RC23113 (98831) March 19, 2001
Electrical Engineering

IBM Research Report

High Frequency Pipeline Architecture Using the Recirculation Buffer

Michael Gschwind, Stephen Kosonocky, Erik Altman
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

High Frequency Pipeline Architecture using the Recirculation Buffer

Michael Gschwind, Stephen Kosonocky, Erik Altman

{*mkg,stevekos,ealtman*}@*us.ibm.com*

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

Traditional processor synchronization methods such as stall and exception handling are quickly becoming bottlenecks for new designs. In particular, both the stall mechanism and the traditional exception handling mechanism are based on the assumption that computation is expensive and communication cheap. As processors move beyond the 1GHz operating frequency, and wire delay dominates transistor switching speeds in new designs, this is no longer the case. As a result, lockstep synchronization mechanisms become increasingly problematic in achieving global synchronization in a processor pipeline.

In this paper, we present a high-frequency design called BOA (Binary-translation and Optimization Architecture). The design is built around two principles. Architecture complexity is handled with aggressive layering based on dynamic compilation techniques. High-frequency is achieved by using a new mechanism for resolving structural and data hazards, and for handling processor exceptions. This approach is based on a streaming model of pipeline execution, where instructions continue to proceed through the pipeline even when a hazard has been detected. Reissue logic then detects such conditions, invalidates instructions which have violated integrity constraints and directs the issue logic to re-issue those instructions.

1 Introduction

High circuit speed is an important enabler for high microprocessor performance. To analyze the impact of high-performance design requirements on microarchitecture and circuit design, several IBM design teams have developed design studies or actual implementations based on different technologies.

These high-frequency design vehicles have included the world's first Gigahertz processor, guTS [1], a fully functional IBM 64 bit POWER implementation, Rivina [2], and a POWER design study using dynamic compilation and VLIW design techniques, BOA [3].

While all these projects had a shared vision of achieving ultra-high performance, they varied widely in approach. guTS and Rivina were test sites based on the use of novel circuit design techniques, and the test site demonstrated the feasibility of the circuit design style. BOA was a design study which focused on using a new microarchitecture coupled with dynamic compilation techniques, which required careful performance modeling of the microarchitecture and software techniques.

Unlike commercial implementations which emphasize system value to customers, these design studies were chartered to explore the most aggressive techniques available. This will allow us to understand the challenges ahead for future systems, and feed back into future design decisions.

Decoupling the public architecture from the internal structure allows many design strategies to minimize complexity and critical paths. Several designs have utilized decoupled architecture/implementation using sophisticated circuitry. Typically, this is achieved by layering implemented as "cracking," wherein **Decode** stages in the pipeline split more complex instructions into a sequence of simple micro-operations.

In designs for complex architectures such as IBM System/390 (and Intel's x86), this led to a partitioning of the architecture into "outer" and "inner" architectures [4]. In these designs the outer architecture is responsible for the cracking step and runs at a slower frequency. The inner architecture implements a pipeline executing these simpler micro-operations and can operate at a higher frequency. This strategy has also been used for implementing IBM PowerPC, but since the PowerPC architecture is a simpler RISC architecture, decode and cracking can work at full speed in POWER4 [5, 6].

To reduce implementation complexity, BOA opted for a simpler approach where the layering is performed using software-based binary translation [3] based on the principles of the DAISY design approach (DAISY stands for **D**ynamically

Architected **I**nstruction **S**et from **Y**orktown). Based on this approach, we set out to design a simple high-performance microarchitecture with an optimized internal instruction set architecture to host a dynamic compilation system. The dynamic compilation system then provides compatibility with the PowerPC system architecture by implementing a virtual PowerPC architecture and incrementally translating (and optimizing) PowerPC code fragments to the host architecture [7, 8, 9, 10, 11].

The dynamic compilation and layering approach employed in BOA is described elsewhere [12, 3, 13]. In this paper we will focus on some microarchitecture techniques which were used to implement the underlying high-performance architecture. In particular, these decisions were driven to minimize the impact of wire delay on the operating speed of circuits, since microprocessor design today is characterized by an increasing importance of wire delay in the overall design assumptions of processors.

This paper is organized as follows: in Section 2, we describe the design challenges facing microprocessor designers beyond the Gigahertz frontier. In Section 3, we introduce our pipeline design strategy based on instruction streaming. Section 4 introduces the BOA microarchitecture designed around these techniques. In Section 5 we present our implementation of streaming pipelines based on the recirculation buffer. In Section 6, we discuss related work and Section 7 presents performance results and draws conclusions.

2 Design Challenges in Multi-GHz Designs

Historically, processors have been designed based on the notion that communication was cheap, and that logic delay was the critical aspect of processor designs. As a result, processor control logic was based on the immediate communication of any exceptional events, so that immediate recovery could be started. This maximizes the throughput achieved with the logic elements.

In processor pipelines, hazard conditions need to be detected and resolved for modern processors to work correctly. Hazard conditions include structural hazards where multiple instructions compete for the same function units, and data hazards where consumer instructions can only access a result after the producer instruction has computed them.

These conditions are usually resolved by introducing a processor "stall". During a "stall", the processor's control logic suspends the processing of instructions which wait for a function unit or an input value to become available. Processing

of the stalled instructions resumes when the hazard has been resolved and the stall condition has been cleared.

The mechanics of stall processing require that stall information be distributed to other pipeline stages which may need to synchronize their stalling with the original instruction. Specifically, when an instruction stalls, it can cause other instructions to stall due to one of two hazards:

data hazard When an instruction must stall, other instructions expecting to receive their input operands from the stalled instructions may have to stall until the stalled instruction generates its result.

structural hazard When an instruction stalls, it may block a function unit required by another instruction. This would require that instruction to stall until the unit becomes available.

Stalls conditions are transitive, so as instructions are forced to stall, they in turn may cause additional instructions to stall. In some implementations, a stall may force *all upstream instructions* to stall, to enforce correct in-order execution behavior.

This involves wire delays across significant portions of a microprocessor design, and can rapidly become a bottleneck in the processing speed. This problem is exacerbated by the late availability of stall condition information: stall conditions can be fairly complex to evaluate, and are often available only towards the end of a processing cycle when processing detects an abnormal condition.

