LaZy Superscalar [*]

Görkem Aşılıoğlu Zhaoxiang Jin Murat Köksal Omkar Javeri Soner Önder
Department of Computer Science
Michigan Technological University
{galolu,zjin3,mmkoksal,oujaveri,soner}@mtu.edu

Abstract

LaZy Superscalar is a processor architecture which delays the execution of fetched instructions until their results are needed by other instructions. This approach eliminates dead instructions and provides the necessary means to fuse dependent instructions across multiple control dependencies by explicitly tracking control and data dependencies through a matrix based scheduler. We present this novel redesign of scheduling, recovery and commit mechanisms and evaluate the performance of the proposed architecture. Our simulations using Spec 2006 benchmark suite indicate that LaZy Superscalar can achieve significant speed-ups while providing respectable power savings compared to a conventional superscalar processor.

1. Introduction

In contrast to deeper pipelining approaches of a decade or more ago, recent superscalar processors emphasize shallow pipelines and modest clock speeds for their implementation technology. Shallow pipelines can extract more instruction-level parallelism (ILP) from instruction streams which have low ILP and provide better power and energy profiles, which makes them more suitable building blocks for multi-core processor chips. On the other hand, shorter pipelines result in unbalanced pipelines where the execution step is almost never on the critical path in the pipeline, providing an opportunity to *fuse* dependent instructions and execute them in the same cycle by using cascaded ALUs [16]. We refer to the technique of combining dependent instructions in this manner as *instruction fusing*.

Instruction fusing can be very effective both with low and high ILP code, as it may reduce the critical path through the dependence graph of the program. The reduction in dependence height increases the amount of *available parallelism* at a given look-ahead distance on the instruction stream and hence promotes a different design goal: instead of buffering as many instructions as possible and aggressively executing independent instructions, we aim to combine as many dependent instructions as possible to improve available parallelism.

Instruction fusing can be implemented as a pure compiler technique through the use of a redesigned instruction set architecture (ISA), or as a pure hardware technique without modifying the ISA. The main challenge for both techniques is to identify those instructions which can be fused together in a profitable way. Fusing instructions which have a large *slack* [5] will not improve available parallelism. Therefore, just as aggressive techniques need a large pool of instructions to choose from, fusing needs a large pool of instructions from which fusible instructions are formed. Although compiler techniques have good look-ahead, control-flow uncertainty causes overhead instructions to be executed or opportunities to be missed. Analogous to *distant ILP*, fusing instructions across multiple branches by a compiler cannot be performed without speculative code-motion and may actually hurt performance. Similarly, existing micro-architecture techniques have a limited look-ahead, since they require instructions to be simultaneously available for inspection together to be fused.

In this paper, we define a *fusible instruction* as an ALU instruction that is not a branch with no more than two operands and a single destination. This condition disqualifies store, load and branch instructions from fusion as well as multi-destination operations present in MIPS, such as divide and multiply. When we study the fusible instructions, we find that for most benchmarks majority of fusible instructions come from different basic blocks. This behaviour is shown in Figure 1. This figure illustrates that simple mechanisms which only harvest the opportunities available within a single fetch group may miss many opportunities for a large set of benchmarks, motivating us to develop a generalized technique that is insensitive to the presence of control-flow when fusing instructions.

LaZy Superscalar[1] is a novel micro-architecture which does not need to have a large pool of instructions for simultaneous inspection and fusing. Instead, it relies on demand-driven

[*] This work is supported in part by NSF grants CCF-1116551 and CCF-1450062.

[1] In reference to *lazy evaluation* and the phrase “Much of my work has come from being lazy” by John Backus as laziness in this respect improves performance. The capital Z comes from the pronunciation of *fuse*, i.e., /fju:z/.

ISCA'15, June 13-17, 2015, Portland, OR USA
Copyright 2015 ACM 978-1-4503-3402-0/15/06$15.00
http://dx.doi.org/10.1145/2749469.2750409

execution such that the execution of instructions are delayed until they are either demanded by another instruction, or, it is proven the instruction is dead and it is eliminated. With this approach, producer instructions delay their execution until their consumers are encountered, in essence providing unlimited look-ahead for fusing. As a result, distant instructions, possibly separated by multiple branches from each other can be effectively fused together. Delaying the execution also provides automatic detection of partially dead code. Any output dependence on a delayed instruction's destination register means the instruction is dead. In other words, if an instruction is fusible it is either eventually fused to another, executed without being fused due to a demand from a non-fusible instruction or is simply killed.

