Loose Loops Sink Chips

Eric Borch
Intel Corporation, VSSAD
eric.borch@intel.com

Eric Tune*
University of California, San Diego
Department of Computer Science
etune@cs.ucsd.edu

Srilatha Manne Joel Emer
Intel Corporation, VSSAD
[srilatha.manne, joel.emer]@intel.com

Abstract

This paper explores the concept of micro-architectural loops *and discusses their impact on processor pipelines. In particular, we establish the relationship between* loose loops *and pipeline length and configuration, and show their impact on performance. We then evaluate the* load resolution loop *in detail and propose the* **distributed register algorithm** *(***DRA***) as a way of reducing this loop. It decreases the performance loss due to load mis-speculations by reducing the issue-to-execute latency in the pipeline. A new loose loop is introduced into the pipeline by the DRA, but the frequency of mis-speculations is very low. The reduction in latency from issue to execute, along with a low mis-speculation rate in the DRA result in up to a 4% to 15% improvement in performance using a detailed architectural simulator.*

1 Introduction

Micro-architectural loops are fundamental to all processor designs. We define micro-architectural loops as communication loops which exist wherever a computation in one stage of the pipeline is needed in the same or an earlier stage of the pipeline. Loops are caused by control, data, or resource hazards.

Figure 1 illustrates the basic components of a loop. The *initiation stage* is the stage where data from a succeeding stage is fed back. The *resolution stage* is the stage that computes the result needed by a preceding stage. The *loop generating instruction* is the instruction which initiates the loop. For branches, the loop generating instruction is a branch, the loop initiation stage is the fetch stage, and the

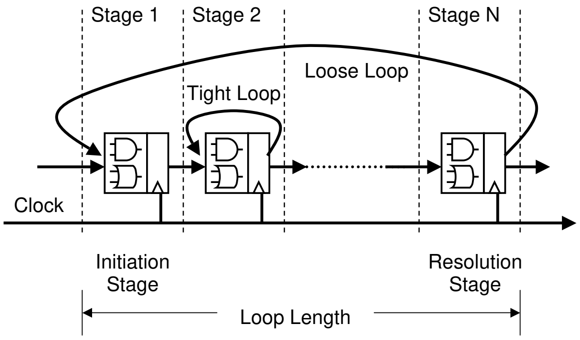

Figure 1. Micro-architectural loops

loop resolution stage is the execute stage. *Loop length* is defined to be the number of stages traversed by a loop, and the *feedback delay* is the time required to communicate from the resolution stage to the initiation stage. *Loop delay* is the sum of the loop length and feedback delay. Loops with a loop delay of one are referred to as *tight loops*; all other loops are referred to as *loose loops*.

With a loop delay of one, tight loops feed back to the same pipeline stage. Figure 2 shows examples of tight loops in the Alpha 21264 processor, such as the next line prediction loop and the integer ALU forwarding loop [3]. The next line prediction in the current cycle is needed by the line predictor to determine the instructions to fetch in the next cycle, while the ALU computation in the current cycle is required in the ALU in the next cycle to support back-to-back execution of dependent instructions. Tight loops directly impact the cycle time of the processor because the information being computed is required at the beginning of the next cycle.

Loose loops extend over multiple pipeline stages. Examples of loose loops in the Alpha 21264 (Figure 2) are the *branch resolution loop* and the *load resolution loop*. The

[1]Eric Tune did this work while at VSSAD.

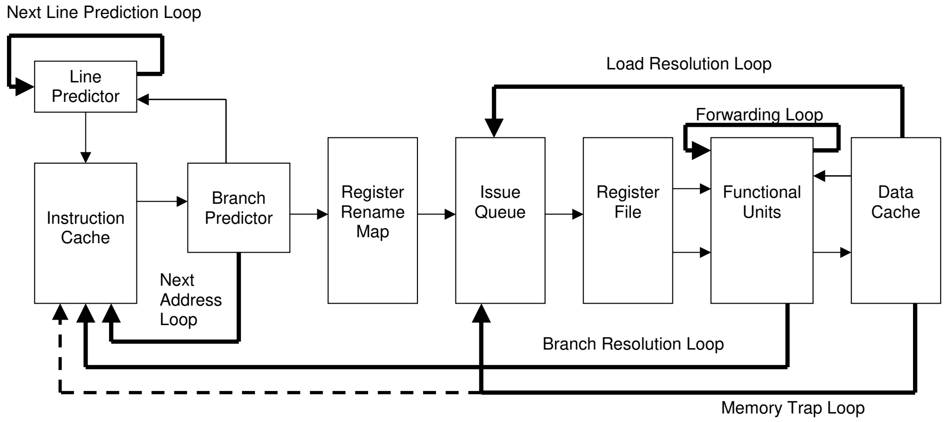

Figure 2. Examples of micro-architectural loops in the Alpha 21264 processor [3], represented by solid lines. If the recovery stage is different from the initiation stage, it is shown by a dotted line.

branch resolution loop is caused by a control hazard resulting from the existence of an unresolved branch in the pipeline. The fetch unit does not know with certainty which instruction to fetch until the branch resolves. The load resolution loop is caused by a data hazard resulting from an unresolved load operation, e.g., did the load hit in the cache or not. The loop occurs because instructions dependent on the load cannot issue until the issue queue knows the time at which the load data will be available.

Loose loops can impact performance by restricting the pipeline's ability to do useful work. The simplest way to manage a loose loop is to stall the pipeline while waiting for the loop to resolve. Since no progress is made while the pipeline stalls, performance will be negatively impacted. Thus, an alternative technique of speculating through the loop is often used. Speculating through a loop improves performance by attempting to make progress while the loop resolves instead of simply waiting. Stalling the processor means that no progress is made every time a loose loop is encountered, while speculating through the loop allows progress to be made, except when there is a mis-speculation.

Stalling the processor is often an acceptable solution when the loop length is small. With a simple, 5 stage pipeline, a single cycle branch bubble may not have a significant impact on performance. Stalling the processor is also a tenable solution when the loop occurs infrequently. The memory barrier loop in the Alpha 21264 is an example of an infrequent loop. When a memory barrier instruction is encountered in the pipeline, the mapping logic stalls the memory barrier instruction and all succeeding instructions. The instructions are released to issue when all preceding instructions have completed.

As pipelines get longer and/or loops occur more frequently, speculation is used to manage loose loops and maximize performance. There are many examples of speculation in the Alpha 21264, such as branch prediction, load hit prediction, and memory dependence prediction. As long as there are no mis-speculations, loose loops do not impact performance because the pipeline is always doing useful work.