Thus, there is little or no overlap between logic and wire delay, and the wire delay represents an additive penalty to the cycle time.

Following the description in [1], it already takes 5% of the available cycle time to cross a single pipeline stage in guTS. In a moderately pipelined architecture with 8 pipeline stages, this may constitute 40% of the overall cycle time following the same technology assumptions. This trend is exacerbated by the choice of deeper pipelines (which reduces the amount of logic FO4 per stage and hence increases the relative contribution of wire delay to a given cycle), and future process technologies which suggest increasing wire delay relative to logic speed and density.

We estimate for a 0.1μm technology the chip to be 15.9mm x 17.39mm for four processor cores plus L2 Cache. This includes a 4MB L2 cache with L3 interface and bus logic, and in each core: 256KB L1 instruction cache, 4 fixed point units, 2 floating point units, 2 load/store units and 64KB L1 data cache.

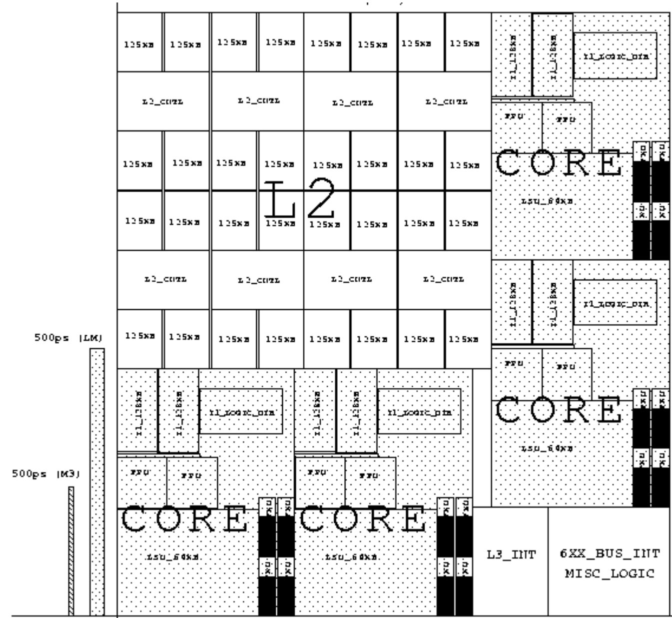


Figure 1: Floorplan for a four core chip-level multiprocessing system based on the BOA architecture. A core consist of four integer pipelines each containing a register file in the lower right corner of the core, two load/store units and the data cache in the lower left corner, two floating point units on the left in the mid-tier, and the instruction cache and instruction decode and dispatch logic in the upper tier.

Figure 1 shows a floor plan at the basis of our architecture study. The figure also shows the estimated wire length for an unrepeated 500ps transition in thick LM and narrow M3 with standard SiO2 oxides and copper wire. It can be seen that it takes approximately 500ps in last metal to traverse the height of the core. Dense wires would only traverse approximately half this distance in the same time.

As a result of the projected wire delays, overall system performance could be reduced by a factor of up to 2 to 3 due to wire delay. While clever floor planning can reduce this penalty to some extent, a significant cycle time penalty remains. In synchronous designs, this budget for communication across the chip has to be allocated in every execution cycle even if no stall condition occurs.

In addition to the wire delay penalties, these designs can also produce quite complex control logic, based on different states that each pipeline stage can be in, and makes the design of processor control logic complex, error-prone and slow.

Complexity also biases the design approach towards synthesis-aided HDL design, which can result in slower circuit speeds.

In this paper, we present an alternative pipeline control design mechanism, based on a streaming model of pipeline execution, where instructions continue to proceed through the pipeline even when a hazard has been detected. Reissue logic then detects such conditions, invalidates instructions which have violated integrity constraints and directs the issue logic to re-issue those instructions. This scheme requires no global signals for synchronous communication. Despite a large chip size, the elimination of wire delay from such global signals allows the design to achieve a very high operating frequency (currently estimated at more than 3GHz).

3 Streaming pipeline control

To eliminate the frequency penalty inherent with the global communication requirements of stall-based pipeline synchronization, we departed from the traditional design techniques employing sophisticated control logic to sequence the flow of instructions. Instead, we defined a new streaming dataflow pipeline management architecture for BOA.

We first summarize the limitations of traditional stall processing. According to this traditional sequencing technique, logic in each pipeline stage determines whether there is a need to raise a stall condition. Typical causes triggering stall conditions are data dependent processing (e.g., for iterative shift, multiply and divide, for large normalization shifts due to massive cancellation in floating point processing, for special handling of denormalized numbers) or a miss in caching structures (data caches, TLBs, segment tables, and so forth). Depending on the specific nature of the stall condition and instruction sequence, this can trigger a data hazard, a structural hazard, or both.

Figure 2 (a) shows the processing of instructions with a traditional pipeline stall design. Each cycle shows time allocated for processing (white) and communication between logic, such as propagating stall signals. For illustrative purposes, these are uniformly fixed fractions across all stages, but these may (and will) actually vary across different stages. The broadcast pipeline stage BC allows for the propagation of wider buses using dense and slow metal layers across a single processor core (which may or may not be necessary for some designs).

Figure 2 (b) shows how immediate propagation of a stall condition (a TLB miss in this illustrative example) can cause other instructions to suspend execution at the start of the next processor cycle until the stall condition is resolved.

