL Chips V)*, pages 194–200, April 2002.
5) Hajime Shimada, Hideki Ando, and Toshio Shimada. Pipeline stage unification: A low-energy consumption technique for future mobile processors. In *Proc. of the Int. Symp. on Low Power Electronics and Design 2003*, pages 326–329, August 2003.
6) Jinson Koppanalil, Prakash Ramrakhyani, Sameer Desai, Anu Vaidyanathan, and Eric Rotenberg. A case for dynamic pipeline scaling. In *Proc. of the 5th Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 1–8, October 2002.
7) Jun Yao, Hajime Shimada, Kosuke Ogata, Shinobu Miwa, and Shinji Tomita. Improving effectiveness of pipeline stage unification via alu cascading. In *Proc. of the 12th IEEE Symposium on Low-Power and High-Speed Chips (CoolChips XII)*, pages 423–436, April 2009.
8) M.Ozawa, M.Imai, Y.Ueno, H.Nakamura, and T.Nanya. Performance evaluation of cascade alu architecture for asynchronous super-scalar processors. In *Proc. of the 7th Int. Symp. on Asynchronous Circuits and Systems*, pages 162–172, March 2001.
9) H. Sasaki, M. Kondo, and H. Nakamura. Energy-efficient dynamic instruction scheduling logic through instruction grouping. In *Proc. of the Int. Symp. on Low Power Electronics and Design 2006*, pages 43–48, October 2006.
10) Jun Yao, Kosuke Ogata, Hajime Shimada, Shinobu Miwa, Hiroshi Nakashima, and Shinji Tomita. An instruction scheduler for dynamic alu cascading adoption. *IPSJ Transactions on Advanced Computing Systems*, 26(1-2), 2009.
11) J.Phillips and S.Vassiliadis. High-performance 3-1 interlock collapsing alu's. *IEEE Transactions on Computers*, 43(3):257–268, March 1994.
12) M.Goshima, K.Nishino, Y.Nakashima, S.Mori, T.Kitamura, and S.Tomita. A high-speed dynamic instruction scheduling scheme for superscalar processors. In *Proc. of the 34th Annual Int. Symp. on Microarchitecture*, pages 225–236, December 2001.
13) C-C. Lee, I-C. Chen, and T.Mudge. The bi-mode branch predictor. In *Proc. of the 30th Annual Int. Symp. on Microarchitecture*, pages 4–13, December 1997.
14) Doug Burger and ToddM. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison Computer Sciences Dept., July 1997.
15) Chunho Lee, Miodrag Potkonjak, and WilliamH. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual Int. Symp. on Microarchitecture*, pages 330–335, December 1997.