When mis-speculations occur, however, the pipeline has done work which must be thrown away. The pipeline must then recover, and restart processing from the mis-speculated instruction. Mis-speculation recovery may occur at the loop initiation stage or, due to implementation reasons, at an earlier stage in the pipeline which we call the *recovery stage*. The presence of a recovery stage introduces a recovery time, that is, the time it takes the useful instructions to refill the pipeline from the recovery stage to the initiation stage. For example, the initiation stage for load/store reorder traps in the 21264 is the issue stage, while the recovery stage is the fetch stage. The dotted lines in Figure 2 illustrate where the recovery stage is earlier than the initiation stage.

The best performance is achieved when there are no mis-speculations. Every mis-speculation degrades performance. Clearly the frequency of loop occurrence (i.e., the number of loop generating instructions) and the mis-speculation rate are first order determinants of the performance lost. How much performance is lost per mis-speculated event is a complex function of a number of parameters. One measure of this is the amount of work discarded due to each mis-speculation. We term this *useless work*.

The product of the frequency of loop occurrence and the mis-speculation rate determines the number of times useless work is done. For example, the number of branch mis-speculation events is greater in programs with a high occurrence of branches and a high mis-prediction rate, such as integer programs. The amount of useless work due to each mis-speculation is a function of the time required to resolve the loop, and the time required to recover from the mis-speculation. The greater these latencies, the greater the impact of the mis-speculation.

The lower bound of this time is equal to the loop delay plus the recovery time. The actual latency is augmented by any queuing delays within the loop. For instance, the branch resolution loop length for the Alpha 21264 encompasses 6 stages, the feedback delay is 1 cycle [3], and there is no recovery time. Thus, the minimum impact of a mis-speculation is 7 cycles.

The long pipelines in current generation processors increase the loop delay for many loose loops. The Pentium4 design, for example, has a pipeline length greater than 20 stages and a branch resolution latency on the order of 20 cycles [12]. Pipeline lengths are increasing for two reasons: higher operating frequency and more architecturally complex designs. A high operating frequency shrinks the clock cycle, resulting in fewer stages of logic fitting within a cycle [1]. Operations that required only one cycle in the past now take multiple cycles. Architectural advances that increase overall processor throughput also increase pipeline lengths. Wide issue widths, out-of-order instruction issue, speculation, and multi-threading increase the amount of logic on the processor, and more logic often requires more pipeline stages to complete an operation.

This paper looks at the impact pipeline lengths, pipeline configurations, and loose loops have on processor performance. We show that performance is not just affected by the overall length of the pipeline, but by the length of critical portions of the pipeline that are traversed by key loose loops. In particular, we focus on the load resolution loop and the impact loop length and queuing delay have on this loop. Based on our analysis, we propose a design modification called the **distributed register algorithm** (**DRA**). It moves the time consuming register file access out of the issue to execute path and adds a register caching mechanism. Using the detailed ASIM [4] architectural simulator, we show speedup improvements of up to 4% to 15% relative to the base model.

The rest of the paper is organized as follows. Section 2 describes the base architecture and details the load resolution loop. Section 3 shows the impact pipeline lengths and configurations have on performance. Sections 4 and 5 discuss the reasoning behind the proposed design modification and detail the DRA. Results are presented in section 6. Section 7 explains the relation of our work to prior work related to register files, and section 8 concludes the paper.

2 Processor Model

We modeled our base processor to be comparable to next generation super-scalar processors [5, 10]. It is an 8-wide issue machine with branch prediction, dynamic instruction scheduling, and multi-threaded execution. It contains a unified, 128 entry instruction queue (IQ), and has up to 256 instructions in flight at any time. The minimum pipeline delay for an integer operation with single cycle execution latency is similar to the Pentium4 [12], around 20 cycles, assuming no stalls or queuing delays in the pipeline. Other operations may take longer depending on their execution latency.

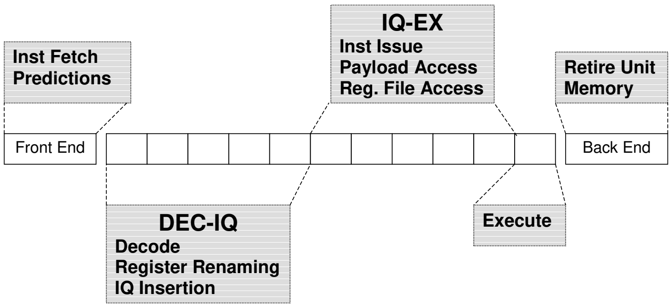

Figure 3. Pipeline of base architecture

Figure 3 shows the pipeline of the simulated machine. In this work we focus on the latency between instruction decode and execute. These are labeled in the figure as **DEC-IQ** (decode to insertion into the instruction queue) and **IQ-EX** (issue from instruction queue to execute). Many micro-architectural loops traverse this portion of the pipeline, such as the branch resolution loop and the memory dependence loop, and processor performance is highly sensitive to the length of this region.

Our simulated architecture uses a clustered design similar to the 21264 [14]. In particular, we cluster the instruction scheduling logic. As noted in [2], issuing M instructions out of an instruction queue of N entries is difficult to implement when M and N are large. In our case, $M = 8$ and $N = 128$. Therefore, we allocate or slot the instructions at decode to one of eight functional unit clusters. Now the problem of issuing M out of N instructions is reduced to issuing one out of approximately $N/8$ instructions, assuming that instructions are distributed uniformly to all clusters. The DRA uses the cluster allocation information to determine which functional units receive which source operands. This is described in detail in Section 5.

2.1 Pipeline Latency

The DEC-IQ latency is 5 cycles in the base model. The 5 cycles are required for instruction decoding, register renaming, wire delay, and insertion of the instruction into the IQ. The latency from IQ-EX is also 5 cycles, which is required for determining which instruction to issue, reading the pertinent information from the instruction payload, performing register file access, and delivering the instruction and the source operands to the functional units. The IQ only contains the dependency information required to determine when an instruction is ready to issue. The instruction payload contains the rest of the state for the instruction, such as physical register numbers, op codes, and offsets.

Register file access takes 3 cycles, with two of the cycles required purely to drive data to and from the register file. The wiring delay is due to the large register file required to support an 8 wide issue, out-of-order, SMT machine with both physical registers and architectural state for all threads. Furthermore, we require 16 read ports and 8 write ports to support 8-wide issue.