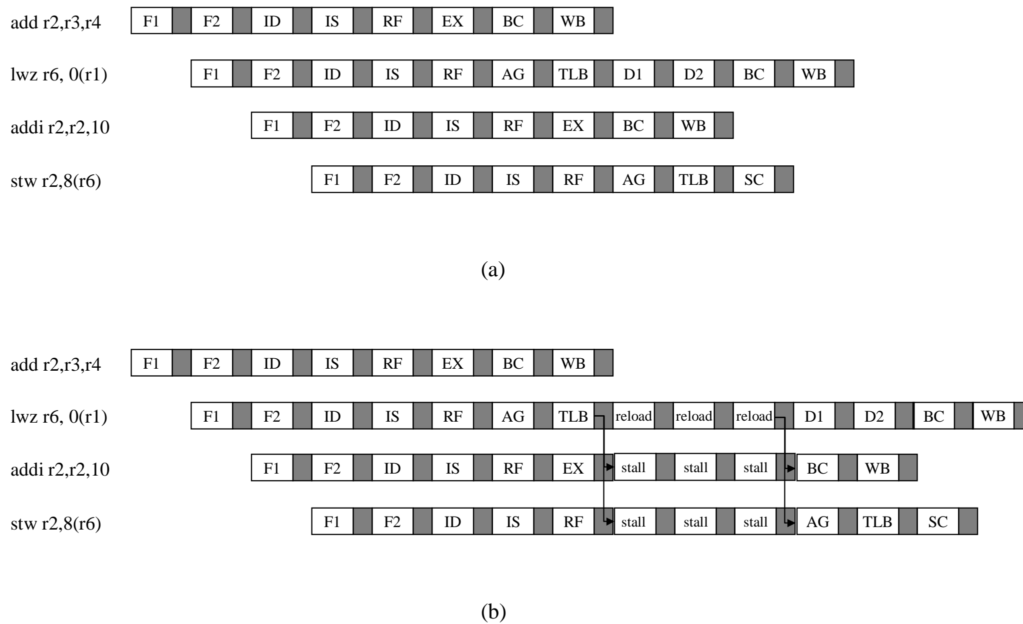


Figure 2: (a) This pipeline diagram includes time budgeting information: for each cycle, a fraction of the overall processing time is allocated for processing (white) and communication for managing stall signals (gray): with increasing wire delays, a significant time budget must be allocated to propagating stall information across a processor core. (b) When a stall occurs, the condition can be signaled to other pipeline stages to prevent processing to occur in the immediately following cycle.

According to the streaming pipeline control we adopted for the BOA microarchitecture, time for communication is *not* allocated in each processing cycle. Instead, we aim to set the processor cycle to match the cycle of a single block of processing logic (such as a 64b adder), and make communication explicit in the microarchitecture. Specifically, this involves providing cycles for result transfers, and dealing with the time needed to propagate an extraordinary condition through the core.

The decision to not allocate time for distributing stall information implies that when a hazard is detected, it cannot be resolved by suspending operation of dependent instructions. By the time control information reaches all units, instructions have latched into the next pipeline stage, and started processing. As a result, instructions may have violated integrity constraints, as they may be unable to bypass

correct inputs when necessary, access units which may still be blocked, etc.

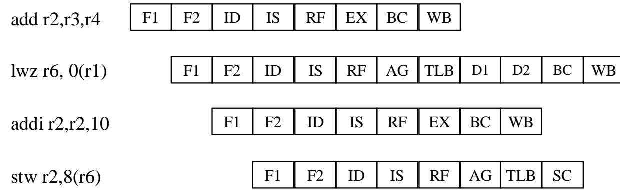
Since it is impossible to reach other units (or even far away stages within the same unit), no attempt is made to stop execution. Instead, each instruction is allowed to proceed in the pipeline and when a hazard is detected, the instruction is marked as carrying an incorrect result. To ensure that the architected state cannot be corrupted, marking an instruction suppresses its ability to commit state changes, and, for an in-order implementation, all subsequent instructions. Blocking all subsequent instructions from changing the state is required because in in-order processors, integrity violation is transitive, i.e., when an instruction has violated its integrity constraints and cannot complete successfully, then all its successor instructions have violated these constraints.

Instructions which have violated their integrity constraints are eventually aborted, and re-issued. This is achieved by downstream logic embedded in each pipeline which tests tests instructions for their execution integrity. If an instruction has executed successfully, it is allowed to complete and update the processor state. If it has violated any integrity constraints, it is aborted, and the **Issue** stage is told to reissue the instruction to the processor pipeline. When the instruction is reissued into the processor pipeline, hazards will most likely have been resolved, and the required resources be available.

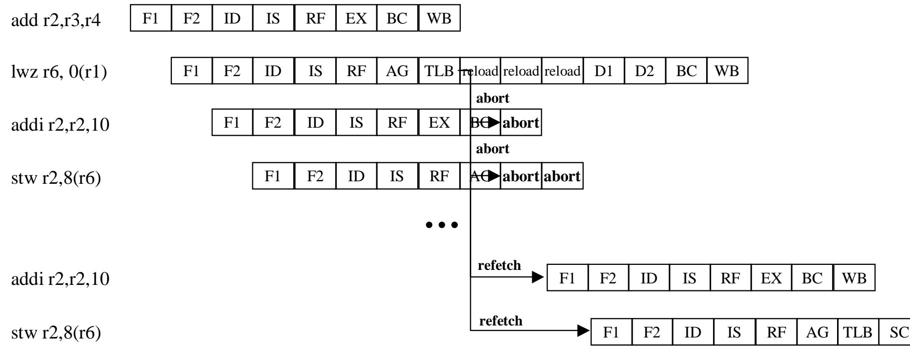
If for any reason, the resources are still not available, the instruction will again be invalidated and re-issued, and so forth. To reduce cost (and power consumption) of recirculating the same instruction multiple times, more sophisticated control may be provided. This may take the form of specifying a data or unit availability time (e.g., in the form of a hold-off cycle count to indicate a minimum bound for resource availability), or by providing a second level of issue logic.

When using recirculation-based pipeline management, it is imperative that the *entire* state of the processor prior to the recirculated instruction be recoverable. This includes not only the register file, but also implicit machine state registers and memory state. In an out-of-order processor, this may be accomplished using register renaming or history files. However, simpler in-order processors may want to forgo such complexity and use a strategy based on matching pipeline depths for different execution pipelines to ensure that *any* state is only changed after potential cancel signals have been distributed. Results can be made available via result bypassing to eliminate any CPI degradation as a result of such a microarchitectural decision.

