A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy

Angshuman Parashar, Sudhanva Gurumurthi, and Anand Sivasubramaniam
Dept. of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802
{parashar,gurumurt,anand}@cse.psu.edu

Abstract

Previous proposals for implementing instruction-level temporal redundancy in out-of-order cores have reported a performance degradation of upto 45% in certain applications compared to an execution which does not have any temporal redundancy. An important contributor to this problem is the insufficient number of ALUs for handling the amplified load injected into the core. At the same time, increasing the number of ALUs can increase the complexity of the issue logic, which has been pointed out to be one of the most timing critical components of the processor. This paper proposes a novel extension of a prior idea on instruction reuse to ease ALU bandwidth requirements in a complexity-effective way by exploiting certain interesting properties of a dual (temporally redundant) instruction stream. We present microarchitectural extensions necessary for implementing an instruction reuse buffer (IRB) and integrating this with the issue logic of a dual instruction stream superscalar core, and conduct extensive evaluations to demonstrate how well it can alleviate the ALU bandwidth problem. We show that on the average we can gain back nearly 50% of the IPC loss that occurred due to ALU bandwidth limitations for an instruction-level temporally redundant superscalar execution, and 23% of the overall IPC loss.

Keywords: Complexity-effective design, Instruction Reuse, Temporal Redundancy.

1. Introduction

Higher transistor integration densities, coupled with increasing clock frequencies and lower operating voltages, make reliability an important consideration for future processor designs. Deep submicron process technologies and lower transistor threshold voltages make circuits more susceptible to soft errors [8] caused by cosmic ray strikes [36]. It is no longer just the data stored in memory cells that needs to be protected (using parity, ECC, etc.). Technology trends mandate verifying the integrity of the combinational circuits within the processor datapath as well [28]. Consequently, several research and developmental efforts in the past decade have embarked on enhancing datapath reliability against transient errors [26, 20, 25, 33, 24, 15]. Reliability enhancements are typically provisioned using spatial and/or temporal redundancy mechanisms, which require additional hardware, additional complexity in design, and/or incur performance penalties. If we are not careful, provisioning such redundancy can defeat the purpose of the fundamental drive towards technological innovations for high performance. With this underlying philosophy in mind, this paper presents a technique for narrowing the performance gap between an instruction-level temporally redundant out-of-order execution and an execution on a normal out-of-order core without any temporal redundancy.

Hardware reliability enhancement is usually implemented with spatial or temporal redundancy. In spatial redundancy, hardware units are replicated and the same workload is run on each of them to verify that they provide the same results. The granularity of the units that are replicated can vary from simple functional units to complete pipelines (e.g. [16]). There are also spatial redundancy techniques where one can use a different hardware unit to verify the integrity of the output produced by the actual hardware (e.g. [4]). While one can argue that spatial redundancy is a useful way of exploiting future billion-transistor capabilities, the downside is the possible design complexity and the possibility of those extra transistors being used to boost performance in the first place. On the other hand, in temporal redundancy [31], the operations are performed multiple times (though temporally separated) on the same hardware to verify the outcome. While temporal redundancy is an effective use of the available hardware, without requiring additional support, the downside is the performance penalty of slowing down the normal execution. As a trade-off between these extremes, one would like to take a temporal redundancy mechanism, provide some minimal hardware support (not as much as what spatial redundancy mandates), and make its execution approach that of the spatially redundant execution. This raises the following questions: *What is the needed hardware support? How do we ensure that this hardware does not significantly complicate design? How much of a performance boost can it provide?*

Temporal redundancy can again be implemented at different granularities: instruction-level [20, 24] or thread-level [26, 25, 33, 15]. In our terminology, we refer to proposals that exploit hardware supported multithreaded architectures for temporal redundancy as being thread-level, and proposals such as [20, 24] which implement mechanisms in an ordinary out-of-order core as instruction-level (with temporal separation between the two executions being more fine-grained). In the past few years, thread-level temporal redundancy has been extensively investigated with several promising proposals for enhancing its performance. The problem has been more difficult when implementing instruction-level temporal redundancy without significant performance conse-

quences. Since the resources in an out-of-order (OOO) core have been tuned with one instruction stream in mind (referred to as Single Instruction Execution, or SIE, henceforth), resource contention becomes a serious impediment to performance when these instructions are replicated temporally (we focus on Dual Instruction Execution mechanisms in this paper, which we refer to as DIE). Previous studies [24] have reported up to 45% performance loss for SPEC2000 applications for DIE compared to SIE. Any effort to alleviate this loss, without significantly complicating hardware design, can have considerable impact on future processor designs. It can be directly employed in a superscalar design to enhance reliability without significantly degrading performance. It can also be used to enhance the OOO core within a SMT or CMP architecture, to either supplement or even replace the coarser-grain thread level redundancy mechanisms. The evaluations in this paper mainly focus on a superscalar design.

Contention for different resources in the OOO core, including the Reorder Buffer (ROB), the ALUs, and the issue bandwidth, in the DIE execution can cause longer delays thereby slowing down the execution compared to SIE (note that a previous study has shown ways of not inducing additional memory traffic [24], and consequently our focus is primarily on the datapath components). While it may be possible to increase some of these resources to ease such contention in future billion-transistor designs, at the core of the problem lies the functional units/ALUs[1] that need to be shared between the instruction streams and the issue unit that needs to schedule those ALUs to the waiting instructions. It has often been argued [22, 17] that this functionality can become quite complex to design, and is not very scalable. Even though it may be possible to add more ALUs, the consequence is that: (i) the issue unit needs to schedule more ALUs, and (ii) the outputs of these ALUs have to feed back to the issue window to provide data forwarding for waiting instructions. The issue logic design complexity obviously grows with the number of ALUs that we provision, making it a less desirable option for reducing the gap between DIE and SIE, since this has been shown to be on the critical path.

An alternative is to use a clustered architecture – where there are separate (dual) issue units scheduling separate sets of ALUs – and direct the primary and secondary streams to different clusters. Partitioning resources into clusters, however, leads to problems such as load imbalance, limited ILP within each cluster and long inter-cluster communication delays for SIE [10, 5, 1, 3]. On the other hand, replicating resources to form clusters almost resembles a spatial redundancy approach, leading again to the criticism that we may have been able to use such additional hardware for improving SIE.

Without going to the extreme of replicating many of the hardware resources, and at the same time attempting to simplify design complexity, this paper presents a novel adaptation of a hardware technique (that has been previously examined in performance optimizations of single streams), called *instruction reuse*, to bring a DIE on an OOO core closer to the performance of SIE. The concept of instruction reuse for optimizing SIE was first proposed in [29], and is based on the following observation: when encountering an instruction which was executed at some time in the past, we could directly use the output of the previous execution (instead of re-executing it) as long as the operands match. A simple cache (referred to as Instruction Reuse Buffer or IRB in this paper) of instructions and their operands and result-values, can be used to exploit such reuse [29].

An IRB can not only speed up the execution of a multi-cycle instruction, but can also enhance the overall execution bandwidth by allowing multi-ported lookups (resembling multiple ALUs). Previous studies [29, 12, 13] of IRB for single streams have pointed out that it is more useful for speeding up long latency operations, rather than the execution bandwidth, since the OOO cores have already been designed in a balanced manner for bandwidth. As a result, the work on IRBs gradually evolved into specialized mechanisms (such as value prediction) targeting long latency operations. While an IRB may seem like an effective way of amplifying ALU bandwidth without impacting the scheduling costs of the issue logic, it still does not solve the problem of having to propagate the results (from the lookup) back to waiting instructions.