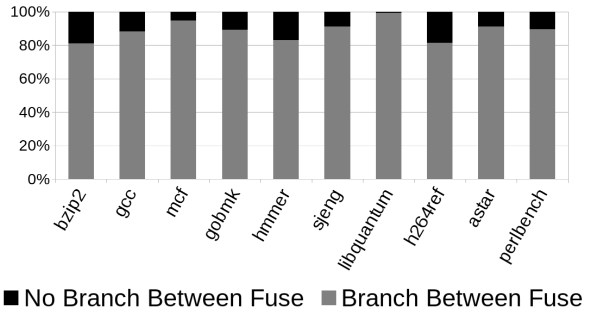
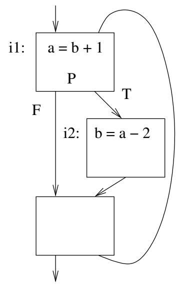


Figure 1: Origin of fused instructions

Although this policy would be detrimental to performance without fusing, with fusing it will deliver at least the same level of performance as the aggressive execution policy, subject to resource constraints. We illustrate through a simple example shown in Figure 2.



Cycle	P	Superscalar	LaZy Superscalar
1	T	$i1_1$	—
2	T	$i2_2$	$i1_1 + i2_2$
3	F	$i1_3$	—
4	F	$i1_4$	Eliminate $i1_3$
5	F	$i1_5$	Eliminate $i1_4$
6	T	$i2_6$	$i1_5 + i2_6$

Figure 2: Partially dead code

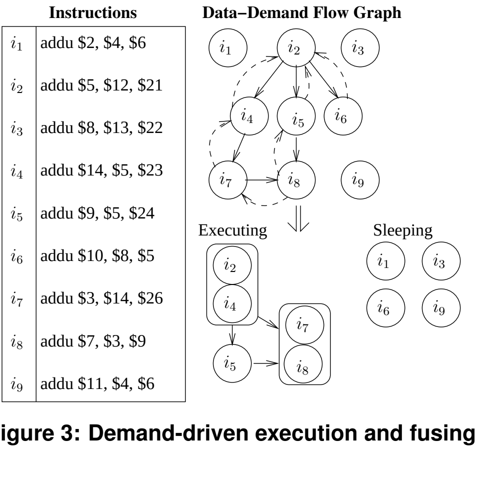
For the simplicity of illustration, we ignore the execution of branches except that we assume they separate fetch groups. We also ignore the pipeline start-up times. Subscripts under the instruction names indicate the cycle number at which the instruction was fetched. During 6 cycles of execution, the superscalar processor yields an IPC of one but executes only four useful instructions. During the same period, LaZy executes four useful instructions and squashes two, providing essentially the same performance. This is not a coincidence. If a useful instruction is delayed, LaZy can make-up for the missed execution opportunity by executing the dependent instruction together with the producer. If a dead instruction is delayed, the missed execution opportunity cannot contribute to the overall execution. The real power of delayed execution surfaces when there is a sufficient number of fetched and fused instructions buffered for execution. This permits the LaZy processor to consume dependent chains at a rate of two instructions per cycle at each execution unit, yielding a higher IPC than the aggressive superscalar. Furthermore, since killed instructions do not compete for execution units, their presence increases the execution bandwidth of LaZy processor compared to the superscalar processor. For example, if a sufficient number of ready instructions are available, LaZy processor could execute those instructions in cycle 3, 4, and 5 of our motivating example, whereas the superscalar schedule would be extended by three cycles to arrive at the same point in execution. In this case, the LaZy processor would have achieved an IPC of roughly 1.5 using a single execution unit, and the superscalar would remain at one IPC. Although the amount of dead instructions in highly optimized code is quite low, they are automatically eliminated as a by-product of our fusion algorithm and their elimination is still beneficial. Our studies show that less than 1 % of dynamic instructions in Spec2006 integer benchmarks are dead instructions. We demonstrate that there are many opportunities for fusing, provided we have the capability to delay the execution of producer instructions until their consumers are encountered.