Figure 3 shows a pipeline diagram employing streaming pipeline control. Explicit cycle time allocated to propagating control signals has been eliminated from each cycle. The cycle allocated for result communication is also used to propa-



(a)



(b)

Figure 3: (a) Pipeline execution of a code fragment. (b) As no time has been allocated for transferring stall conditions, pipeline management is performed using a strategy based on aborting and re-executing dependent instructions. This allows control signals to arrive at any time before state changes are committed to the architected state.

gate control information throughout the entire core leading to the suppression of instructions (by inhibiting writeback to the register file, the store buffer (SC), or any other architected resources), and to trigger re-execution of suppressed instructions.

We conclude our description of the basic pipeline control mechanism with two observations. First and foremost, it should be noted that the information to re-issue an instruction could use multiple processor cycles to be conveyed to the **Issue** stage, and hence does not determine the processor frequency.

Secondly, this approach can be employed to implement other global signals which might otherwise limit the achievable operation frequency. Examples where this approach can be employed include the exception handling, branch mispredic-

tion, and micro-code entry for complex corner cases. In each case, the instruction which causes the control event is returned to the issue queue, together with additional information.

In the case of an instruction which has raised an exception, the exception-raising instruction is returned to the **Issue** stage together with control information describing the nature of the exception. An exception-raising instruction also blocks state changes to the architected state until the first instruction of the exception handler is encountered.[1] The **Issue** stage will then not reissue the original instruction, but start issuing the exception handler in its place. A similar mechanism can also be employed to transfer control to microcode to handle special instruction conditions using microcode.

In the case of a mispredicted branch instruction, a mispredicted branch is recirculated together with the correct branch target address. In addition, the incorrectly executed branch instruction prevents state changes by instructions starting at the mispredicted branch target address until the correctly predicted branch is issued.

Recirculation is necessarily more expensive in terms of CPI impact, since a single stall cycle is replaced by multiple execution cycles which will later be invalidated, followed by one or more cycles of transmission back to the issue queue, and finally the instruction re-execution. Additional penalties may be involved to actually suppress the execution of some instructions. Thus, the cost of encountering a recirculation condition is the sum of the pipeline depth, the cost to suppress instructions, and the recirculation penalty. This compares to degradation corresponding to the latency of actual stall event as seen by the traditionally managed (stall-based) execution sequence.

However, we feel that careful management of the occurrence of recirculation conditions can reduce the overall impact using a variety of factors. First, careful code scheduling can reduce the need to dynamically discover dependencies. This is particularly easy to guarantee in the BOA system where code is generated dynamically at runtime for the specific underlying machine model. Secondly, using issue logic which is careful to issue instructions only when they have a high probability of succeeding, and using hold-off information to reduce spurious recirculation for long latency operations.

A final strategy to reduce the impact of the recirculation cost on performance is to reduce the cycle penalty incurred. To reduce this penalty, we have introduced a *recirculation buffer* which reduces the recirculation penalty significantly over the case when instructions are re-fetched from the instruction cache prior to

[1] Very limited state changes are allowed in this case to reflect the exception information.

recirculation.

Recirculation-based pipeline management offers significant advantages and is an important enabler to ensuring high operating frequency with robust pipeline control, since it is possible to eliminate global stall signals, and hence critical paths from the design. In trading off lower cycle time incurred over all execution cycles, and CPI degradation which can be held low if carefully managed, we have optimized for overall execution time.

4 The BOA microarchitecture

We have used the principles outlined in the previous section in the design of the BOA microarchitecture. The BOA processor architecture is specifically targeted as the execution engine for the BOA binary translation system and contains a variety of optimizations for this purpose. The underlying architecture was designed as a simple in-order variable-width VLIW architecture with an ultra-high frequency design target. The BOA dynamic compiler implements the PowerPC instruction set as a software-based virtual machine layer on this high performance execution engine.

BOA's binary translator can schedule operations to execute in a different order than the original PowerPC code, and BOA's system software can recover the proper in order PowerPC state when needed. Thus, while the underlying microarchitecture implements an in-order model, code can be aggressively rescheduled out-of-order with respect to the original PowerPC code.

The BOA processor architecture contains support for the dynamic compilation system as described in [3], and has the following execution pipelines:

- 1 Branch unit (uses some fixed point resources, and thus its presence limits the number of fixed point unit instructions which can be issued to 3 fixed point instructions)
- 2 Dedicated Load/Store units
- 4 Fixed point units
- 2 Floating point units

As depicted in Figure 4, individual BOA operations are encoded in **bundles** of three. However, these bundles are an encoding abstraction *only*. When the instruction stream is decoded, stop bits are interpreted to identify boundaries of

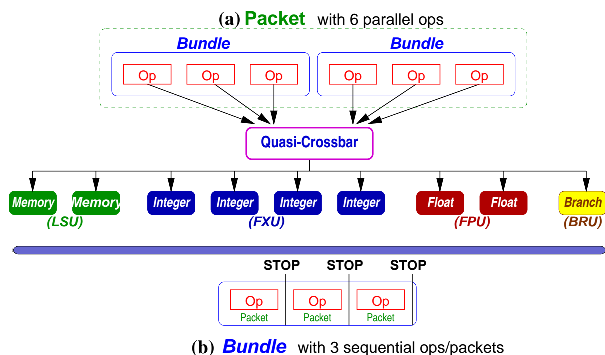


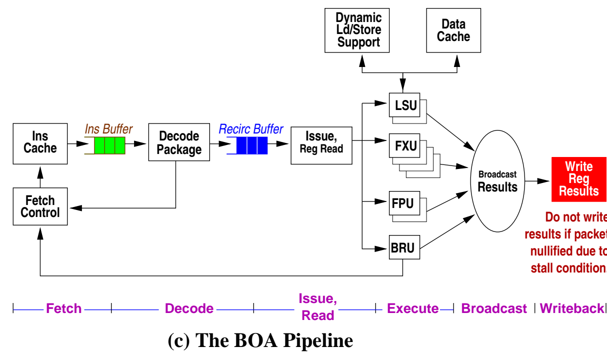
Figure 4: BOA instructions

packets. In BOA, packets represent parallel operations which are issued together and can contain one to six operations for the nine execution pipelines. The semantics of parallel operations in a packet are parallel, i.e., all operations within a packet read the same values even if an operation's input within a packet refers to another operation's output within that same packet. (Unlike in scalable VLIW architectures [14], there is no intergenerational compatibility impact due to this decision, as the code for the architecture is generated by the system-specific and system-resident binary translation and optimization system.)