Using the IRB as a starting point, this paper makes the following main contributions:

- We illustrate the use of an IRB to ease the ALU resource contention problem of a DIE system. The IRB is used to not only speedup the long latency instructions, but to amplify the execution bandwidth, which is extremely useful to handle the doubling of load imposed in a DIE system. Whenever possible, we direct the duplicate stream through the IRB to ease the ALU requirements. At the same time, we point out that the IRB does not need to be protected from faults with any additional hardware mechanisms.
- We show how this amplification of execution bandwidth can be achieved without requiring an increase in issue width (and the scheduling costs). Consequently, it is a simple hardware extension over the resources that one would provision anyway for a balanced SIE system.
- We identify an interesting property of a DIE system, wherein we can use the output from instructions of one stream as input operands for another, without really compromising on the desired level of temporal redundancy. This property helps us devise an IRB that does not require its outputs to feed back into the issue window, consequently not affecting the issue logic design complexity. We give details on the implementation of such an IRB, together with its integration with the rest of the datapath.
- Using several applications from the SPEC2000 suite, and a detailed cycle-level model, we demonstrate how our enhancement can narrow the performance gap between DIE and SIE.

The rest of this paper is organized as follows. The next section reviews the prior DIE proposal [24], together with an experimental study showing the bottlenecks of the DIE execution. The details of our hardware enhancement are given in section 3. Section 4 presents experimental results with our enhancements. Finally, section 5 summarizes the contributions of this paper.

2. Motivating the Need for Boosting ALU Bandwidth on a DIE

2.1. Provisioning Dual-Instruction Execution (DIE) in a Superscalar Processor

Temporal redundancy in a superscalar processor is implemented by first defining a *Sphere of Replication (SoR)* [25]. All components within the SoR are protected via redundant execution

1 In the platform that is modeled, branch target calculations are handled by the ALUs, and so are memory address calculations (memory accesses themselves are not something we are trying to optimize). Consequently, we use the terms *functional unit* and *ALU* synonymously in this paper.

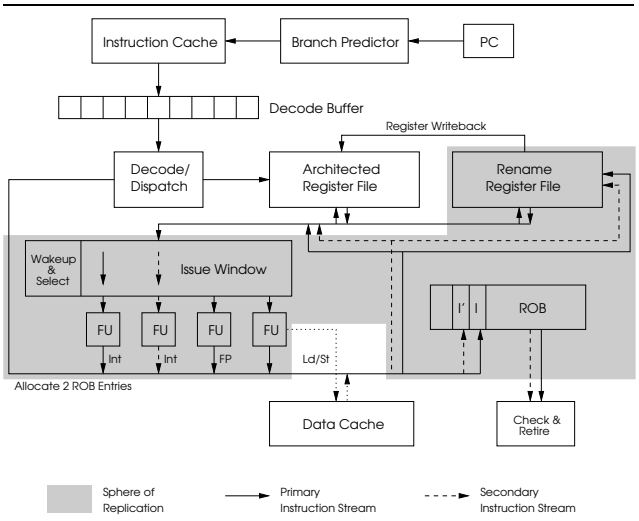


Figure 1. Implementing DIE in a Superscalar Processor, as proposed in [24]. The gray shaded area is the Sphere of Replication. The solid lines show the flow of the primary instruction stream and that for the secondary/dual stream is given in dashed lines. The access to the data-cache initiated by either primary or dual, though not both, is shown in dotted lines.

and any additional spatial/informational redundancy is not necessary for them. Any inputs coming in to the Sphere and the outputs exiting the Sphere need to be checked for inconsistencies. In the context of superscalar processors, previous proposals have implemented the SoR by instruction duplication [20, 24]. In [20], it is assumed that the program counter (PC), decode logic, register file, rename tables, and ROB are outside the SoR, and hence trustworthy. In [24], the authors bring the ROB into the SoR as well. They, however, leave the PC and branch-prediction structures outside the SoR since control-flow errors can be detected at the time the respective branch-instructions are resolved (i.e. though outside, these structures can still be considered safe for program execution). This modified superscalar processor, which we refer to as DIE in the rest of this paper, is shown in Figure 1. At decode/dispatch, each instruction is duplicated, whereby two adjacent entries in the ROB are created. The primary instructions and their duplicates are then dispatched to the issue window and both of them complete independently based on the dataflow order of their respective streams (there is no communication between the primary and secondary streams). At the commit point, the corresponding primary-dual pairs of instructions are checked against each other to detect any inconsistencies. If no inconsistency is detected, the architected instruction is retired; otherwise an instruction-rewind is triggered (using the existing mechanism that is used to recover from incorrect speculation) from the inconsistent instruction.

In the proposed DIE [24], instruction-level temporal redundancy has been provisioned without significantly additional hardware resources. The two streams (primary and dual) share the ROB capacity, the ALUs, etc. Consequently, an IPC loss (shown to be around 30% on the average for some SPEC2000 and SPEC95 applications) is to be expected compared to SIE. However, the dual execution is exploited for reducing the overhead of branch mispredictions and memory accesses. For instance, as soon as one stream detects a branch misprediction (instead of waiting for each to find out), the instructions for both streams along the mispredicted path are squashed and execution is initiated along the correct path. Similarly, since the memory system is assumed to be outside the SoR, for loads/stores, only the memory address calculation is performed for both, and the actual access is performed only once. This optimization also allows for preserving the semantics of uncached accesses and strong memory-ordering, that are typically required for memory-mapped devices.

2.2. Impact of Hardware Resources

To motivate the rest of this paper, we first conduct a set of experiments to understand the impact of the extra load imposed by a DIE system, compared to SIE. There are several hardware resources in the SoR (e.g. ALUs, Issue Window, ROB) shared by the two streams, which need to handle this extra load. In general, we observed that by amplifying the number of ALUs (that actually execute the instructions), the issue width (which determines the maximum number of instructions that can be sent for execution to the ALUs every cycle), and the RUU[2] size (which determines the number of instructions in flight in the processor), we can gain back most of the IPC loss for DIE compared to SIE. Consequently, we focus on these three parameters.

The base configuration of the SIE and DIE under consideration are presented later in section 4, and the important parameters to note are the number of ALUs (4 integer adders, 2 integer multipliers/dividers, 2 floating point adders, 1 floating point multiplier/divider/square-rooter), the RUU/Load-Store queue size (128/64-entries) and the issue width (8). We consider configurations that double the quantity of each of these hardware units, namely 8/4/4/2 ALUs (called *DIE-2xALU*), 256 RUU-entries and 128 Load-Store Queue entries (called *DIE-2xRUU*), and a decode/issue/commit width of 16 (called *DIE-2xWidths*). This gives us the following 7 configurations: *DIE-2xALU*, *DIE-2xRUU*, *DIE-2xWidths*, *DIE-2xALU-2xRUU*, *DIE-2xALU-2xWidths*, *DIE-2xRUU-2xWidths*, and *DIE-2xALU-2xRUU-2xWidths*, which together with the base DIE are compared with the IPC of the base SIE (i.e. the base DIE and SIE have the same capacity of resources for these units). In Figure 2, we plot the IPC slowdown for these configurations with respect to the base SIE for 12 applications from SPEC2000.

Performance degradation of DIE is anywhere from 1% (in *ammp*) to nearly 45% (in *art*) with 22% on the average compared to SIE. This reiterates the earlier observation that temporal redundancy imposes a high performance penalty on the system. At the same time when we double the capacity of the three aforementioned components (*DIE-2xALU-2xRUU-2xWidths*), we are able to achieve IPCs very close to that of the SIE system (note that the rightmost bar is very short and may not even visible for many benchmarks in the figure). This reiterates the importance of these three factors in determining performance.