In the remainder of paper, we first discuss the concept of demand-driven execution in a superscalar processor, the basic execution paradigm for LaZy in Section 2 and how various dependencies can be appropriately represented under this model. We then discuss two main issues in implementing this model. The first is how to handle the processor state in a machine where the notion of *program order* is blurred. Most existing mechanisms for handling the processor state rely on a sequential notion of state and will not work properly under these conditions. For example, what should happen when an instruction which was not demanded yet arrives at the head of the reorder buffer (ROB)? Since it was not yet demanded, it cannot execute. Consequently, it cannot retire since it has not yet executed. If the instruction is never demanded before all resources are consumed, the machine would deadlock. Our solution is to separate the concepts of *committing* and *retiring* an instruction and to permit a delayed instruction at the head of ROB to *commit to the state* and leave the ROB, but not *retire* from the processor. In other words, we know that the result of the instruction would be the correct value for that register although it has not yet been computed. We discuss these issues in Section 3.

The second challenge is designing a wake-up/select mechanism which takes into account whether an instruction is demanded or not. As it is well known, wake-up/select logic is almost always on the critical path in contemporary superscalar processors and incorporating another piece of data on this logic is challenging. Our solution is to employ a *matrix scheduler* which can track the demand status as well as operand availability of enqueued instructions. We discuss the layout and operation of our scheduler in Section 4. We put together the concepts and present the operation of the LaZy pipeline and its individual stages in Section 5. We follow with a possible circuit-level implementation and evaluation of the matrix scheduler in Section 6. Our experimental evaluation of the processor design conducted at the cycle level presented in Section 7 compares the performance of the architecture to that of a typical superscalar architecture. We conclude by discussing related work in Section 8 and summarize our contributions in Section 9.

2. Demand-driven Execution

Demand-driven execution schedules instructions by using *demand signals* which are sent from consumer instructions to their producers, as opposed to conventional dataflow style wake-up/select mechanisms which send wake-up signals from producers to consumers. As we have discussed in the introduction, demand-driven execution enables us to identify which instruction results are needed immediately, whether a given instruction is dead or not, and finally, fuse the dependent instruction with its producer when that is feasible. This execution approach reduces the actual dependence height of the dataflow graph of the program, and hence increases the available parallelism.



	Instructions
i_1	addu \$2, \$4, \$6
i_2	addu \$5, \$12, \$21
i_3	addu \$8, \$13, \$22
i_4	addu \$14, \$5, \$23
i_5	addu \$9, \$5, \$24
i_6	addu \$10, \$8, \$5
i_7	addu \$3, \$14, \$26
i_8	addu \$7, \$3, \$9
i_9	addu \$11, \$4, \$6

Figure 3: Demand-driven execution and fusing

Consider the code sequence and the combined data-demand flow graph shown in Figure 3. Assuming a 3-wide fetch unit, the execution of this code sequence unfolds on LaZy Superscalar as follows. In cycle 0, i_1,i_2 and i_3 are fetched. All three instructions are buffered as none has been demanded yet. Fetching of i_4, i_5 and i_6 causes three demand signals to be placed to i_2. Instruction i_4 fuses with i_2 and the pair starts executing in cycle 1. Instructions i_5 and i_6 sleep as they are not yet demanded. Fetching of i_7, i_8, and i_9 causes demand signals to be sent to i_4, i_5 and i_7, which causes fusing of i_7 and i_8 and waking-up of i_5. As a result, instructions i_5 and the fused pair i_7, i_8 start executing, yielding the schedule shown at the right-bottom of the figure.

Demand-driven execution in this manner can uncover the critical path through the program and can reduce the dependence height of programs by fusing dependent instructions. For this example, the sequence $i_2 \rightsquigarrow i_4 \rightsquigarrow i_7 \rightsquigarrow i8$ make up the critical path, yielding a dependence height of four instructions. Due to instruction fusing and demand-driven execution the dependence height of the code is now reduced to three. Clearly, demand-driven execution in this manner can be extended so that an instruction is scheduled for execution only when its child is demanded, or even when the grand-children are demanded. If we define an architectural parameter, *wake-up depth* in reference to the length of a dependent chain of instructions to be buffered, the paradigm represents a spectrum of architectures, each becoming increasingly LaZy. Note, for the purpose of efficiently pipelining the wake-up logic, Stark et. al., used the paternal relationship among instructions [30].