As shown in Figure 4, a single packet can span multiple bundles (up to 3), or multiple packets can be contained within a bundle. Packet encoding is positional. For example, a branch operation must be the first operation in its packet. Since this positional encoding is internal to the binary-translation system surrounding the core this does not lead to compatibility issues, but does simplify instruction **Decode** and **Issue**. In particular, it reduces the switch which aligns and distributes operations to their respective pipelines.

BOA bundles are 128 bits and as already stated, encode 3 basic BOA operations. Bundles must be aligned on 128-bit boundaries. BOA can issue a packet consisting of up to 6 BOA operations each (in two bundles), as shown in Figure 4(a). To achieve good code density and manage code explosion, packets can straddle bundle boundaries, and a packet can contain instructions from multiple bundles as in Figure 4(a). However, branch target addresses must be aligned at double-bundle boundary to simplify instruction decoding and formatting after a branch.

BOA decouples the decoding of bundles and the issuing of packets through the use of the decoupled fetch/execute architecture depicted in Figure 5. The



(c) The BOA Pipeline

Figure 5: BOA decoupled fetch/execute pipeline.

Fetch and **Decode** portions of the pipeline are responsible for fetching bundles and aligning them into packets under the control of the instruction encoding's **Stop Bits** (See Figure 4(b)) in several pipeline stages. The instruction **Fetch** unit also implements static branch prediction. Branch prediction instructions are encoded as branch unit instructions which are defined to be the first operation within a bundle, reducing the hardware necessary to decode and execute static branch prediction instructions in the **Fetch** unit.

In the BOA architecture, packets are an atomic execution unit, and all operations within a packet are either committed to the architecture state, or all writebacks are suppressed. This decision simplifies control sequencing and allows a wide issue architecture to execute with little control overhead.

Each pipeline contains a separate copy of the appropriate register file to reduce the number of ports required. To ensure consistent architectural state, writes are performed to all register file copies, and a full broadcast cycle is allocated for write results to propagate to all function units. Since one write port is allocated for each execution pipeline, this requires a maximum of 6 write ports, and three read ports (for fused-multiply-add and store X-form instructions). This decision has allowed to reduce the number of required ports, and the wire delay incurred to access the register file during the register file read stage.

To simplify the implementation and hold the number of write ports low, the BOA binary translator and optimizer cracks update memory instructions from the PowerPC instruction set architecture into a memory operation and an add instruction which can be scheduled independently. This allows to hold the number of required ports to one per execution pipeline.

The BOA architecture does not support back-to-back dependent operations,

since forwarding would impose significant cycle time degradation if performed across an entire BOA core encompassing the branch unit, two load-store units and four fixed-point execution units. As a result, the effective latency for simple ALU operations is 2 cycles, for the operation execution (EX) and result broadcast (BC).

The binary translation system is expected to schedule instructions to cover this latency. While no hardware support is provided for two-cycle latency basic operations, a limited form of scoreboarding is provided in the **Issue** stage to implement *stall-on-use* semantics for memory operations and variable latency floating-point operations. Thus, BOA stalls on a cache miss only if a subsequent operation needs the (LOAD) result of a data cache miss.

Unlike traditional scoreboarding architectures, dependencies are only detected for instructions after a first dead cycle to reduce critical paths and hence frequency inhibitors which can be addressed in software. In particular, this reduces the crucial time to read the scoreboard bits, analyze them, and update them. Since the register files are distributed, scoreboards are also distributed, and an update to remote scoreboards is assumed to take a cycle. Thus, dependent operations cannot be in back-to-back packets, reducing the need to deal with wire delay in a moderately aggressive ILP architecture without frequency penalty [15]. The net effect is that predictable latencies are dealt with by software, but unpredictable latencies such as memory access and variable-latency floating point operations are handled by hardware. Using the provided scoreboarding facility, memory operations do not have to incur *stall-on-miss* penalties, which would penalize performance regardless stall the machine even if a cache-miss value being loaded was not used until many cycles in the future (or incur the cost of recirculating dependent operations). Similarly, providing scoreboarding for variable-latency floating point operations allows to schedule for typical execution latency, without the need to schedule the code for worst case latencies – possibly by introducing explicit NOP packets –, or incurring the cost of recirculating dependent operations.

4.1 Exception handling in BOA

To avoid the need to perform state rollback to the previous checkpoint on frequent events such as TLB misses, the BOA architecture offers precise behavior on most memory faults. To this end, the pipelines are architected to perform address generation and TLB access before a packet containing memory operations is committed (see Figure 6. Thus, if such a packet incurs a TLB miss (or page fault), the entire packet can be canceled and TLB reload can be attempted.

After address translation has been performed, a load operation is enqueued

Fixed-Point	Load	Store
Fetch 1	Fetch 1	Fetch 1
Fetch 2	Fetch 2	Fetch 2
Decode	Decode	Decode
Issue	Issue	Issue
GPR Read	GPR Read	GPR Read
Execute	AGEN	AGEN
Broadcast	TLB Access	TLB Access
Writeback	Mem Access 1	S-CAM
	Mem Access 2	
	Broadcast	
	Writeback	

Figure 6: The integer pipelines are matched in depth such that any exception conditions are detected before a packet is committed to the processor state. This decision eliminates history buffers and other mechanisms to resolve race conditions between different pipeline depths and ensure in-order semantics.

into a decoupling FIFO at the head of the load store units.[2] At this point, the operation is architecturally considered as having executed, even if the result is not available (which would cause a recirculation condition for any dependent packet). When a memory slot becomes available, load operations are resolved from the memory hierarchy, and store results are stored in the store CAM which implements a multiprocessor-capable gated store buffer [16].

Since all architectural faults have been resolved previously during address translation, the only exceptions which can occur are related to bus errors and ECC failures. These can be caught as asynchronous exceptions and their handling deferred until a convenient break point at which precise PowerPC state can easily be materialized. Transitions between groups of translated code fragments provide such convenient break points. Synchronous exceptions such as page faults are required to be precise by the PowerPC architecture. Such exceptions can be handled by rolling back to a previous checkpoint (start of the current translated group) and interpreting PowerPC instructions until the error is re-encountered.