While doubling the capacity of each of the three units improves IPC, we find that, among the three, doubling the number of ALUs (the *DIE-2xALU* scheme) provides the maximum reduction in the IPC loss (13% loss in IPC on the average, compared to 16% for the RUU and 21% for the widths). An exception to this is *art* where

2 SimpleScalar, where our simulations are conducted, models a unified ROB and issue window, called a Register Update Unit (RUU).

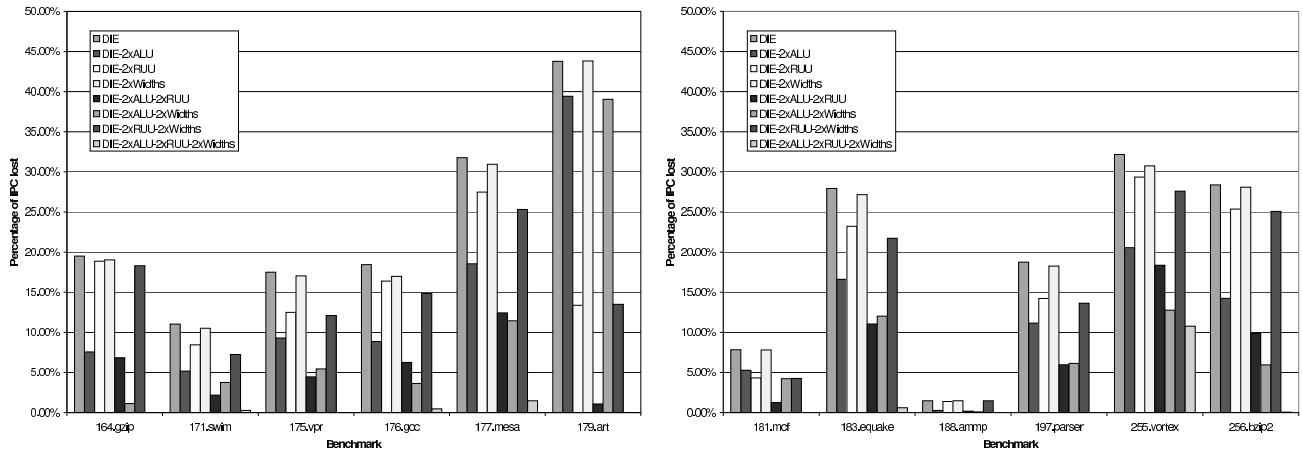


Figure 2. Percentage IPC Loss with respect to SIE

we find that doubling the capacity of the RUU provides greater savings possible because this exposes more ILP to the hardware for exploitation in this low IPC application (its IPC for SIE and DIE are 0.7316 and 0.4113 respectively). In many other cases, just doubling the ALU capacity provides better improvement than doubling all the other factors. Further, even when higher capacity in the other units can improve performance, an improvement in ALU bandwidth can further amplify their gains (note the bars for the different configurations where the ALU capacity is doubled). Rather than go into the details for each application to explain these results, our point is to merely note the need for amplifying the ALU bandwidth on a DIE, which is the focus of this paper. Future work can look at the other resources. At the same time, we want to develop a solution for ALU bandwidth which does not affect the scalability of the system, especially the issue logic.

3. Microarchitectural Design of IRB-Based DIE

In the previous section, we observed that a large fraction of the IPC loss in DIE was primarily due to the bandwidth limitation of the functional units/ALUs. Two possible solutions to this problem are:

- *Increase the number of ALUs:* Though intuitive (and as the previous section shows, the rewards can be substantial in an idealistic setting), the problem with this solution is the design complexity of the the wakeup, selection, and bypass logic, which are in the critical path.
  - *Selection Logic Complexity:* The issue logic needs to allocate a larger number of ALUs to the ready instructions, thereby increasing the scheduling complexity. If one goes for stacked arbiters for the selection logic, increasing the ALUs increases the stacking depth linearly. On the other hand, an unstacked implementation increases the size of each arbiter cell, whose consequences can be even worse than a stacked implementation [21].
  - *Wakeup and Bypass Logic Complexity:* The output from each of these ALUs needs to feed back to dependent/waiting instructions in the issue window. The consequent quadratic growth in delay (due to the additional complexity in wakeup and bypass logic) tends to be on the critical timing path of the processor [22, 17, 14].
- *Use a Clustered Architecture:* A decentralized clustered architecture can avoid the design complexity of a centralized large pool of ALUs. However, clustered processors tend to suffer from problems of limited ILP within each cluster (compared to a monolithic design) and long inter-cluster communication delays. In addition, currently proposed instruction-steering policies [10, 5, 1, 3] also cause load-imbalance between the clusters. Even if one uses separate clusters for each stream (primary and duplicate), this still requires replicating the issue window and the register file, which borders on spatial redundancy rather than a temporal redundancy approach. This goes back to the question of whether one could use those extra resources for enhancing even SIE. Though this may be a reasonable option, we postpone such an investigation for future work.

Instead, our solution for amplifying ALU bandwidth uses a novel adaptation of an approach that has previously been proposed to minimize resource conflicts on a SIE system for the functional units, namely, *Dynamic Instruction Reuse* [29].

3.1. Dynamic Instruction Reuse

Dynamic Instruction Reuse or, simply, Instruction Reuse (IR) is a non-speculative technique and is similar to the concept of memoization. It is based on the observation that there tends to be significant reuse of the instructions in a program, and when an instruction appears again with the same input operands, it will produce the same result as before. IR attempts to exploit this property by buffering previously executed instructions in a small hardware table called an Instruction Reuse Buffer (IRB), indexed by the PC [29] (there have been variations proposed [11, 12]). Every instruction performs a lookup of the IRB to check if it has an entry, and if so, whether its input operands match those in its entry (called a reuse test). If there is a match ("IRB-hit"), then the instruction does not require a functional unit and it re-uses the value from the previous execution. If there is a miss, it has to execute on the functional units. IR was initially proposed to overcome the

dataflow limit for collapsing true dependencies and to minimize resource conflicts [29, 30]. In a follow-up research, Citron et. al. [12] found that IR is effective only for long-latency operations. Since then, IR research evolved more into the study of value prediction [19, 18].

In this paper, we re-visit IR in a new light. In a DIE-based system, we observe that the primary cause for the loss in IPC is more due to *large number of single-cycle instructions contending for a small number of functional units*, rather than the presence of long latency operations, since we have twice the load being imposed on the system.

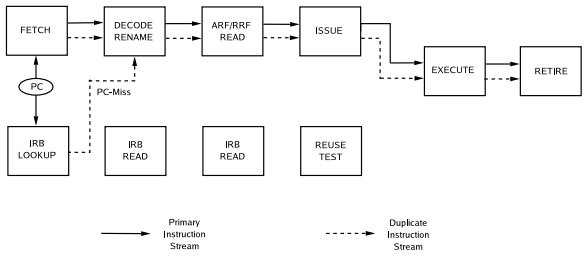
We employ the IRB to relieve this extra load imposed by the duplicate instruction stream in the following ways to reduce the overheads that temporal redundancy imposes.