The register file design could have fewer ports. The full port capability is not needed in most cases because either the operands are forwarded from the execution units, or the number of instructions issued is less than 8, or not all instructions have 2 input operands and one output operand [15]. However, there are implementation problems with reducing the number of ports. First, which operands are being forwarded is not known at the time of issue, and the register file and forwarding structure are accessed simultaneously, not sequentially. Accessing them sequentially would add additional delay into the IQ-EX path. Therefore, we cannot suppress the reading of the register file on a hit in the forwarding logic. Second, if there are fewer read port pairs than functional unit clusters, a complex switching network is needed to move operands to the correct functional units. This also adds additional delay into the IQ-EX path. Third, if 16 operands are needed in a cycle, there must be some logic to stall or suppress instructions that will not be able to read their operands. For these reasons, a register file with full port capability can be easier to implement and manage, and reducing the number of ports adds unnecessary complexity.

2.2 Managing Issue Loops

The 5 cycle IQ-EX latency introduces some challenging architectural problems. The long latency produces two loose loops that are managed in different ways to optimize performance.

2.2.1 Forwarding Buffer

A loose loop exists between the execution stage and the register file read stage. A value computed in the execution stage needs to be written back to the register file before dependent instructions can read the value. Forwarding logic is added in the execution stage to shrink this loop from a loose loop to a tight loop. Without forwarding, dependent instructions must wait to issue until their operands are written to the register file. While these instructions stall, the pipeline has fewer instructions to execute, resulting in less available work. Forwarding enables instructions to get their operands from the forwarding logic in the ALUs without having to wait for them to be written to the register file. It replaces this loose loop with a tight loop in the execution logic that makes the result computed in a previous cycle available in the current cycle.

The base model contains a *forwarding buffer* which retains results for instructions executed in the last 9 cycles. Five of these cycles are required to cover long latency operations and limit the number of write ports on the register file. The other four cycles cover the time it takes for the resulting data to be written to the register file. The forwarding buffer is required for the base architecture to work. As will be discussed in Section 5, it is also an integral part of our redesign for the register file.

2.2.2 Load Resolution Loop

The load resolution loop is caused by the non-deterministic latency of loads. Although the latency of a cache hit is known, whether the load will hit, miss, or have a bank conflict in the cache is unknown. This makes it difficult to know when to issue load dependent instructions so that they arrive at the functional units in the same cycle that the load data arrives. It is this unknown that necessitates a loose loop. As with all loose loops, the pipeline could either stall or speculate. The Alpha 21064 and 21164 processors stall the pipeline [7, 8] until the load resolves, while the Alpha 21264 can speculatively issue dependent instructions and recover on a load miss [3].

In our base processor, stalling load dependent instructions effectively adds 5 cycles to the load-to-use latency. Stalling may prevent useful work from getting done. To avoid this, the base processor speculatively issues load dependent instructions, i.e., it predicts that all loads hit in the data cache.

To help discuss the load resolution loop, we define two terms. The *load dependency tree* is defined to include the instructions directly or indirectly dependent upon the load. The set of cycles in which any instructions within the load dependency tree may issue speculatively is called the *load shadow*. In the 21264, the load shadow for integer loads starts 2 cycles after the load issues, and ends when the load signals a data cache hit or miss [3].

Load Mis-speculation Recovery As long as the load hits, there is no penalty for the speculation. A miss, however, requires a costly recovery action. For correct operation, all instructions within the load dependency tree that have already issued must be reissued after the load resolves. Implementation choices can further increase the mis-speculation penalty. In the Alpha 21264, the amount of useless work can be even higher on an integer load miss because all instructions issued in the load shadow, whether they are in the load dependency tree or not, are killed and reissued. Fortunately, most programs have a high load hit rate, and the overall cost of recovering from mis-speculations is significantly less that the cost of a non-speculative policy.

We also have a choice of recovery points for a load mis-speculation. We could make the recovery stage the issue stage, by reissuing load dependent instructions from the IQ. Alternatively, the fetch stage could be the recovery stage by flushing the pipeline and re-fetching instructions starting with the first instruction after the mis-speculated load. Re-fetching is easier to implement, but it dramatically increases the loop recovery time. Our results show that it performs significantly worse than reissue. Hence, it was not considered further.

IQ Pressure Our base architecture reissues instructions on a load miss. Unlike the 21264, we do not reissue all instructions issued within the load shadow; we only reissue instructions that are within the load dependency tree. The number of instructions reissued is equal to the useless work performed due to load mis-speculations. For each reissued instruction, there was a previous issue of the same instruction that was killed.

Although load speculation with reissue performs better than no speculation or speculation with re-fetch, it puts additional pressure on the IQ. The reissue mechanism requires the IQ to retain all issued instructions until it is notified by the execution stage that the instructions do not have to reissue. The occupancy time of instructions in the IQ is therefore a function of the IQ-EX latency. The longer the instructions reside in the IQ after issuing, the less space there is for new, unissued instructions. In our base model, the loop delay is 8 cycles (loop length of 5 cycles and feedback delay of 3 cycles). Thus, it takes 8 cycles from the time an instruction issues to the time the IQ is notified by the execution stage that the instruction does not have to reissue and can be removed. Once an instruction is tagged for eviction from the IQ, extra cycles are needed to clear the entry. If the machine is operating near full capacity of 8 IPC, more than half the entries in the IQ may be already issued instructions in the load dependency tree waiting for the load to resolve. The instruction window effectively shrinks, resulting in less exposed instruction level parallelism (ILP), and potentially, a reduction in useful work.

3 Impact of Loop Length

As pipeline lengths increase, so do the loop lengths for many loose loops, resulting in more useless work done per mis-speculation. In this section we quantitatively look at the impact longer pipelines have on performance. Furthermore, we also investigate various pipeline configurations and show that performance is a function of not just the length of the pipeline, but the configuration of latencies across the pipeline.

3.1 Increasing Pipeline Lengths

Figure 4 shows the results from increasing the pipeline length of our base machine running a sampling of integer, floating point, and multi-threaded benchmarks. The decode to execute portion of the pipeline is varied from 6 to 18 cycles, in increments of 4 cycles (2 cycles each for DEC-IQ and IQ-EX). Speedup numbers are shown relative to the 6 cycle case.

We use the Spec95 benchmark suite in our analysis. For single threaded runs, we skip from 2 to 4 billion instructions, warm up the simulator for 1 to 2 million instructions, and simulate each benchmark from 90 to 200 million instructions. The multi-threaded benchmarks are simulated for 100 million instructions total for both threads. Each program in the multi-threaded run skips the same number of instructions as in the single-threaded run. All benchmarks are compiled with the official Compaq Alpha Spec95 flags.