When a BOA exception occurs, the exception target is neither the PowerPC

[2] FIFO entry availability can be checked either during the issue phase for load operations, or load instructions finding a full FIFO can be rejected and recirculated.

exception address nor its translation, but a BOA stub which sets up the exception context. This BOA stub sets up registers as expected by the PowerPC architecture. For example, the stub places the effective address of the excepting PowerPC instruction in register **SRR0**, and the contents of the PowerPC *Machine State Register (***MSR***)* prior to the exception in register **SRR1**. Some exceptions require additional setup of this type by the stub. Only after this setup is complete, does the stub branch to the translation of the PowerPC exception handler.

5 Implementation using a Recirculation Buffer

High frequency in BOA is achieved by eliminating expensive wiring to implement the synchronous operation of the pipeline using global stall and exception conditions. Instead, the pipelines are implemented as dataflow elements. Correct operation is achieved through the use of the recirculation buffer.

Instead of checking for the existence of a stall before proceeding, the pipeline is automatically advanced every cycle. Upon issuing a new packet, the packet is both issued and copied into the recirculation buffer, which holds a copy of the contents of every packet currently executing. The existence of a stall in the execution pipeline may then be determined late in the execution process and indicated to the appropriate packets prior to their committing results during the **Writeback** stage.

Several approaches are possible to implement instruction streaming and recirculation. The main idea is to return the instruction to the issuable state if the instruction is canceled due to execution hazards. In its simplest form, this may involve returning the instruction address to the **Issue** stage and re-fetching the instruction. However, this approach has the disadvantage of increasing the penalty when a recirculation event occurs.

Thus, a better implementation choice buffers instructions which have previously been issued and which may need to be recirculated. Some implementations may propagate the instruction bits with the dataflow, but this will result in more state to be carried in the datapath. A better implementation of this concept logically passes the instructions along the pipeline stages, but *physically* keeps this information closer to the **Issue** stage by conceptually clustering the instruction portion of the pipeline registers in a *recirculation buffer*.

Thus, when packets are cancelled, the first dependent packet and all subsequent packets can be reissued from the recirculation buffer. The recirculating packets will repeat the process of issuing, progressing down the execution

pipeline. While the stall condition remains during reissue, the packets are continually canceled and reissued from the recirculation buffer until the processor stall completes. This assumed pipeline advancement strategy simplifies pipeline control. [17]

Although the conceptual picture implies that instructions which fail are retransmitted to the issue stage, the implementation of this approach is optimized with the recirculation buffer. The recirculation buffer keeps track and maintains a copy of instructions as they progress through the pipeline.

In BOA, we track a packet as the basic unit of instruction control. Since instructions are issued in packet, packets are the most natural way to track instructions. Notably, due to the decoupled fetch/execute pipeline, only mispredicted branches and exceptions need go further back than the **Issue** stage. Instead of reissuing on branch mispredictions and exceptions, BOA flushes the recirculation buffer and the **Fetch/Issue** stage and directs the **Issue** stage to fetch from the new address, which has either the correct branch target or the exception target, as appropriate.

Managing recirculation is particularly simple if the pipeline structure has been planned carefully so as to have predictable and matched pipeline depths (see Figure 6. For example, in the BOA design study, the **Issue** stage was followed by a register file access stage, an execution stage, and a **Broadcast** stage responsible for propagating the results of the execution stage throughout the entire chip (see Figure 5). Thus, after three stages it is known if any instruction cancel and reissue condition has been detected.

Figure 7 shows the recirculation buffer consisting of the issue queue and the reissue queue. The reissue queue matches three stages following **Issue** in the *Fixed-Point* pipe in Figure 6. After an instruction has been issued by the **Issue** logic, it is also copied into the recirculation buffer. Then, every cycle, the recirculation buffer progresses one cycle, matching the progress of the instructions in the execution pipeline. As can be seen in the Figure below, after three cycles (at the end of the **Broadcast** cycle), it is decided whether a recirculation condition is present. If so, the recirculation buffer contents are issued by the issue queue, otherwise the contents are retired from the recirculation buffer when the results of an instruction are committed to the processor state.

Introducing the recirculation buffer as an issue stage recirculation point reduces the recirculation penalty of aborted instructions by several cycles and helps to reduce the overhead of recirculation based pipeline management.

Referring to figure 6, this reduces the penalty for re-executing an instruction from 8 to 5 cycles. Figure 8 shows a possible execution sequence for a BOA-like

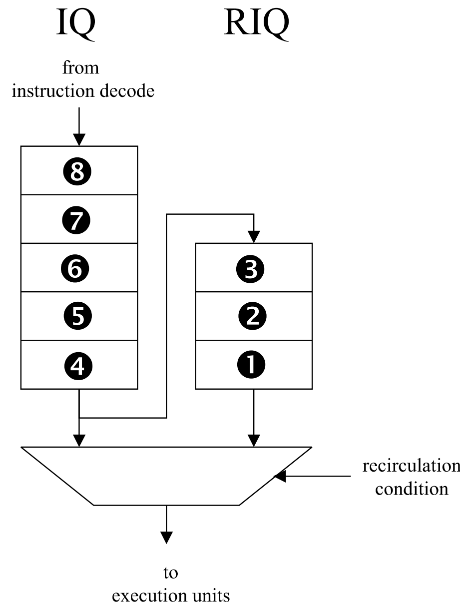


Figure 7: The recirculation buffer consists of the issue and re-issue queues. The issues queue holds as yet unissued packets (labeled 4 through 9), whereas the re-issue queue holds a copy of the packets in the register file stage (packet 3), the execute/address generation phase (packet 2), and the broadcast stage (packet 1). If a stall condition had occurred in packet 1, the results of packets 1 through 3 would not be committed and instead be re-issued from the re-issue queue.

recirculation-buffer based pipeline management scheme. The figure assumes a one cycle latency to distribute the control signal to cancel a packet. In the next cycle, state changes are disabled during the aborted operation's writeback phase, and packet reissue from the recirculation buffer is activated. Packet re-issue starts in the following cycle. According to this exemplary timing, the net CPI degradation for a 3 cycle stall event is 2 cycles.[3] Also, any stall conditions which have a latency longer than the exemplary 5 cycle recirculation penalty do not result in any penalty for this specific implementation.