- In our DIE enhancement, referred to henceforth as DIE-IRB, the primary stream is always executed by the functional units as in SIE. Instructions in the duplicate stream, on the other hand, first look up the IRB. If there is a hit, they skip the functional unit and move onto the completion stage of their pipeline. If they miss, they contend for the functional units as before. If there is good instruction reuse, then the pressure on the functional units can be reduced.
- The port requirements of the IRB can be kept low (comparable to an IRB designed originally for SIE) in order to achieve a fast access latency. This does not produce significant contention for the IRB ports since (i) only the duplicate stream accesses the IRB and (ii) the effective dispatch/commit width of a DIE system is half of that of a SIE system.
- As in a normal DIE, each pair (primary, duplicate) of instructions needs to be checked before retirement. The IRB does not need any special/extra protection, since temporal redundancy is provisioned for it by the primary stream executing in the functional units and vice-versa. Therefore, *the IRB lies within the Sphere of Replication* of the processor.
- Our IRB enhancements for a DIE do not incur the problems that arise when increasing the number of ALUs, that were identified earlier.
  - The number of ALUs that the selection logic needs to schedule for remains the same as in SIE.
  - In previous proposals that exploit Instruction Reuse on SIE, the IRB behaves like a functional unit and would therefore broadcast any results from a hit (from all its output ports) to its dependent instructions in the issue window. This would entail additional complexity to the issue logic, in terms of tag/result forwarding lines from the IRB to the issue window, thereby increasing the complexity of the wakeup-logic. However, using some properties unique to DIE, we show how an IRB can be incorporated in the datapath with *no extra forwarding-buses* and very little additional overhead to the issue logic.
- Finally, we incorporate a simple mechanism that can possibly reduce conflict misses in the IRB to further improve performance of DIE.

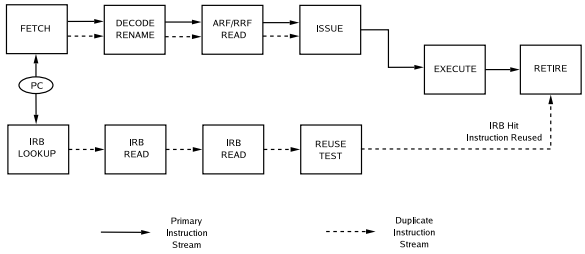
The rest of this section gives more details on these issues, together with the hardware modifications we propose to the datapath for DIE-IRB.

3.2. DIE-IRB Pipeline

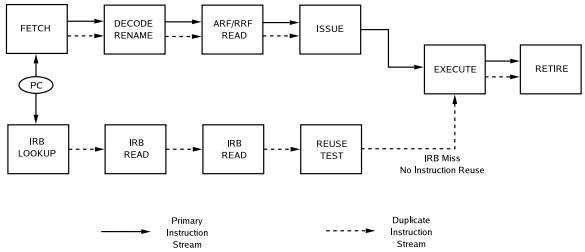
The design of the proposed hardware mandates a close scrutiny of the datapath pipeline, in order to make sure that the cost of the IRB access can be accounted for, while still being able to ensure the benefits of an IRB. At the same time, it is important to allow more than one instruction to exploit the IRB in any one cycle, which is really needed for easing the ALU bandwidth requirements. In order to ensure that the IRB does not become a bottleneck in the system, we allocated the number of ports carefully. We used 4 read-ports, 2 write-ports, and 2 read-write ports. Using Cacti 3.2 [9], we found that for a 180nm process technology, a 1024 entry direct-mapped IRB (which is the eventual configuration that we suggest, and whose hit rate has been shown to be fairly good [29, 35]) can be clocked at our simulated processor clock frequency of 2 GHz using a 3-stage pipelined access as shown in Figure 3. The IRB is looked up in parallel with instruction-fetch using the PC as an index and there are three subsequent possibilities. In the first case, there is a PC-miss in the IRB, in which case the execution proceeds as in a normal DIE (without IRB) for the duplicate instruction. In the second and third cases, the PC hits in IRB, and it takes two more cycles to read the operands/result from the IRB entry, and one more to perform the reuse test, which, as we shall show, can be effectively overlapped with instruction-wakeup.



(a) PC-miss Pipeline



(b) Pipeline with successful instruction-reuse (IRB-hit)



(c) Pipeline with no instruction-reuse (IRB-miss)

Figure 3. Instruction execution pipeline under different instruction-reuse scenarios. The solid-lines show the flow of execution of instructions in the primary-stream and the dashed-lines for those in the duplicate-stream.

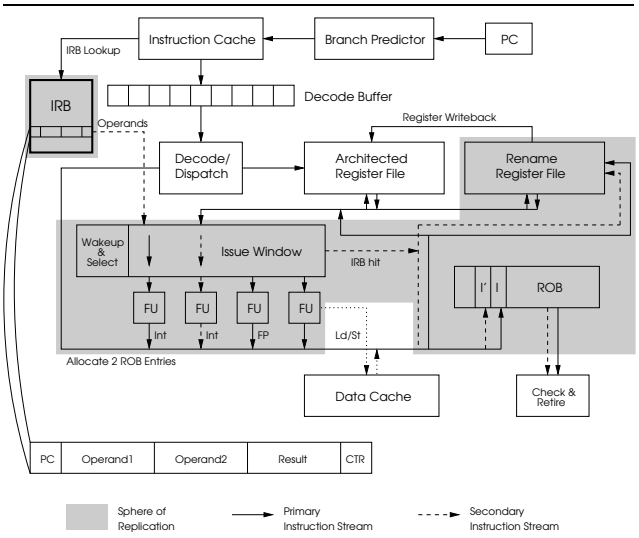


Figure 4. Proposed Datapath Enhancements for DIE-IRB. Note that IRB lookup is done concurrently with Fetch/Decode/Dispatch and there is no dotted line from the result forwarding path to the issue window.

Meanwhile, this duplicate instruction has made its way through the normal pipeline (decode/dispatch) and is at the issue stage. If the reuse test succeeds, then the duplicate instruction can bypass the execute phase, and directly move to the retire/commit stage. Else, from the issue stage, the duplicate instruction goes to the functional units as in the original DIE. If the IRB needs to be updated, it is done on completion of instruction commit/retire, and is not in the critical path.

In our implementation, we employ the IRB for integer and floating-point ALU instructions, and also for branch target-address calculation. For load/store instructions, the IRB is used only for address calculation. This simplifies our instruction reuse scheme, as we do not have to perform a potentially timing-critical memory disambiguation step to check for intervening stores for a load-instruction (which would normally require a full scan of the IRB). The modification to the DIE pipeline by the inclusion of the IRB, along with the format of an IRB-entry is shown in Figure 4.

3.3. Complexity-Effective DIE-IRB Design

In previous proposals for IR, the goal has been to overcome the dataflow limit by collapsing true dependencies. In this respect, the IRB has been treated as a functional unit. As pointed out in [12], this would be effective only if IR was targeted at long latency operations and not for single-cycle ALU operations. Moreover, if the IRB is to behave as a functional unit and provide multiple reads and updates in a cycle, it has to be a multi-ported unit. This implies that each read-port of the IRB needs to have a result-forwarding bus to broadcast the value (on a hit) to the dependent instructions in the issue window. This overhead is similar to the overhead of increasing the issue width, due to the extra match-lines and comparators in each cell of the issue window to check for input operand availability. This increase has a *quadratic* effect on the delay of the wakeup logic and the data-bypass logic [2], both of which are on the critical timing path of modern superscalar processors [22].