For example, if the wake-up depth of the architecture is zero, no instructions are buffered and every fetched instruction is considered to be demanded and is immediately scheduled for execution, corresponding to a conventional processor. A wake-up depth of one implies that a fetched instruction must be buffered in the wake-up window until the time another instruction is fetched which references its result and demands it. In other words, the instruction is woken-up by its children. This is the architecture we focus on in this paper. As the wake-up depth of the architecture increases, there will be additional penalty of not executing an instruction when there is an opportunity to do so. On the other hand, the scheduler will converge rapidly towards the critical path through the program and the approach can provide significant energy savings. Note that some earlier designs such as the dependence based micro-architecture design proposed by Palacharla et. al. [22], direct wake-up architecture proposed by Önder and Gupta [20] can be viewed as wake-up depth zero (i.e., no buffering) demand-driven (dependence based) architectures, among others.

Dependencies among instructions are not limited to dependencies through register names. Out-of-order processors track the dependencies among instructions both implicitly and explicitly through the use of several hardware structures. Data dependencies through instruction-set architecture (ISA) registers are translated to physical register names and output and anti data dependencies are removed by renaming the instruction stream. The dependencies through memory typically are represented implicitly by forcing the memory operations into first-in-first-out (FIFO) structures such as load/store queues. Control dependencies are also represented implicitly by following the program order through another FIFO structure, *reorder buffer*, and carrying out branch resolution in program order.

LaZy moves the heavy lifting of dependency checking to the front end of the pipeline by mapping all control, register and memory data dependencies onto register names. At the rename phase, all instructions are assigned a register name, although names assigned to branch and memory instructions do not need to have a corresponding full 32-bit physical register. This approach allows us to represent all dependencies in a unified manner and permits dependence-driven execution of the entire execution sequence. We therefore equip every instruction with a destination register name. For this purpose, the renamer is given four distinct register name pools. The first pool has a one-to-one correspondence with physical registers. The second pool and the third pool are used to "rename" store and load instructions respectively and do not have corresponding physical registers. Finally, the fourth pool is used to "rename" branch instructions which also do not have corresponding physical registers. This way, dependences (either predicted, or actual) among all instructions can be represented properly. We illustrate through a simple example shown in Figure 4. The example shows how a total memory order can easily be implemented using the outlined mechanism. This total order requires all prior stores to issue before later loads or stores can commence.

Code Sequence	Renamed Sequence
lw r1, ...	p_0 = lw $[\perp]$
sw ...,...	s_1 = sw $[p_0]$
lw r2, ...	p_1 = lw $[s_1]$
lw r3, ...	p_2 = lw $[s_1]$
sw ...,...	s_2 = sw $[p_1, p_2]$

Figure 4: Load-Store ordering through renaming

In our example, the first instruction has no prior load/store dependencies, so its additional operand shown in brackets is $\perp$. This instruction's logical destination is renamed to physical register p_0. The second store is made dependent on p_0, and its destination is renamed to s_1 when it is issued. The next two load instructions are dependent on s_1. Finally, the last store instruction has both p_1 and p_2 in its list and is dependent on both of them. This mechanism can be used to implement virtually any memory ordering scheme, including any of the dependence speculation mechanisms such as the store set algorithm [3]. In such a case, the renamer would use the dependences predicted by the algorithm to produce the necessary dependence names. For example, if all loads and stores were known to be independent to the store set algorithm, none of them would have source operand dependencies to memory instruction predicates. Alternatively, if the second load and the first store has collided in the past, the second load would be made dependent on s_1. In this mechanism, each destination register (which may or may not correspond to a physical register) is simply allocated from the corresponding pool and returned to the appropriate pool upon completion. In our evaluations discussed later we model a simple scheduling mechanism which implements a traditional memory renaming mechanism using a store queue. As a result, we permit store instructions to issue as soon as their data operands are ready by marking them as demanded at decode time and permit loads to issue when all prior stores have issued. As well, before a store instruction commits to memory, it is checked to see if all prior memory operations have completed. Thanks to its predicate based dependence checking, LaZy Superscalar does not need a *load queue*, but a store queue is still employed and is operated in an identical manner to a typical superscalar processor. While we do not discuss memory consistency models in this work, LaZy dependence mechanism can successfully implement any consistency model implementable by conventional superscalar processors by altering how the dependencies are mapped among loads and stores through predicate names and when those predicates are enforced.