The greater the pipeline length, the longer the loop delay for loops which traverse the pipeline between decode and execute. Figure 4 shows that increasing the pipeline length by 12 cycles results in a performance loss of up to 24% due to mis-speculation on loose loops. The two primary loops in this region are the branch resolution loop and the load resolution loop. All benchmarks show a reduction in performance as the pipeline lengthens.

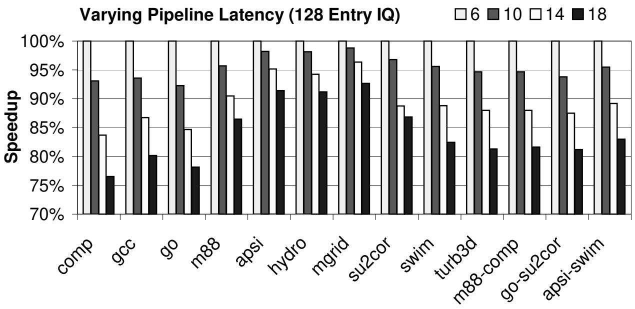

Figure 4. Performance for varying pipeline lengths. Pipeline length is varied between *decode* and *execute*. Performance is shown relative to the case with **6** cycles between decode and execute. Numbers less than 100% indicate a performance loss. Note that the Y-axis begins at 70%.

Integer programs are generally impacted by mis-speculations on the branch resolution loop. *compress, gcc* and *go* perform a significant amount of useless work due to branch mis-predictions. Furthermore, they are also burdened by load misses, resulting in additional performance degradation. *m88ksim*, however, does not have as many branches or branch mis-predictions as the other integer benchmarks, resulting in less sensitivity to the branch resolution loop delay.

We expected floating point programs to be sensitive to the load resolution loop primarily due to a large number of load operations and a high load miss rate. Some programs behave as expected, such as *turb3d* and *swim*. Both programs have a reasonable number of loads and load misses in the data cache, indicating many mis-speculation events. *turb3d* also suffers from a fair number of data TLB misses, where recovery from the beginning of the pipeline impacts performance, not just the latency of the IQ-EX portion.

Two programs, *hydro* and *mgrid*, also have a large number of loads and a high data cache miss rate, but are not particularly sensitive to pipeline length. Unlike *turb3d* and *swim*, both these programs also suffer from misses in the second level cache. Therefore, the performance of these programs is dominated by the long main memory access latency. The loop delay of the load loop, even with a long pipeline, is insignificant in comparison to the main memory access latency.

apsi has a reasonably high data cache miss rate; however, the amount of useless work performed due to load mis-speculations, as indicated by the number of instructions reissued, is small. The relatively low IPC of *apsi*, combined with the small amount of useless work performed per mis-speculation, suggests that *apsi* has long, narrow dependency chains restricting ILP. Therefore, the performance of *apsi* is determined more by program characteristics than by the load resolution loop delay.

su2cor does not suffer from many branch or load mis-speculations. However, analysis shows that there is a measurable amount of useless work resulting from branch mis-predictions, as noted by the number of instructions killed due to branch predictions. Therefore, although the number of branch mis-speculations is small, the resolution latency is large due to queuing delays in the pipeline. Increasing the pipeline length only exacerbates the situation since the loop length defines the lower bound for resolution latency.

Pipeline length impacts multi-threaded performance in the same manner as it impacts the component programs. However, the degree of impact is generally less than that of the worst performing component program. For example, *go-su2cor* has a smaller performance loss than *go* alone. In multi-threaded execution, the availability of multiple threads prevents the pipeline from issuing deeply down a speculative path [16]. Furthermore, when a mis-speculation occurs on one thread, the other thread(s) can continue doing useful work while the mis-speculated thread recovers.

3.2 Not All Pipelines are Created Equal

Figure 5 shows the results from varying the pipeline configuration while retaining the same overall pipeline length. The syntax **X_ Y** represents the latency from DEC-IQ (**X**), and IQ-EX (**Y**). Note that $X + Y$ is constant. The speedup shown is relative to the 3_ 9 configuration, with 3 cycles from DEC-IQ and 9 cycles from IQ-EX. The results indicate that not all pipelines are created equal, and reducing the latency from IQ-EX improves performance by up to 15%, even when the overall length of the pipeline remains the same.

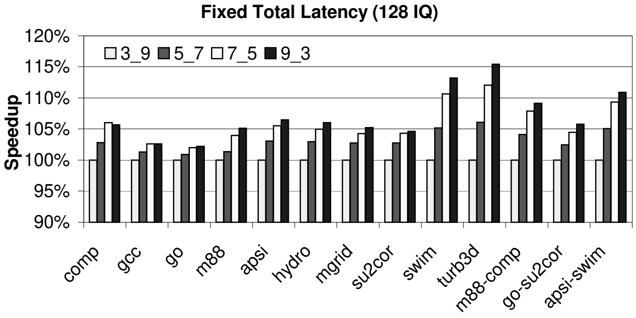

Figure 5. Performance for a fixed overall pipeline length. The first number in the legend represents the cycles from DEC-IQ, while the 2nd number represents the cycles from IQ-EX. All data is shown as speedup relative to to the 3_9 case. Note the y-axis begins at 90%.

Loose loops determine how sensitive the processor is to various pipeline configurations. Loose loops in our base processor either traverse the entire decode to execution portion of the pipeline (branch resolution loop), or traverse just the IQ-EX portion (load resolution loop). There are no loops which exist only within the DEC-IQ section. By moving pipeline stages from the IQ-EX portion to the DEC-IQ portion, we reduce the length of loops contained within the IQ-EX region without jeopardizing loops in other sections.

The benchmarks with the greatest performance improvement in Figure 5 (*swim, turb3d*, and *apsi-swim*) are a subset of the programs that showed the most sensitivity to increasing pipeline lengths in Figure 4. The rest of the benchmarks are either sensitive to the branch resolution loop length, which does not change in these simulations (*compress, gcc, go, m88ksim, su2cor, m88ksim-compress* and *go-su2cor*), or are impacted by other benchmark characteristics such as low ILP (*apsi*) or misses to main memory (*hydro* and *mgrid*).