As noted earlier, in our design, the IRB is not treated as a functional unit/ALU as far as the issue logic is concerned. Further, we can avoid the inclusion of the extra logic and delays of data forwarding from the IRB by observing an interesting property of Dual Instruction Execution. Note that in parallel with the IRB access, the duplicate the duplicate instruction has made its way through the normal pipeline into the issue window, i.e., both primary and duplicate instructions have entries in the issue window (refer to Figure 3). Since we anyway have the output values (register updates) from the execution of the primary stream's instructions on the functional units forwarded to the issue window to wake up waiting primary stream instructions, why not use the same forwarding of information to wake up waiting duplicate stream instructions as well, i.e. *the results from the primary stream can be used to wake up waiting instructions from both primary and duplicate streams in the issue window.* Whenever a duplicate instruction gets its operands (from a primary instruction output), it performs the reuse test as mentioned earlier. If the reuse test passes, it picks up the result, and directly proceeds to the commit point without propagating any results to the issue window. Consequently, we do not need any data forwarding to occur from the IRB, thereby benefiting from the additional ALU bandwidth provided by the IRB, without complicating/extending the wakeup and bypass logic. It is to be noted that we can achieve this complexity-effective design without really compromising on reliability, as we will show shortly.

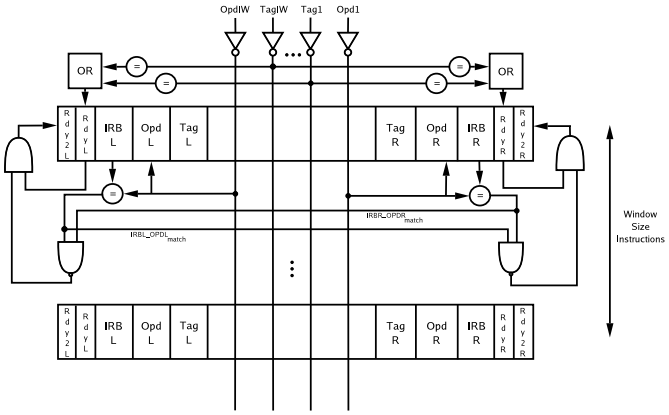


Figure 5. Design of issue window wakeup logic to support input-operand forwarding for DIE-IRB. Note that *IRB L/R* come into issue logic together with the instruction.

The design of the issue logic to support input operand forwarding for DIE-IRB is shown in Figure 5. We use a data capture instruction scheduler [27], where the tag lines (identifying the operand) and operand line (containing the propagated values) are broadcast down the issue window. Normally, each entry of the window has the operand values, and flags (*RdyL* and *RdyR*) to indicate that the operands are available (i.e. the instruction is ready to issue if both these flags are asserted). Our enhancement to the issue window does not require any more entries beyond what is already available in the original SIE/DIE designs. Instead, each entry in the

issue window is augmented with an extra field for each operand to hold the value that has been read from the IRB, designated as *IRBL* and *IRBR* for the left and right operands respectively. In addition, we use 2 additional flags, *Rdy2L* and *Rdy2R*, to indicate whether the operands are available and the instruction is ready to be scheduled. These flags are asserted when (i) the operands are available, and (ii) at least one of the operands do not match the corresponding *IRBL* or *IRBR* values retrieved from the IRB (i.e. this instruction needs to be serviced by the ALUs). Note that the latter condition is the IRB reuse test, and we need two extra comparators for each issue window slot, and the comparison can be performed in parallel with the updating of the values into the *Opd* fields. These flags are then used to determine whether this instruction can be scheduled (instead of the original *RdyR*/*RdyL* flags), and their assertion logic is given below:

$$Rdy2L = (\overline{IRBL\_OPDL_{match}} \vee \overline{IRBR\_OPDR_{match}}) \wedge RdyL$$

$$Rdy2R = (\overline{IRBL\_OPDL_{match}} \vee \overline{IRBR\_OPDR_{match}}) \wedge RdyR$$

When a primary instruction comes to the issue window, two entries are created (one for primary and another for duplicate). In the case of the duplicate, the *IRBL* and *IRBR* entries are filled with values from the IRB, whose lookup is complete by this time. In the case of the primary, one can think of these as having values which are never going to match the operand values propagated to this issue window entry (i.e. it will always execute on the ALUs). Though not explicitly shown in the figure, we point out that the result value from the IRB can directly propagate to the ROB/rename register file by the time the instruction moves to the issue window (even before the reuse test is performed). If the operands match for the duplicate, then it can directly move to the commit stage without using the ALUs. As seen from the figure, our enhancements impose little additional complexity to the issue logic.

Implementing Forwarding in Non Data-Capture Schedulers: In a scheduler that does not employ data-capture, the result tags are broadcast down the issue window and the operands are obtained from the register file. Further, the register file is read *after* the selection process. Therefore, in order to fetch the operands, the instruction scheduler would have to allocate a functional unit to instructions in the duplicate stream as well. Though this allows for a duplicate stream instruction to execute in a functional unit immediately if there is a miss in the IRB, in case of an IRB hit, the allocated functional unit is not used. Further, it cannot be re-allocated to any other ready instruction in the issue window, as any such instruction would first have to perform register file access.

A possible technique to overcome this limitation is to perform the selection *after* the register file lookup, effectively decoupling the wakeup and selection steps. Previous studies have shown that the wakeup and selection steps can be pipelined with very little impact on the overall IPC [6, 32]. In [6], the authors show that pipelining can be achieved by dividing the issue mechanism into a single-cycle wakeup-step and a multi-cycle selection-step. A similar approach can be taken in our case, where, both primary and duplicate instructions can be woken up and made to access the register file. The reuse-test is performed immediately following the register file access for the instructions in the duplicate stream, which may be performed within the same cycle as the register file access. On completion of this step, the ready instructions are assigned the ALUs. Any instruction that did not get an ALU is re-scheduled using a method similar to that described in [6].

While we have use a value-based IRB mechanism for the evaluations in this paper, one could also opt for *name-based* instruction reuse, where the register identifiers (names) are stored in the IRB instead of operand values. Although the hit rates may decrease, name-based approach is more straightforward to implement on a non data-capture scheduler.

3.4. Redundancy Characteristics of DIE-IRB

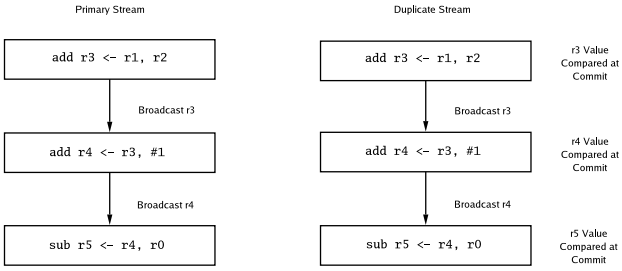
Having covered the design aspects of our proposal, we now discuss its temporal redundancy (reliability) properties. There are two scenarios to consider: (i) the duplicate instruction misses (either PC miss or reuse miss) in the IRB, and (ii) the duplicate instruction hits in IRB. For each of these scenarios, there are two further input operand possibilities: (a) the operands are not provided by any prior instruction (i.e. they are not waiting for any values to be propagated), and (b) they (both primary and duplicate instructions) get one or both operands from another primary instruction.

For the combination of (i) and (a), the primary and duplicate streams will execute on functional units, and our DIE-IRB system is not any different than the original DIE, thereby providing the same redundancy properties. In the case of (ii) and (a), even though the duplicate stream went through the IRB, its result is still compared with that for the primary stream which executed through the functional units. Further, note that in order to create the entry in the IRB, an earlier execution of the same instruction (both primary and duplicate) should have gone through the functional units (and their outputs would have been compared). Only the errors that struck the IRB (after the entry was inserted) to produce an output value which exactly matches the output of a primary instruction that experienced another error during the execution on a functional unit, may not be caught (the resulting probability is comparable to that for the original DIE).