Representation of dependencies in this manner allows us to unify the entire dependence checking into a single structure, namely, the dependence matrix. Branch instructions are handled similarly; they are allocated names from a different pool and all the following instructions become control dependent on that branch. This unified dependency checking mechanism also relies on the concept of fine-grain processor state management, the topic of next section.

3. Fine-grain Processor State Handling

In our discussion about how the processor state is managed, we follow the conventions of *in-order state*, *speculative state* and *architectural state* [12]. As discussed previously, buffering of instructions until they are demanded poses special challenges to state representation and recovery algorithms. This is because when a misspeculation is detected we need to know precisely what constitutes the in-order state so that a recovery can be performed and the execution is resumed. Consider the motivating example shown in Figure 2. Suppose the first instruction was fetched and is buffered. The branch has been incorrectly predicted to be not taken, the loop executed again and the sequence $i_1 \rightsquigarrow i_2$ executed. At this point in time the first branch resolved and misprediction is detected. What is the in-order state and how can we resume the execution with i_2 again? In connection with the above problem, it should be clear that a conventional recovery mechanism cannot tell when it is safe to retire instructions. The solution to this problem is to realize that instead of following the lump-sum approach of representing the state, a fine-grain approach is necessary. Clearly, at the point of misprediction, the first instance of i_1 is part of the in-order state, even though it has not executed yet. Our solution therefore is to employ a reorder buffer, but treat the instruction flow through it differently.

LaZy incorporates a reorder buffer into which identifiers (in our case allocated rename registers) of fetched instructions are fed as the instructions are fetched in program order. The reorder buffer logic checks to see if the instruction at the head of the reorder buffer is a branch. If the instruction is a branch, its completion is awaited as it would happen in a conventional superscalar. If the branch happens to be mispredicted, the processor recovers its in-order state and continues fetching instructions from the correct path. If the instruction is not a branch, then the corresponding instruction is marked to be in in-order state (i.e. "committed"), but it is not retired as it might not have even executed. If the instruction at the head of the reorder buffer is a store instruction, the store queue is signalled to commit the value to the memory.

This simple mechanism individually identifies whether each instruction is part of the in-order state or not and commits the values to the memory as the values represent the in-order state for the memory data when a store instruction reaches at the head of the reorder buffer. This mechanism of separating the commit and retire actions from detecting what constitutes the in-order state allows us to perform commits separately and keep instructions indefinitely until they are either demanded by another instruction or proven to be dead.

4. LaZy Matrix Scheduler

In order to realize lazy scheduling we use a dependence matrix. Using a matrix based scheduler for conventional superscalar processors has been explored before based on the inventions by Zaidi [31] and Henstrom [9]. Our approach however differs significantly. These techniques are designed primarily for scheduling instructions, whereas our approach combines both demand and data signalling and the entire retire process is driven by the matrix based design.