Reducing the IQ-EX latency, even at the cost of increasing DEC-IQ latency, improves performance. The access to a large register file dictates much of this latency. Farkas, et.

al. [11] also note that large register files might limit the performance benefit of wide, super-scalar machines. In the rest of the paper, we present the **distributed register algorithm** (DRA). It reduces the IQ-EX latency and the latency of the overall pipeline by moving the time consuming register file access out of the IQ-EX path and placing it in the DEC-IQ path. This primarily reduces the loop delay of the load resolution loop, resulting in less capacity pressure on the IQ and less useless work done due to load mis-speculations.

4 Distributed Register Algorithm

Much of the IQ-EX latency is determined by the register file access time of 3 cycles. Therefore, the obvious method for reducing the IQ-EX latency is to move the register file access out of this path and replace it with a register cache. Register caches are small, generally on the order of 16 to 32 entries. Given their size, they can be placed close to the functional units. The size and placement reduce register access delays and transfer latencies, allowing a register access latency of one cycle in the general case. This effectively shrinks the IQ-EX latency from 5 cycles to 3 cycles, resulting in a shorter load resolution loop delay and less wasted work on a load miss.

A register cache contains a subset of all registers; hence a register cache, unlike a register file, can suffer from operand misses. The data hazard resulting from not knowing whether the operand will hit or miss in the register cache introduces a new loose loop to the pipeline, called the *operand resolution loop*. The operand resolution loop has a high frequency of loop occurrence — every instruction that has input operands is a loop generating instruction. Since the number of mis-speculation events is the product of the frequency of loop occurrence and the mis-speculation rate, even a small operand miss rate is detrimental to performance. If the number of register cache misses is high enough, then the amount of work wasted due to register cache misses can offset the savings from a reduced IQ-EX latency.

Register caches must be small to reduce access latency. Given that they are fully associative structures, they need to be on the order of 16 to 32 entries to achieve a single cycle access latency. A small register cache results in a high miss rate for our base architecture because determining which values to insert into the cache is a difficult task. Register values are frequently used just once [6], so many of the entries in the register cache may never be accessed if they are forwarded to the consumer through the forwarding buffer. Also, the number of cycles between the availability of operands for an instruction can be quite large in a wide issue, out-of-order machine.

Figure 6 shows the cumulative distribution function for the time (in cycles) between when an instruction's first operand is available, and when the second operand is available for *turb3d*. The time is zero for instructions with only one operand. 25% of all instructions have 25 cycles or more between the availability of operands. In fact, even the 9 cycle forwarding buffer in our base architecture only covers about 50% of all instructions. Other benchmarks show similar characteristics. A register cache may need to be of comparable size to a register file to hold all the relevant information for the instructions in flight.

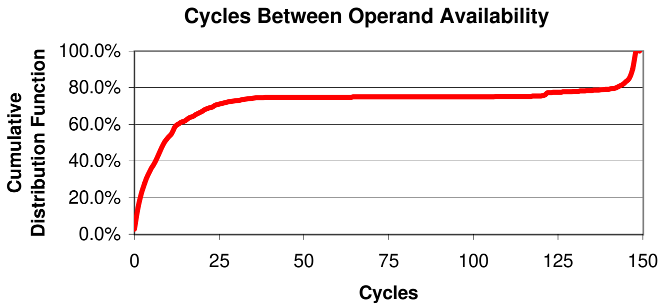

Figure 6. Cumulative distribution function of the time in cycles between when an instruction's first operand and second operand is available. Data is shown for *turb3d*.

To increase the capacity of the register cache without increasing the access latency, the DRA takes advantage of the clustered architecture of the base model and places a small register cache called a *clustered register cache* (CRC) within each functional unit cluster. There are eight CRCs, each with 16 entries. This effectively increases the size of the register cache to 128 entries.

To more effectively manage the entries in the CRC, the DRA does one of two things. First, each CRC only stores those operands required by instructions assigned to that functional unit cluster. An instruction is assigned a functional unit cluster when it is decoded. Therefore, the DRA may direct the operands for this instruction to a specific cluster. Note that the same operand may be stored in multiple CRCs if it is consumed by instructions assigned to different clusters.

Second, each CRC only stores operands for a consuming instruction that is unlikely to get the operand through other means. To achieve this, we note that one can classify operands by how a consuming instruction gets those operands. The three classes are: *completed operands*, *timely operands*, and *cached* operands.

Completed operands are already in the register file when a consuming instruction is decoded and can be read at any time. These operands tend to be associated with global registers such as the global pointer and stack pointer. However, if a register is alive long enough, it can be a completed operand for many instructions. The DRA reads completed operands from the register file in the DEC-IQ path — after the register is renamed and before the instruction is inserted into the IQ. When the instruction enters the IQ, the operand is inserted into the payload for retrieval when the instruction issues. Accessing the register file out of the issue path was proposed by Tomasulo and others [13].

Timely operands are those where the consumer of the operand is issued not long after the producer of the operand has issued. The forwarding buffer already inherent to our base model handles this category of operands by storing all values computed in the last 9 cycles.

Cached operands are those that are inserted into the CRCs. To reduce the capacity pressure on the register cache, only those operands who have consuming instructions that neither pre-read the operand from the register file nor read it from the forwarding buffer are placed in the register cache. This can happen when an instruction's operand is not in the register file for pre-read, nor does the instruction issue soon enough after its producer to get the value from the forwarding buffer.

Note that the classification of an operand is determined by where the consuming instruction got the operand. Thus, an operand with many consumers could be a completed, timely, and cached operand for each different instruction.

5 DRA Implementation

In our proposed architecture, operands are delivered to the functional units in one of 4 ways.

- Pre-read from the register file: If the operand(s) exists in the register file at decode time, (i.e. a completed operand), it is pre-read from the register file and sent to the IQ.

- Read from the forwarding logic: The base pipeline has 9 stages of forwarding logic to handle requests for timely operands that were produced in the previous 9 cycles.

- Read from the CRC: CRC lookup happens in parallel with the lookup in the forwarding buffer. There is a 16 entry CRC in each of the eight functional unit clusters that provides cached operands.

- Read from register file on an operand miss: If, during execution, the operand is not available through any of the means above, then the operand misses. A miss signal is sent to the register file. The operand is read and delivered to the IQ payload where it waits for the instruction to reissue. This is the recovery path resulting from a mis-speculation on the operand resolution loop.

The hardware for this scheme is shown in Figure 7, and consists of a *register pre-read filtering table* (RPFT) combined with one *cluster register cache* (CRC) and one *insertion table* for each functional-unit cluster. This is in addition to structures that already exist in our base model, specifically one forwarding buffer per cluster, and a monolithic register file.