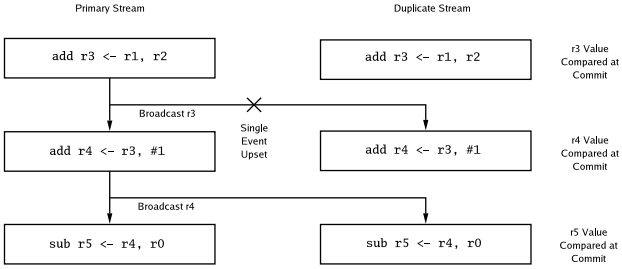
Moving to possibility (b) where the operand values are being forwarded from a prior primary instruction, a pictorial depiction of the situation is given in Figure 6. The original DIE [24] has forwarding only between instructions of its own stream (Figure 6 (a)). However, in our enhancement, forwarding is done only by the primary instructions, which are sent to waiting instructions from both streams. If there is no error occurring in the forwarding itself, then the reliability characteristics are no different from the (a) possibility analyzed above. However, if an error does occur in this forwarding mechanism, it can happen in two ways that are logically shown in Figures 6 (b) and (c). If the error propagates to only one of the streams (either primary or dual, but not both as in Figure 6 (b)), note that the outputs from both instructions (i.e. instruction `add r4 <- r3,#1` in the example) are anyway checked at commit point to ensure they produce the same result.

On the other hand, the situation that we want to address is shown in Figure 6 (c) where the error propagates to both streams. This could happen in a data-capture instruction scheduler, since the result forwarding from the functional units to the issue window and that to the rename register file use separate paths [27]. Therefore an error on the forwarding path to the issue window could go undetected. In order to provide protection under this condition, we propose a minor enhancement to provide additional (spatial) redundancy for these paths (see Figure 7). For instance, by having a duplicate path (which need not traverse through the issue window) the broadcasted tags and data can be compared with that on the original path after the last entry has received it. This adds an extra comparison at the end of the broadcast but the overhead of this operation is relatively small.

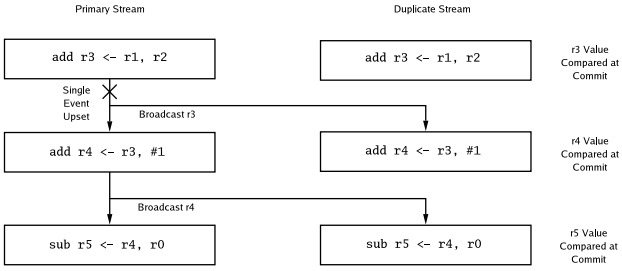
To recover from any detected errors, we do not need any special hardware or pay additional performance costs when incorporating the IRB into the previous DIE design [24, 33].



(a) Value forwarding within a stream



(b) Value forwarding only from primary with errors propagating to one

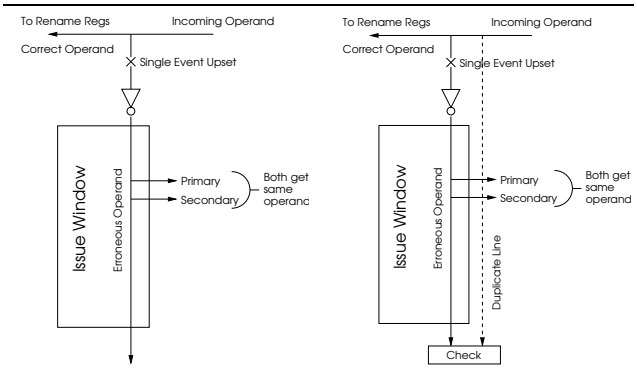


(c) Value forwarding only from primary with errors propagating to both

Figure 6. Value forwarding within/across instruction streams.

3.5. Minimizing Conflict Misses in the IRB

We use saturating counters, similar to those used in prediction structures like the Branch History Table [34], to reduce the conflict misses in the IRB. In our default configuration we use a 4-bit saturating counter, which is initially set at a value of 7 (a more detailed sensitivity study can be found in [23]). If any instruction accesses the IRB and finds that the entry to which it maps to is currently occupied by another instruction, then the value of the counter is decremented by one. If it, however, finds that the entry is occupied by the same PC but there is an operand mismatch, the value of the counter is decremented by 5 (this instruction is less likely to have the same operands in the future, and should be a less favorable IRB entry). On the other hand, if there is an IRB-hit, the value of the counter is incremented by 5. When the counter-value reaches zero, the entry is a candidate for replacement. In essence, the rate at which the counter degrades to 0 is an indication of the usefulness of that entry.



(a) Possible fault-vulnerability (b) Solution

Figure 7. Handling faults on the result-forwarding bus using duplicate forwarding paths.

4. Experimental Results

4.1. Simulation Platform and Workloads

Our experiments were carried out using SimpleScalar 3.0 [7]. In addition to using its models as is for the SIE execution, we extended it (as per the details given in [24]) for DIE, and also simulated the IRB extensions explained earlier for the DIE-IRB scheme. Unless explicitly stated/varied, the default simulation parameters that we use are given in Table 1. For the workloads, we used 12 benchmarks from the SPEC2000 suite. The benchmarks were compiled for the PISA instruction set architecture with the `-O2 -funroll-loops` optimization flags. We used the reference input set for the simulations. Each benchmark was fast-forwarded by 1 billion instructions and then detailed simulation of the next 1 billion instructions was performed.

Please note that our focus in the subsequent evaluations is to reduce the performance gap between SIE and DIE, and we are not studying the performance under error occurrences (which though important, is not the frequent case). Consequently, the metric that we focus on is the percentage reduction in the gap between these two with the different enhancements. The IPCs themselves in the base SIE, and that for the DIE execution, together with the percentage degradation of the latter (referred to as the *gap* henceforth), are given in Table 2.

4.2. Benefits of DIE-IRB

Figure 8 presents the percentage of IPC gap between SIE and DIE recovered with our scheme, denoted as *DIE-IRB-1K-sat*, which uses the default 1K entry IRB parameters given in Table 2 with the saturating counters. In addition to our DIE-IRB scheme, we also present the benefits obtained by doubling the ALUs (the *DIE-2xALU* scheme). The third bar gives the percentage of IPC gap recovered using our scheme where we artificially

Processor Parameters

Fetch/Decode/Issue/Commit Width	8
Fetch-Queue Size	8
Branch-Predictor Type	Combined predictor with 16K meta-table, 16K L1 and L2 tables 11-bit history-width XORed with address in L2 predictor
RAS Size	64
BTB Size	2K-entry 2-way
Branch-Misprediction Latency	7 cycles
RUU Size	128
LSQ Size	64
Integer ALUs	4 (1-cycle latency)
Integer Multipliers/Dividers	2 (3,20)
FP ALUs	2 (2)
FP Mult./Div./Sqrt.	1 (4,12,24)
L1 Cache Ports	2
L1 D-Cache	32KB, 2-way with 32B line-size (2)
L1 I-Cache	64KB, 2-way with 32B line-size (2)
L2 Unifi ed Cache	512 KB, 4-way with 64B line-size (12)
TLB Miss-Latency	30 cycles
Memory Latency	112 cycles
Processor Clock-Frequency	2 GHz
Process Technology	180nm

IRB Parameters

Number of Entries	1024
Associativity	Direct-Mapped
Access-Latency	3-Cycles (Pipelined)
Number of Saturation-Counter Bits	4
Decrement-Value on PC-Miss	1
Decrement-Value on Reuse-Miss	5
Increment-Value on Reuse-Hit	5

Table 1. Default simulation parameters. Latencies of ALUs/caches are given in parentheses. All ALU operations are pipelined except division and square-root.

set the hit-rate to be 100% in the IRB (i.e. all duplicate instructions go through it), denoted as *DIE-IRB-ideal*. The second and third bars thus give an approximate estimate of what we can hope to gain by removing the ALU bottleneck of DIE, and how best we can do with our IRB scheme, respectively. In Table 3, we give the characteristics of the IRB behavior in the *DIE-IRB-1K-sat* execution. The number of references by the duplicate stream to the IRB is broken down into those that do not have a PC match, those that have a PC match but fail the reuse test, and those that have a IRB hit.