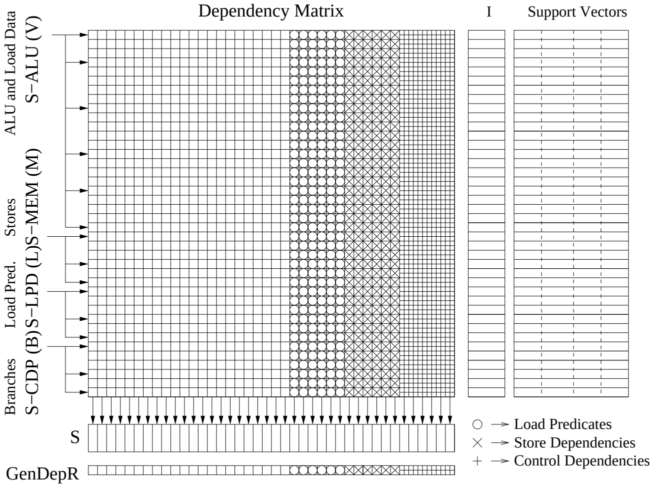


Figure 5: Demand Driven Matrix Scheduler

The dependency matrix (DEPMAT) shown in Figure 5 is a single bit matrix where each line in the matrix represents the dependencies of a single instruction. A set bit at a column c at any line means that instruction represented by that line has a dependency on the instruction at line c. This could be a data dependency, memory dependency or a control dependency. The matrix is divided into four sections. The first section (S-ALU) is reserved for instructions which need a physical destination register such as arithmetic/logic and load instructions and each row of the section corresponds one-to-one to a physical register. The second section (S-MEM) is used for memory dependencies and provides one row per in-flight store instruction. The third section (S-LPD) is used for load predicates. A load predicate is assigned to each load instruction to represent its memory dependencies. Finally, the fourth section (S-CDP) is used to track control dependencies and provides one row per in-flight branch instruction. S-MEM, S-LPD, S-CDP do not provide physical registers. Assuming the number of physical registers is V, number of store queue entries is S, number of load predicates is L and the number of in-flight branches is given by B, the dependence matrix will have $T = V + S + L + B$ rows and columns, i.e., TxT.

Each line in the matrix represents an instruction, either completed or waiting to be completed. An instruction occupying a matrix line implies a hold on a physical register. A matrix line is only released when the physical register corresponding to this line is released. The obvious exception is load predicate, branch and store sections in the matrix, which do not occupy physical registers. Branch and store lines are released as they are completed and confirmed. Load predicates are released when their partner load executes.

In the matrix, the OR result of a column c is true if another instruction is dependent on the instruction on line c. The result of a horizontal OR of the B entries tells us if the instruction at that line has any unresolved control dependencies. Similarly, the result of a horizontal OR of L or S entries yields if the instruction has any load or store dependencies.

The renaming subsystem follows Alpha 21264 style [13]. A simple vector of RAM holds instructions until they are retired. This vector is accompanied by several bit-vectors which provide support information. The support vectors contain the following data: (1) Line is free (FREE), (2) Instruction is complete (COMT), (3) Instruction is part of the in-order state (STATE), (4) Instruction can fuse (CFUSE), (5) Instruction is executing (INEX), (6) Line is valid (VALID), (7) Instruction data may still be used to obtain the current in-order state (INUSE), (8) Instruction has an exception (EXCP). There are two pointers per line as well. The instruction pointer (IPTR) contains the line number of a related instruction (i.e the second half of a two destination instruction or the predicate for a load). The fuse pointer (FUSEPTR) contains the line number of the second part of a fused pair.

LaZy also models additional dependencies in the matrix. A general dependency register (GenDepR) is maintained with all the persistent dependencies instructions should have. For instance, every instruction will be control dependent on all unresolved branches preceding them. GenDepR is a register of size T. The first V bits are reserved, the next S bits map to store dependencies, the next L bits map to load dependencies, and the final B bits map to control dependencies.

5. LaZy Superscalar Pipeline

LaZy Superscalar's pipeline shown in Figure 6 follows identical structures to conventional superscalar processors at the front and the rear of the pipeline. The renaming mechanism has been enhanced to rename all instructions. Since dependency checking is unified, the machine does not incorporate load queues. A store queue is provided for buffering the speculative values from store instructions until they can retire. The Commit phase of the pipeline commits instructions in program order irrespective of their completion status, as each instruction is flagged to belong to the in-order state. Instructions are retired later in an out-of-order manner as they are completed or squashed. In the following sections, we follow the pipeline flow and describe the operation of each stage in relation to registers and other storage that needs to be updated.

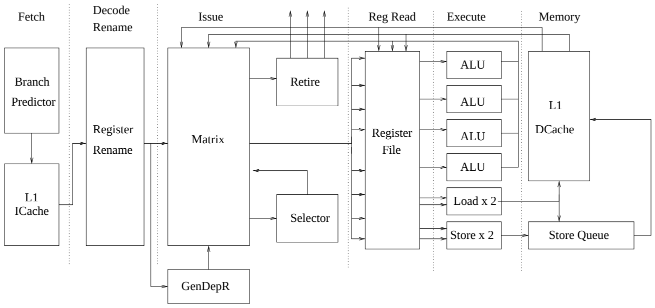


Figure 6: LaZy Superscalar pipeline

5.1. Instruction Fetch and Decode

Instruction fetch unit supplies the rest of the processor with a predicted instruction stream as in a conventional superscalar processor. Compound instructions are split into two separate instructions. These instructions include all load and store instructions as well as jump and link and jump to register instructions. Loads and stores are split into an addition and memory operation only if the immediate value in the original memory instruction is nonzero. The split instructions are then treated as separate instructions from then on in the pipeline. This also implies that on multi issue fetch units, each split instruction counts as two instructions fetched. For instance, an 8 issue fetch unit can only fetch 4 store word instructions if their immediate fields are non-zero.