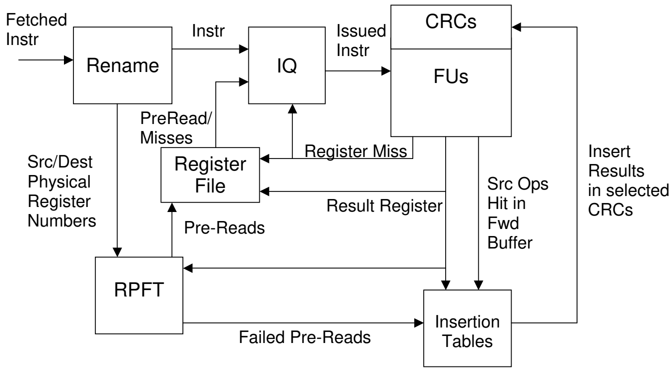

Figure 7. Distributed register algorithm (DRA) block diagram.

5.1 Cluster Register Cache (CRC)

There are 8 functional unit clusters in our base architecture, and there is a CRC associated with each cluster. Each CRC is placed close to it's functional unit cluster to reduce wire delays. Our studies show that a 16 entry CRC is more than adequate to meet our needs. Since only one instruction per cycle executes in a functional unit cluster, only 2 read ports are required per CRC. However, 8 write ports are needed to handle the maximum number of register values computed per cycle. The CRC, similar to the forwarding buffer, uses a fully associative lookup requiring the use of a CAM structure.

The CRCs use a simple FIFO mechanism to manage insertion and removal of entries. A more complex mechanism would be cumbersome and unnecessary because most register values are only read once before being overwritten [6]. We modeled a few mechanisms that had almost perfect knowledge of which values were needed, but the performance improvement over our simple FIFO scheme was negligible. Furthermore, register cache capacity pressure is reduced by filtering the operands that get inserted in the CRC.

Register cache insertions are filtered in one of two ways. First, only the CRC associated with the functional unit cluster an instruction will execute on receives the input register operands for that instruction. Our base model uses clustered issuing logic similar to the Alpha 21264, and instructions are slotted to a particular functional unit cluster (or arbiter according to the 21264 nomenclature) at the time of decode. Therefore, it's known at decode which source operands are required in each functional unit cluster. Second, only those operands that have consumers that have not read the operand when it leaves the forwarding buffer get inserted into the CRC's. This means that if all consumers of an operand either pre-read it from the register file or receive the value from the forwarding buffer, then the value is not stored in any CRCs.

The base architecture already contains the logic to determine the functional unit cluster an instruction will execute on. However, the second filtering mechanism, determining whether the operand is procured from the register file or the forwarding buffer, requires additional hardware. Two new structures, the *register pre-read filtering table (RPFT)* and the *insertion table*, address these issues.

5.2 Register Pre-read Filtering Table (RPFT)

The RPFT stores information about the validity of the registers. It has one bit associated with each physical register. When the bit is set, it indicates that the register is valid in the register file. The operand stored in that register is a completed operand and can be pre-read prior to issue. The bit is set when an operand is written back to the register file. If the bit is clear, the producer of that operand is in flight, and the operand is not in the register file. The bit is cleared when the renamer notifies the RPFT that it has allocated a physical register to be written by an instruction.

After the register renaming stage, the physical register numbers for an instruction's source operands are sent to the RPFT. If the bit for a register is set, then the value in the register file is pre-read and forwarded to the payload portion of the IQ. If the bit is clear, the source register number for the input operand is sent to the insertion table associated with the functional unit cluster the instruction is slotted to.

The number of 1-bit entries in the RPFT equals the number of physical registers in the machine. The structure requires 16 read ports, and 8 write ports to handle 8-wide issue. Weiss and Smith used a similar algorithm to work around stalling instructions when they saw a true dependency [13]. In their algorithm, a bit set in the scoreboard indicated a true dependency on an un-computed result, and the dependent instruction was placed in the reservation stations along with the register identifiers for the un-computed operands.

5.3 Insertion Table

There is an insertion table associated with each CRC and functional unit cluster. It keeps count of the number of outstanding consumers of an operand that will execute on the functional unit cluster and that have not yet read the operand. The number of entries in an insertion table is dictated by the number of physical registers. Each entry is 2 bits wide. A non-zero entry value indicates that the operand is needed by instructions assigned to the insertion table's functional unit cluster. An entry is incremented when the insertion table receives the source register number from the RPFT, and it is decremented every time the associated register is read from the forwarding buffer.

With 2 bits per entry, the insertion table entries can indicate a maximum of 3 consumers for each operand per cluster. However, most operands have few consumers, so 2 bits is more than adequate.

When an operand is written back (from the forwarding buffer) to the register file, a copy is also sent to each of the insertion tables. If the insertion table entry associated with the operand is zero, it is highly likely that there are not any consumers of this operand in-flight and the value is discarded. For all functional unit clusters where the insertion table entry for an operand is non-zero, there are consumers in flight. The operand is written into the CRCs for those functional units and the insertion table entries are cleared. Note that operands can reside in multiple functional unit clusters as long as there are outstanding consumers for that operand that will execute on each of those clusters.

5.4 Misses

Mis-speculations occur on the operand resolution loop because the DRA, as implemented, does not guarantee a successful pre-read or a hit in the forwarding buffer or CRCs. Misses happen for one of two reasons. Operands may get dropped from the CRCs before being read due to capacity pressure and the FIFO replacement policy. Operands may also not get inserted into the CRCs because we saturate at 3 consumers per operand. This occurs when an operand has more than 3 consumers slotted to the same functional unit. For each operand hit in the forwarding buffer, the count for that operand in the insertion tables gets decremented by one. If there are at least 3 hits in the forwarding buffer on a single operand, then the count in the functional unit's insertion table goes to zero and indicates no consumers are in-flight that need this operand. Thus, the operand does not get inserted in the CRC, and any subsequent consumers executing on the same functional unit take an operand miss.

If one (or both) of an instruction's operands miss in the CRC or forwarding buffer, an invalid input is returned in place of the real value, and the instruction produces an invalid operand. When this happens, signals are sent to both the register file and the IQ. The correct input operand value is read from the register file and sent to the IQ, and the IQ readies the instruction for reissue. The instruction is ready to reissue as soon as the operand reaches the IQ payload. In addition to reissuing the instruction with a missing operand, all instructions in the dependency tree that have already issued will also signal the need to reissue as soon as they read the invalid operand resulting from a miss in the CRC. The logic to manage mis-speculations on the operand resolution loop is similar to the logic that manages mis-speculations on the load resolution loop. The only additional hardware required is the wiring to stall the front end of the pipeline while the missing operands are read from the register file and forwarded to the instruction payload.