We find that the DIE-IRB scheme can recover between 5% (for *art*) to 42% (for *gzip*) of the IPC that is lost due to redundant execution, with the average being around 23%. Several factors contribute to the effectiveness of the IRB in bridging the performance gap between DIE and SIE: (i) the demand for the ALU resources itself, (ii) the reuse hit rate in the IRB, and (iii) the occurrence of long latency ALU operations, and how they can be satisfied by the IRB. The relative impact of these factors on the applications is noted below:

Benchmark	IPC_{SIE}	IPC_{DIE}	$\frac{IPC_{SIE}-IPC_{DIE}}{IPC_{SIE}}$(%)
164.gzip	1.5069	1.2129	19.51
171.swim	1.9807	1.7622	11.03
175.vpr	1.2098	0.9981	17.5
176.gcc	1.2764	1.0410	18.44
177.mesa	2.2230	1.5166	31.78
179.art	0.7316	0.4113	43.78
181.mcf	0.3216	0.2964	7.84
183.equake	1.7404	1.2541	27.94
188.ammp	0.1082	0.1066	1.66
197.parser	1.2685	1.0307	18.75
255.vortex	2.3868	1.6190	32.17
256.bzip2	1.8070	1.2941	28.38

Table 2. IPC under SIE and DIE, and their gap, for simulated SPEC2000 benchmarks.

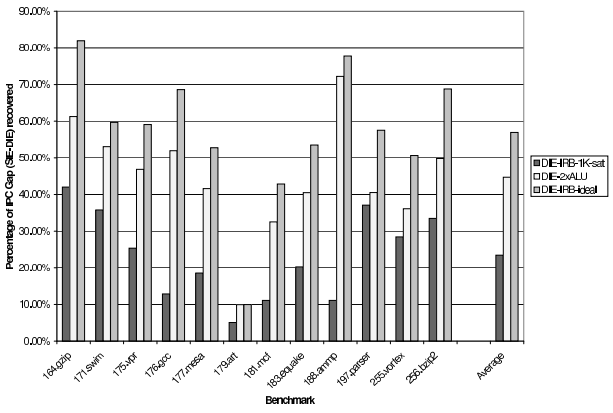


Figure 8. Percentage of IPC Gap Recovered from DIE using Instruction-Reuse. The y-axis plots the value of $\frac{IPC_{scheme}-IPC_{DIE}}{IPC_{SIE}-IPC_{DIE}}$.

- In applications such as *gzip*, *swim*, *parser*, *bzip2* and *vortex*, we are not only getting more than 25% reduction in the IPC gap, but we are getting much of the IPC gains of doubling the number of ALUs. These are applications with high ALU demands (as evidenced by the boost in performance by doubling the ALUs - the bar for *DIE-2xALU*), which at the same time exhibit good instruction reuse, as seen by the IRB hit rate numbers given in Table 3 (close to 40% or higher).
- On the other hand, in *gcc* where even though the ALUs are in demand, the IRB hit rate is less than 25%, thereby leaving a considerable gap between *DIE-2xALU* and *DIE-IRB-1K-sat*.
- There are a few applications where the IRB hit rates in Table 3 seem reasonable, but they still do not provide as much IPC improvements since the ALU bandwidth may not be a significant bottleneck. For instance, in *art* (and perhaps in *mcf* to a certain extent), ALUs are not that much of a problem (note that doubling the ALUs only provides 10% IPC improvement in art). As noted earlier in section 2, this is due to the low ILP in this application, and there are not enough ready instructions to exploit the available bandwidth, thereby limiting the benefits of an IRB. In the case of *ammp*, the gap between SIE and DIE is itself not very significant (only 1.6%),

and the absolute difference between an IRB based approach and doubling the ALUs is quite small.

- In a few other applications, despite the ALU limitations and the reasonably good IRB hit rates, there is still a gap between what doubling the ALUs can achieve, and what is provided by the IRB. This is a consequence of how the long latency ALU operations are executed. Let us consider the case of *mesa*, where despite the 44% IRB hit rate, the benefit for *DIE-2xALU* is more than twice that of *DIE-IRB-1K-sat*. This is because, of the long latency ALU operations, only 5.6% go through the IRB (compare this with *swim* in which we found around 18% long latency operations serviced by the IRB, even though its overall IRB hit rate is even lower than *mesa*). Collapsing of long-latency operations is also one of the reasons why *DIE-IRB-ideal* does even better than *DIE-2xALU* (another reason is the fact that *DIE-IRB-ideal* does not contend for issue ports, unlike *DIE-2xALU*). A consequence of this could be the earlier resolution of branches.

Overall, we find that the IRB does a fairly good job of bridging the gap between SIE and DIE, while not incurring the hardware complexity of design involved in doubling the number of ALUs. On the average, we obtain 23% reduction in IPC gap between SIE and DIE, which is over 50% of that provided by doubling the number of ALUs. At the same time, the performance of *DIE-IRB-ideal* indicates that there is more room to grow for this scheme, if we can somehow enhance its hit rate (as is investigated in the next subsection).

Benchmark	PC miss	IRB miss	IRB hit
164.gzip	8.46	49.44	42.09
171.swim	3.29	56.71	39.98
175.vpr	38.70	23.75	37.54
176.gcc	28.70	46.33	24.96
177.mesa	39.62	16.18	44.19
179.art	0.01	59.19	40.79
181.mcf	0.02	64.37	35.60
183.equake	5.34	60.69	33.96
188.ammp	8.57	54.03	37.39
197.parser	9.95	36.41	53.63
255.vortex	37.55	23.59	38.84
256.bzip2	0.09	56.42	43.48

Table 3. Classification of IRB-Accesses

4.3. Enhancing IRB Reuse Characteristics

Techniques for enhancing the hit behavior of the IRB could prove to be useful in alleviating ALU bandwidth even further (note that *DIE-IRB-ideal* shows we have much to gain by improving IRB hit behavior). There can be three different ways of achieving this: (i) We can enhance the IRB configuration to hold more (possibly useful) entries to reduce capacity/conflict misses (i.e. PC-misses), (ii) Even if we have to be confined to a small IRB (to meet timing requirements) we can try to make sure only useful entries are maintained in the limited space of the IRB so that non-useful entries do not evict useful ones (reducing both PC-misses and reuse misses), and (iii) We can also proactively insert entries into the IRB to anticipate what would be needed (reducing reuse misses). We have investigated each of these issues, and detailed results can be found in [23].