5.2. Renaming

LaZy renames all instructions as described above. Therefore, there will be a stall if there is no free matrix line of the requisite type (value producing, load predicate, store or branch). One special case is a load instruction which requires a data result as well as a memory access result. Load instructions are assigned two registers: one physical register for storing their data and one load predicate to indicate the memory dependencies. Physical source registers are read through the front-end map table. After the instruction is renamed, a dependency vector of size T is prepared for each instruction to be inserted into the matrix. An instruction is dependent on its operands. This dependency vector is also ORed with the relevant parts of GenDepR based on the instruction type. Every instruction has a control dependency on all branches that precede it - therefore the last B bits are always ORed with the dependency vector. Load instructions depend on stores preceding them to issue to the store queue before accessing memory and their dependency vectors are ORed with the S section of the GenDepR as well. Finally, store instructions OR with the L section of GenDepR to prevent a store from writing to cache while a preceding load is incomplete.

Instruction fusion is also done during this stage. All instructions read bits from the CFUSE vector to determine if they can fuse to one of their operands. This vector contains a 1 for instructions that are fusible, and is reset on read. This means a non-fusible instruction will also reset the CFUSE bit for its operands. Not following this policy may cause a dependency cycle. If fusibility is detected, the identifier of the instruction being fused to is added to the corresponding FUSEPTR vector entry. The dependency for the newly fused instruction is removed from the current dependency vector before insertion into the matrix. If an instruction A is fused with another instruction B, it should not be dependent on B as they will execute at the same time. Dependencies of the instruction fused with is ORed with the dependency vector being prepared. The V section of the current dependency vector is then inserted into the matrix for the entry the instruction fused to.

Finally, the prepared dependency vector is then passed along to the matrix to be inserted in the slot obtained from the renamer. At the same time, the instruction's matrix line number is inserted into the ROB, the instruction is written into the instruction buffer, the corresponding bit in the INUSE arrays is set. The INUSE bit shows the instruction may be part of the in-order state and should not be retired until proven otherwise. The GenDepR is updated to account for new persistent dependencies for load, store and branch instructions.

5.3. Instruction Selection

In this stage, instructions are selected for execution based on information output from the DEPMAT. For an instruction to be selected for execution, another instruction must be dependent on that instruction. We call such an instruction a demanded instruction. Being demanded is not the only requirement for an instruction to issue - all its operands must also be ready. This information is provided via the matrix as well, by means of the COMT and EXCP arrays. If one operand for an instruction has an exception, we must be executing this instruction as a courtesy to another instruction which it was fused to. Since its result will not affect the in-order state, we can go ahead and let it execute regardless of operand readiness. Once we detect an instruction to be executable, we consider its FUSEPTR vector. Since we equalized all dependencies of the two fused instructions, a fused pair will be ready at the same time - one part of the fused pair can't be ready while the other is not. The value from FUSEPTR is decoded, then the entry corresponding to the set wire is also sent for execution in the same execution unit. When there are multiple candidates for execution, instructions without control dependencies are given priority since they are the oldest and won't be rolled back in case of a misprediction. Otherwise, instructions are selected starting from the higher index numbers in the matrix. This gives priority to branches above other computation. Once an instruction is sent to an EU, its INEX flag is set to indicate it's currently executing.

5.4. Execution

Instructions read their registers, execute their operation and write back their results in this stage of the pipeline (represented as 3 stages in the cycle accurate simulator). Once an instruction finishes, it sets its COMT bit to indicate that there is now valid data in this entry and resets its INEX bit to indicate it is no longer executing. A completed instruction clears its line (except control and load predicate dependencies) in the DEPMAT - it has completed, has all it needs from other instructions, and is no longer dependent on anything except incomplete branches as well as load predicates in case of stores. Completion for a load instruction means a successful access to the store queue or an access to the cache (not necessarily a hit as long as the cache fulfills misses in correct order). A completed load also clears the load predicate it was attached to. Completion for a store instruction means a successful write to the store queue. Completed load, store and branch instructions reset their bits from the GenDepR, since newer instructions should not be dependent on these any longer.