5.5 Stale Register Values

The CRC associated with each functional unit is implemented as a simple FIFO structure to avoid the problems associated with managing a complex insertion and replacement algorithm. As a result, stale data needs to be accounted for in the CRC in order to guarantee correctness. Although rare, a CRC could have stale operands if there is not much pressure on the structure. A physical register may be reallocated while the old register value resides in the CRC.

This case is handled when the register is reallocated. The destination register numbers are sent to the RPFT, and these are also forwarded to all CRCs. If the CRC contains an operand for a reallocated register, then that entry is invalidated. Note that there are many cycles between when the CRC receives the reallocated registers and when the register is written with a new value. Therefore, we have enough time to invalidate the entries in the CRC. An alternate method would time out the operands in each CRC after a certain period of time.

6 Results

The basic premise behind the DRA is that we remove the expensive register file access from the IQ-EX stage and overlap it with part of the DEC-IQ stage. By doing so, we remove latency from a critical portion of the pipeline and possibly increase the latency in other portions of the pipeline.

In our base processor model, moving the register file access reduces the IQ-EX latency from 5 to 3 cycles while the DEC-IQ latency remains the same. Register file lookup takes 3 of the 5 IQ-EX cycles. However, one of these cycles is still required for accessing the forwarding buffer and the CRCs, resulting in a 3 cycle IQ-EX latency. In the DEC-IQ portion, the physical register numbers are available at the end of the second cycle, providing 3 cycles for accessing the register file and sending the data to the IQ payload. Given that accessing the register file and driving data to the functional units takes 3 cycles in the base machine, we should be able to drive data to the payload in the same time. Hence, the DEC-IQ portion remains 5 cycles.

We also ran experiments with longer register file access latencies of 5 and 7 cycles to determine the impact the DRA has on potential future designs. In the case of a 5 cycle access latency, the base architecture's IQ-EX latency is 7 cycles. The DRA implementation removes the 5 cycle register file access latency, but needs 1 of these cycles to access the forwarding buffer and CRCs. Thus, it shrinks the IQ-EX stage to 3 cycles. The DEC-IQ latency increases by 2 to a total of 7 cycles. This is because the register renaming is complete after the 2nd cycle of DEC-IQ, and it still takes 5 cycles to access the register file and deliver the operands to the IQ. For the 7 cycle register read latency, the IQ-EX stage remains at 3 cycles and the DEC-IQ stage increases to 9 cycles.

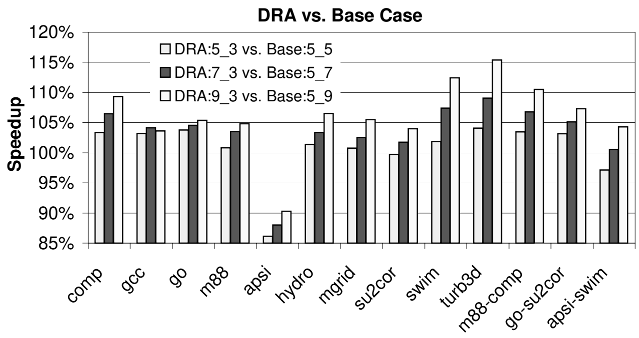

Figure 8. Performance improvements with the DRA relative to the base architecture. The **DRA:D1_D2 vs. Base:B1_B2** syntax shows the pipeline latencies for each configuration. D1 and D2 are the latencies from DEC-IQ and IQ-EX, respectively, for the DRA. B1 and B2 represent the same latencies for the base configuration. Both configurations have the same register file access latency. Note the graph starts at 85%.

We modeled the architecture described using the ASIM [4] simulation infrastructure with a very detailed, cycle level, execution driven processor model. ASIM forces consideration of logic delays by mimicking hardware restrictions within the processor model. This makes it very difficult to model instantaneous, global knowledge over the entire model. In hardware, for example, there is a non-unit delay between the IQ and the functional units. Therefore, there is a lag between the time events occur in the functional units and the time the IQ makes decisions based upon these events. ASIM enforces propagation delay restrictions in the simulated model, and does not allow us to make decisions based upon global knowledge that may lead to inaccuracies in our simulations.

Figure 8 shows the results using the DRA for the three different register access latencies. Performance is shown as speedup of the DRA implementation relative to a non-DRA implementation. For example, the first bar, **DRA:5_3 vs Base:5_5**, shows the relative speedup of a DRA implementation with a 5 cycle DEC-IQ latency and a 3 cycle IQ-EX latency relative to a base pipeline with no DRA and a 5 cycle latency for both DEC-IQ and IQ-EX. Both configurations have a 3 cycle register file access latency. The second and third bar in each cluster shows similar information for a 5 and 7 cycle register file access latency, respectively.

With the exception of *apsi* and *apsi-swim*, performance improves with a DRA for all configurations. We see an improvement of up to 4%, 9% and 15% for register file access latencies of 3, 5, and 7 cycles, respectively. Performance improves not only because we shift the cycles from IQ-EX to DEC-IQ, but because we also shorten the pipeline by 2 cycles in each case. Those programs that are the most sensitive to pipeline lengths (*compress, m88ksim-compress*) and/or IQ-EX latencies (*swim, turb3d*) benefit the most.

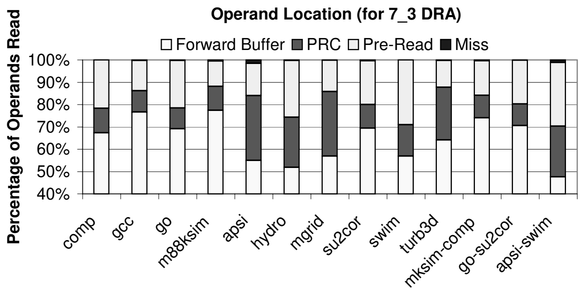

Figure 9. Hit and miss rates for operand values. Hits are further segmented into hits from register pre-read, hits from the forwarding buffer, and hits from the DRA. Numbers are shown for the 7_3 DRA case, i.e., 7 cycles from DEC_IQ, 3 cycles from IQ_EX, and a 5 cycle register file access latency. Note the graph starts at 40%.