This scheme impacts the design of the overall pipeline in several ways. *First*, if a resource is not available, an instruction does not remain in that processing element until the resource is available. Instead, the instruction is aborted and

[3]Aggressive upper level metal distribution of a few select cancel signals may allow the cancel signal to arrive early enough to start packet re-issue one cycle earlier.

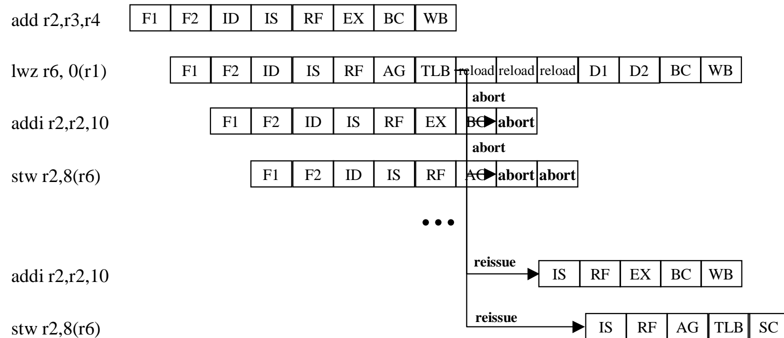


Figure 8: Pipeline diagram for recirculation-based pipeline management using the recirculation buffer.

immediately passed to the downstream pipeline stage. As a result, each instruction can remain in a pipeline stage for exactly one cycle. If the instruction cannot complete by that time, it is aborted.

Aborted instructions continue to pass through the processor pipeline until they reach a pipeline stage which checks for aborted instructions. When this stage encounters an instruction which has been aborted, it recirculates this instruction back to the **Issue** stage which then re-issues the instruction. Recirculation can take multiple cycles using staging buffers, and thus wire delay for communication between the stage deciding which instructions to recirculate and the **Issue** stage is non-critical.

Second, the operation of the recirculation buffer leads to a design in which updates of the processor state are centralized.

Third, this scheme integrates exception and stall handling in a single structure, so that no separate mechanisms have to be maintained. Just as we have described for stalls, all pertinent state describing exception conditions is saved, and the instruction passes downstream in the pipeline.

6 Related Work

In high frequency design, the guTS test site led to the first GHz processor design. The guTS design employed novel, self-resetting dynamic logic that resulted in high circuit performance. However, guTS was a test system which implemented

the basic architecture for a processor core, but not a fully compatible POWER architecture. By contrast, Rivina and BOA were architected as fully POWER compliant designs. To achieve full compatibility while maintaining the design simplicity needed to achieve high performance, Rivina and BOA use different methods to deal with complex aspects of the PowerPC architecture. Rivina relies on traps into microcode where complex functions are implemented for full PowerPC compatibility. Other microarchitecture techniques were used in Rivina to achieve compatibility without burdening the implementation's critical path. For example, Rivina employs *dependency prediction* to avoid a full dependence analysis in the critical path. By contrast, BOA resolves this problem during software rescheduling in the binary translation phase. As discussed in Section 4, BOA's instruction format tells the hardware what operations may execute in parallel.

Current processor control mechanisms have been adapted in a number of ways for complex, high-frequency designs. For example, partitioning of a design allows simpler smaller subdesigns. [18].

An alternative approach has been described in [8] which uses backup registers for the entire processor state. Global communication signals are distributed during the subsequent computation cycle, in parallel with continued processing. If the controller determines that any stall or exception conditions were encountered, the controller can back out any changes and revert the processor state to the previous cycle using the backup registers. This leads to complicated stall and exception logic, which is undesirable.

Sutherland describes an approaches for pipeline control without a central controller based on micropipelines and counterflow pipelines [19]. Micropipelines use only communication with adjacent pipeline stages, eliminating the need for a global controller. This has made this approach attractive for implementation in asynchronous designs. Eliminating global control does not remove the cost of the wire delay, since the wire delay now ripples through successive elements in the pipeline.

Several asynchronous processor designs are based on the micropipeline processor concept referenced previously. Unlike synchronous processor designs, these designs do not use a global processor clock to synchronize processor operation. This eliminates the need to allocate time for communication in each clock cycle, and communication time is only allocated when it is actually required.

The AMULET processor described by Woods et al [20] is an asynchronous implementation of an in-order processor. The processor uses a scoreboard for determining operand availability [21]. Other resource availability requests are resolved using stalls which can propagate through the processor. Exception han-

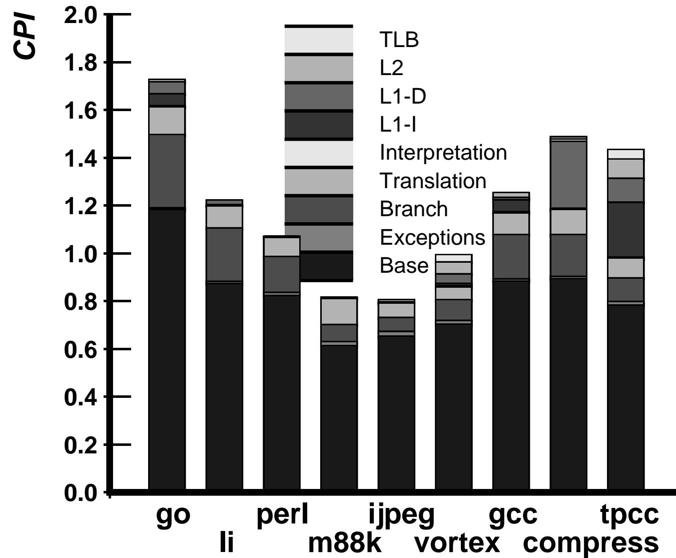


Figure 9: BOA CPI

dling in this processor is performed using a scheme based on squashing operations after an exception has occurred using "color bits." These prevent any operations from committing until the exception handler starts to pass through the pipeline.