In general, we find that the 1K direct-mapped IRB with saturating counters, used until now, does a fairly good job in terms of issues (i) and (ii) mentioned above. We find that predicting what operands to expect for an instruction, and pre-computing the results for those instructions ahead of need, can on the average reduce the IPC gap between SIE and DIE-IRB by 35.95% (for a six-application subset that we used for sensitivity analysis), compared to 26.03% without any proactive insertion. It should be noted that the additional hardware for performing such computations are not part of the issue/execute logic, and do not complicate their design. At the same time, our point is to merely note the benefits of predictive IRB insertion, which we hope to investigate in detail in future work.

5. Concluding Remarks

This paper has proposed a novel solution to alleviate the performance loss incurred when implementing instruction-level temporal redundancy in an out-of-order superscalar processor. As in earlier studies, we observed between 1% to 44% loss in IPC, in a temporally redundant (dual) execution (called DIE) over a single instruction execution (SIE) for a set of 12 SPEC2000 benchmarks. An important contributor to this performance gap is the insufficient ALU bandwidth, since the same number of ALUs provisioned for an SIE core needs to handle twice the number of instructions imposed on a DIE. Increasing the number of ALUs may not be an easy solution to this problem since it increases the complexity of the issue and bypass logic.

We propose a novel adaptation of an existing idea, Instruction Reuse (previously examined to improve single stream execution), to alleviate the ALU bandwidth problem on a DIE. The primary instructions always use the ALUs. However, the duplicate instructions are directed to an instruction reuse buffer (IRB) from where they can directly pick up the result (instead of execution on ALUs) as long as that instruction has been previously encountered with the same set of operands. Else, the duplicate instructions are directed to the ALUs as in a normal DIE. This mechanism is able to provide the required temporal redundancy semantics without requiring any extra protection from faults for the IRB. The temporal redundancy is being provided by instruction reuse rather than re-execution.

In our proposal, the issue logic does not manage the IRB (its lookup is done ahead in the pipeline) and thus does not increase the scheduling complexity. At the same time, we exploit a unique feature of DIE, wherein the primary stream can forward data values to waiting instructions for both primary and secondary streams, thereby not requiring us to complicate the wakeup and bypass logic.

Through detailed simulations, we show that our proposal can provide between 5% to 40% reduction in the IPC gap between SIE and DIE, with around 23% reduction on the average over 12 SPEC2000 benchmarks. This amounts to around 50% of what is achievable by removing the ALU bandwidth overload imposed by DIE. These savings are achievable with a 1024 entry direct mapped IRB, with read/write ports that can be easily provisioned in a 3-stage pipelined access manner for a 2 GHz processor.

There are several interesting directions for further research. As mentioned, the *DIE-IRB-ideal* scheme (which assumes 100% IRB hits) provides substantially more savings than what we got with our practical design. It would be interesting to examine possibilities to bridge this gap not just by keeping only useful entries, but also by proactively inserting entries anticipating their need. A preliminary investigation along such lines can be found in [23].

We pointed out that one could employ a clustered architecture for instruction-level temporal redundancy, and use separate clusters for each stream (to alleviate load imbalance and inter-cluster communication). However, this still requires replicating the issue window and the register file (spatial redundancy), going back to the question of whether we could use those resources for enhancing SIE itself. Investigating these trade-offs, together with more hybrid (a combination of temporal and spatial redundancy) approaches, is part of our future work.

Acknowledgments: This research has been funded in part by NSF grants 0103583 and 0130143, and an IBM SUR equipment grant. We would like to thank the anonymous referees for detailed comments which helped us improve the quality of the presentation.

References

[1] A. Aggarwal and M. Franklin. Instruction Replication: Reducing Delays Due to Inter-PE Communication Latency. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 46–55, September 2003.

[2] P. Ahuja, D. Clark, and A. Rogers. The Performance Impact of Incomplete Bypassing in Processor Pipelines. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, November 1995.

[3] A. Aletà, J. Codina, A. González, and D. Kaeli. Instruction Replication for Clustered Microarchitectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2003.

[4] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 196–207, November 1999.

[5] A. Baniasadi and A. Moshovos. Instruction Distribution Heuristics for Quad-Cluster Dynamically-Scheduled, Superscalar Processors. In *Proceedings of the International Symposium on Computer Microarchitecture (MICRO)*, pages 337–347, December 2000.

[6] M. Brown, J. Stark, and Y. Patt. Select-Free Instruction Scheduling Logic. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 204–213, December 2001.

[7] D. Burger and T. Austin. The SimpleScalar Toolset, Version 3.0. http://www.simplescalar.com.

[8] C. Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4):14–19, July-August 2003.

[9] Cacti 3.2. http://research.compaq.com/wrl/people/jouppi/CACTI.html.

[10] R. Canal, J. Parcerisa, and A. González. Dynamic Cluster Assignment Mechanisms. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 132–142, January 2002.

[11] D. Citron and D. Feitelson. The Organization of Lookup Tables in Instruction Memoization. Technical Report 2000-4, Hebrew University of Jerusalem, March 2000.

[12] D. Citron and D. Feitelson. Revisiting Instruction Level Reuse. In *Proceedings of the Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, May 2002.

[13] D. Citron and D. Feitelson. "Look It Up" or "Do The Math": An Energy, Area, and Timing Analysis of Instruction Reuse and Memoization. In *Proceedings of the Workshop on Power-Aware Computer Systems*, December 2003.

[14] T. Ehrhart and S. Patel. Reducing the Scheduling Critical Cycle using Wakeup Prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, February 2004.

[15] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 98–109, June 2003.

[16] HP NonStop Himalaya. http://nonstop.compaq.com/.

[17] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar. The Optimal Logic Depth Per Pipeline State is 6 to 8 FO4 Inverter Delays. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 14–24, June 2002.

[18] S.-J. Lee and P.-C. Yew. On Some Implementation Issues for Value Prediction on Wide-Issue ILP Processors. In *Proceedings of the International Conference on Parallel Architectures and Compliation Techniques (PACT)*, pages 145–156, October 2000.

[19] M. Lipasti and J. Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 226–237, December 1996.

[20] A. Mendelson and N. Suri. Designing high-performance and reliable superscalar architectures-the out of order reliable superscalar (O3RS) approach. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 473–481, June 2000.

[21] S. Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, University of Wisconsin - Madison, 1998.

[22] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 206–218, June 1997.

[23] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy. Technical Report CSE-04-008, The Pennsylvania State University, March 2004.

[24] J. Ray, J. Hoe, and B. Falsafi . Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 214–224, December 2001.

[25] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 25–36, June 2000.

[26] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*, pages 84–91, June 1999.

[27] J. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors (Beta Edition)*. McGraw Hill, 2003.

[28] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on Soft Error Rate of Combinational Logic. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2002.

[29] A. Sodani and G. Sohi. Dynamic Instruction Reuse. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 194–205, June 1997.

[30] A. Sodani and G. Sohi. Understanding the Differences Between Value Prediction and Instruction Reuse. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 205–215, December 1998.

[31] G. Sohi, M. Franklin, and K. Saluja. A Study of Time-Redundant Fault Tolerant Techniques in High Performance Pipelined Computers. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*, pages 436–443, June 1989.

[32] J. Stark, M. Brown, and Y. Patt. On Pipelining Dynamic Instruction Scheduling Logic. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 57–66, December 2000.

[33] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery via Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 87–98, May 2002.

[34] T.-Y. Yeh and Y. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 124–134, May 1992.

[35] J. Yi, R. Sendag, and D. Lilja. Increasing Instruction-Level Parallelism with Instruction Precomputation. In *Proceedings of Euro-Par*, August 2002.

[36] J. Zeigler. Terrestrial Cosmic Rays. *IBM Journal of Research and Development*, 40(1):19–39, January 1996.