The reason performance degrades is because of misspeculations on the operand resolution loop. Not only will the instruction that suffered an operand miss reissue, but all of the instructions in the dependency tree that have issued will also reissue. Figure 9 shows that performance is very sensitive to operand miss rate. The figure shows the hit and miss rates for register operand values. Hits are further segmented into hits in the register file during pre-reading, hits in the forwarding buffer, and hits in the CRCs. On average, more than half the operands are read from the forwarding buffer. The remaining operand reads are distributed equally between being pre-read from the register file, and read from the CRCs. Most benchmarks have an operand miss rate well under 1%, and do not suffer a performance impact from operand resolution loop mis-speculations. However, even a small miss rate of 1.5% can have a substantial impact on performance, as *apsi* shows. In this case, the work wasted due to mis-speculations on the operand resolution loop outweighs any benefit resulting from a shorter pipeline.

There are two reasons why *apsi* suffers a 10%-14% performance loss. First, there is the relatively high miss rate of 1.5%. This, combined with a high frequency of loop occurrence, results in a large number of reissued instructions and much wasted work. Second, *apsi* is not particularly sensitive to pipeline lengths as shown in Figure 4. A 12 cycle increase in pipeline length only degraded performance by 9%. Therefore we gain little by shortening the pipeline, and suffer the penalty of high operand miss rates and instruction reissue. The combination of the two situations contributes to the performance loss in *apsi*.

7 Related Work

Hierarchical register files are not a new idea. The Cray-1 had two sets of two-level register files. More recently, Zalamea et. al. explored two-level hierarchical register file designs[17]. However, in both cases, compiler support was required to explicitly move values between different levels of the register file.

Cruz et. al. proposed a hierarchical register file design that does not require compiler support [6]. They use a single register file with a highly ported upper-level portion that acts as a register cache, and a lightly ported lower-level that acts as a register file. The design proposed by Cruz has a number of shortcomings for our architecture. First of all, they use mechanisms to manage the entries in their register cache that require current information from the instruction scheduling unit. However, given the latencies in our pipeline, it is impossible to gather this knowledge and act on it in a timely manner. Another problem with the design is the non-deterministic delay for instruction execution that depends on whether the operands are attained from the register cache or register file. Due to the non-deterministic delay, the dependents of an instruction cannot be scheduled with certainty. If an instruction's dependents are issued with the assumption that the instruction will "hit" in the register cache, then the dependent instructions must stall if the instruction ends up accessing the slower register file. Stalling instructions which have been issued entails complex control which can add to the critical path of the processor [9]. Finally, the lower-level register file design has fewer ports than the number of functional units. Hence, there is no mechanism to handle the case where all instructions issued miss in the register cache. Even though this is an unlikely event, it must be accounted for.

8 Summary and Conclusions

In this paper, we explored micro-architectural loops resulting from hazards in the pipeline. In particular, we focused on a subset of micro-architectural loops, called loose loops, that impact processor performance by forcing the pipeline to stall or speculate until the loop resolves. We showed that the performance impact of loose loops is related to the pipeline length and configuration. In particular, performance is especially sensitive to the length of the issue to execute section of the pipeline due to the load resolution loop. Reducing the latency of issue to execute improves performance even as the overall length of the pipeline remains the same.

Based on our analysis, we proposed the the DRA as a way of reducing the issue to execute latency. The DRA moves the time consuming register file access out of the issue to execute path and replaces it with the clustered register cache (CRC). Using a very detailed architectural simulator, we showed performance improvements of up to 4% to 15%, depending on the pipeline configuration, with the DRA.

Much of our future work focuses on improving the design of the DRA. For example, retaining pre-read operands in the instruction payload requires a large amount of hardware. Therefore, a more efficient design might be to forward the pre-read values to each cluster to be held in another register cache close to the functional units. In addition, we'd like to investigate a more efficient method of invalidating stale entries in the CRCs. Also, further analysis of benchmarks like *apsi* needs to be done to determine how we can reduce or eliminate the performance loss.

Acknowledgments

The authors would like to acknowledge the following individuals for their contribution to this work: Matt Reilly for his insightful comments and corrections, Chaun-Hua Chang for clarifying the register file design, and Dean Tullsen for reviewing the paper. We would also like to thank the reviewers for their comments and suggestions.

References

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, Vancouver, British Columbia, June 12–14, 2000. IEEE Computer Society and ACM SIGARCH.

[2] R. I. Bahar and S. Manne. Power and energy savings via pipeline balancing. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Gotenburg, Sweden, July 2001. IEEE Computer Society and ACM SIGARCH.

[3] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.

[4] Compaq Computer Corporation. *The ASIM Manual*, August 2000.

[5] Compaq Computer Corporation. *Alpha 21464 Internal Design Specification, Rev. 0*, 2001.

[6] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 316–325, Vancouver, British Columbia, June 12–14, 2000. IEEE Computer Society and ACM SIGARCH.

[7] Digital Equipment Corporation. *Alpha 21164 Microprocessor Hardware Reference Manual*, 1994.

[8] Digital Equipment Corporation. *DECchip 21064 and DECchip 21064A Alpha AXP Microprocessors Hardware Reference Manual*, 1994.

[9] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizaabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, Anil K. Jain, Shekhar Mehta, Jeanne E. Meyer, Ronald P. Preston, Vidya Rajagopalan, Chandrasekhara Somanathan, Scott A. Taylor, and Gilbert M. Wolrich. The internal organization of the Alpha 21164, a 300-mhz 64-bit quad-issue CMOS RISC microprocessor. *Digital Technical Journal*, 7(1):119–135, 1995.

[10] Joel Emer. Ev8: the post-ultimate alpha. Keynote at International Conference on Parallel Architecture and Compilation Techniques, September 2001.

[11] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. Register file design considerations in dynamically scheduled processors. In *Proceedings of the Second International Symposium on High Performance Computer Architecture*. IEEE, January 1996.

[12] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. *The Microarchitecture of the Pentium4 Processor*. Intel Corporation, February 2001.

[13] M. Johnson. In *Superscalar Microprocessor Design*, 1991.

[14] R. E. Kessler, E. J. McLellan, and D. A. Webb. The alpha 21264 microprocessor architecture. In *Proceedings of the International Conference on Computer Design*, October 1998.

[15] M. Reilly. Lost cycles due to register port contention. In *htttp://segsrv.shr.cpqcorp.net/arana/qbox/register_cache2.html*. Compaq Computer Corporation, February 1998.

[16] J. S. Seng, D. M. Tullsen, and G. Z. N. Cai. Power-sensitive multithreaded architecture. In *Proceedings of the International Conference on Computer Design*, October 2000.

[17] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Two-level hierarchical register file organization for VLIW processors. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (Micro-33)*, pages 137–146, Los Alamitos, CA, December 10–13 2000. IEEE Computer Society.