The Fred processor described by W. Richardson and E. Brunvand [22] implements an asynchronous out-of-order processor. Stalls and resource management are similar to the AMULET1 processor implementation, but the exception mechanism relies on maintaining a buffer of out-standing instructions which may raise exceptions in an issue buffer. When an instruction encounters an abnormal situation, it returns to the issue buffer and raises an exception.

7 Performance Results and Conclusion

We have presented the BOA high-frequency pipeline architecture and high-frequency design constraints influenced the overall processor structure. The architecture uses binary translation to layer the PowerPC architecture on a simpler EPIC core and uses a novel pipeline management approach to reduce the impact of wire delay on the processor control functions.

To investigate the impact of the binary translation approach and the two-cycle latency for even simple ALU operations, we have performed a detailed CPI analysis based on instruction traces for SPECint95 and TPC-C (see figure 9), which

have previously been reported in [12, 3, 13].

To eliminate the impact of wire delay, we have opted for a pipeline control mechanism using only local communication. The centerpiece of this architecture approach is the recirculation buffer, which is responsible for issuing and tracking instructions. When a pipeline hazard occurs, compromised instructions are aborted and re-issued by the recirculation buffer. This design approach allows the design of a dataflow-like architecture in the datapath, and reduce the cost of global synchronization to interlock the system during stall and exception handling.

References

[1] J. Silberman, N. Aoki, D. Boerstler, J. Burns, S. Dhong, A. Essbaum, U. Ghoshal D. Heidel, P. Hofstee, K.T. Lee, D. Meltzer, H. Ngo, K. Nowka, S. Posluszny, and O. Takahashi. 1.0-GHz single-issue 64-bit PowerPC integer processor. *IEEE Journal of Solid State Circuits*, 33(11):1600–1607, November 1998.

[2] P. Hofstee, N. Aoki, D. Boerstler, P. Coulman, S. Dhong, B. Flachs, N. Kojima, O. Kwon, K. Lee, D. Meltzer, K. Nowka, J. Park, J. Peter, S. Posluszny, M. Shapiro, J. Silberman, O. Takahashi, and B. Weinberger. A 1GHz single issue PowerPC processor. In *2000 IEEE International Solid State Circuits Conference – Digest of Technical Papers*, pages 92–93, San Francisco, CA, February 2000. IEEE.

[3] M. Gschwind, E. Altman, S. Sathaye, P. Ledak, and D. Appenzeller. Dynamic and transparent binary translation. *IEEE Computer*, 33(3):54–59, March 2000.

[4] R. Hilgendorf and W. Sauer. Instruction translation for an experimental S/390 processor. *Computer Architecture News – Special Issue: Workshop on Binary Translation 2000*, 29(1):37–42, March 2001.

[5] C. Moore. Power-4. Presentation at Microprocessor Forum 2000, 2000.

[6] K. Diefendorff. Power4. *Microprocessor Report*, 2000.

[7] K. Ebcioğlu and E. Altman. DAISY: dynamic compilation for 100% architectural compatibility. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 26–37, Denver, CO, June 1997. ACM.

[8] K. Ebcioğlu, J. Fritts, S. Kosonocky, M. Gschwind, E. Altman, K. Kailas, and T. Bright. An eight-issue tree-VLIW processor for dynamic binary translation. In *Proc. of the 1998 International Conference on Computer Design (ICCD '98) – VLSI in Computers and Processors*, pages 488–495, Austin, TX, October 1998. IEEE Computer Society.

[9] K. Ebcioğlu, E. Altman, S. Sathaye, and M. Gschwind. Execution-based scheduling for VLIW architectures. In *Euro-Par '99 Parallel Processing – 5th International Euro-Par Conference*, number 1685 in Lecture Notes in Computer Science, pages 1269–1280. Springer Verlag, Berlin, Germany, August 1999.

[10] M. Gschwind, K. Ebcioğlu, E. Altman, and S. Sathaye. Binary translation and architecture convergence issues for IBM System/390. In *Proc. of the International Conference on Supercomputing 2000*, Santa Fe, NM, May 2000. ACM.

[11] K. Ebcioğlu, E. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 2001. in press.

[12] S. Sathaye, P. Ledak, J. LeBlanc, S. Kosonocky, M. Gschwind, J. Fritts, Z. Filan, A. Bright, D. Appenzeller, E. Altman, and C. Agricola. BOA: Targeting multi-gigahertz with binary translation. In *Proc. of the 1999 Workshop on Binary Translation*, IEEE Computer Society Technical Committee on Computer Architecture Newsletter, pages 2–11, December 1999.

[13] E. Altman, M. Gschwind, and S. Sathaye. BOA: the architecture of a binary translation processor. Research Report RC21665, IBM T.J. Watson Research Center, Yorktown Heights, NY, March 2000.

[14] J. Moreno, M. Moudgill, K. Ebcioğlu, E. Altman, C. B. Hall, R. Miranda, S.-K. Chen, and A. Polyak. Simulation/evaluation environment for a VLIW processor architecture. *IBM Journal of Research and Development*, 41(3):287–302, May 1997.

[15] M. Gschwind. Method and apparatus for the selective scoreboarding of computation results. *Research Disclosures*, 2001. in press.

[16] E. Altman, K. Ebcioğlu, M. Gschwind, and S. Sathaye. Methods and apparatus for reordering and renaming memory references in a multiprocessor computer system. Filed for U.S. Patent, March 2000.

[17] M. Gschwind. Pipeline control mechanism for high-frequency pipelined designs. US Patent 6192466, January 1999.

[18] Subbarao Palacharla, Norman P. Jouppi, and J.E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, Denver, CO, June 1997. ACM.

[19] I. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[20] Woods et al. AMULET1: An Asynchronous ARM Microprocessor. *IEEE Transactions on Computers*, 46(4):385–398, April 1997.

[21] N. Paver. Condition Detection in Asynchronous Pipelines. UK Patent No. 9114513, October 1991.

[22] W. Richardson and E. Brunvand. Fred: An Architecture for a Self-Timed Decoupled Computer. In *Proceedings of the Second International Symposium in Asynchronous Circuits and Systems*, pages 60–68. IEEE Computer Society Press, June 1996.