5.5. ROB Commit

When an instruction reaches the head of the ROB, it leaves immediately after doing some bookkeeping with a few exceptions. Result producing instructions must reset the INUSE bit of the previous physical mapping of their logical destination and record their mapping in the rear end map table. Branch instructions must check to see if they've completed before leaving the ROB. Each instruction successfully leaving the ROB sets its STATE bit to indicate they are part of the in-order state. Store and branch instructions additionally reset their INUSE bit as soon as they leave the ROB. A store may leave the ROB but not be able to leave the store queue due to a load predicate it depends on not being complete. Therefore stores do not reset their INUSE bits until they write to memory. A successfully speculated branch instruction will clear its corresponding column in the matrix to indicate the resolution of the control dependency it represents. A mispredicted branch will trigger the misprediction recovery mechanism.

During misprediction recovery, the processor needs to do the following: (1) Retirement map table (RMAP) must be copied over the front end map table (FMAP); (2) Since everything fetched after this branch is incorrect, and branches and stores do not leave the ROB until they are complete, any remaining branch and store instructions in the matrix must be discarded and all corresponding bits must be reset; (3) Any remaining instructions which depend on the mispredicted branch (encoded by DEPMAT) are marked as an exception by setting their EXCP bit, and resetting their INUSE bit as they are proven not to be part of the in order state; (4) Fetch, decode and rename stages as well as the ROB must be flushed.

Note that the only selective operation we have to do to recover from a misprediction is the modification of EXCP and INUSE bits. The instruction retire stage will retire these incorrect instructions whenever convenient - no additional logic is required. In fact, with instruction fusion across branches in place, discarding these instructions may require breaking a fused pair, which would be a costly operation in hardware.

5.6. Instruction Retire

An instruction may be cleared out of the matrix, free its instruction buffer entry and release any registers it's holding when the following conditions are met: (1) Instruction has left the ROB (indicated by a set STATE bit) OR the instruction had an exception (indicated by a set EXCP bit); (2) No other instruction depends on the instruction (indicated by the demand signals on the DEPMAT); (3) The instruction can no longer possibly be part of the in-order state (indicated by a reset INUSE bit);

Any instruction fitting this criteria is guaranteed to have no more effect on the output of the processor. Therefore, all resources used by these instructions are immediately released and all control bits pertaining to their operation are cleared. Note that an instruction being complete is not a requirement for it to retire.

5.7. Deadlock Prevention

A demand-driven processor that holds instructions for an arbitrary amount of time may be at a risk of deadlock. Here we will show that LaZy will never deadlock given appropriate resources. Stores are demanded automatically in LaZy, and on completion will clear their INUSE bits to retire. Branches also clear INUSE bits when complete. This release scheme ensures there will be no deadlock due to load predicate, branch and store line unavailability. As long as the processor contains at least one more physical register than double the number of logical registers, there will also be no deadlock for result producing instructions. Consider the pathological case of a series of n load immediate instructions each writing to a different logical register in a machine with n logical registers. None of these instructions would demand another. However, the next result producing instruction has to either demand one of those load immediate instructions, in which case that instruction will get executed and retired. Or, the instruction will end up using a logical register already in use. In this case, the previous definition of the register will be marked dead and squashed.

Inst.	Code Sequence	Fusion Status	Action
i_0	add r1, ...	Fusible (ALU)	Awaiting demand
i_1	lw r2,r1	Not Fusible (Load)	Demands r1
i_2	add ...,r1,r2	Fusible (ALU)	Demands r1 and r2

Figure 7: Fusion Dependency Cycle Example

Incorrectly applied fusion may cause dependency cycles which will cause deadlocks. An example case is given in Figure 7. i_1 and i_2 are dependent on i_0, but i_1 is not a fusible instruction. i_2 is additionally dependent on i_1. If i_2 is fused to i_0, the i_0,i_2 pair must now wait for i_1 to execute. However, i_1 also can't execute since it is dependent on i_0. To prevent such dependency cycles, we follow the policy of resetting each CFUSE bit every time it is read. With this policy, i_0 would be marked as can't fuse once i_1 is renamed, therefore i_2 will not fuse to i_0.

6. Matrix Implementation

We require several different operations from the matrix. When an instruction is retired from the Matrix we need to reset the entire row to eliminate it. We also want to check the control dependency of each instruction to allow for commits to happen. The OR result of the last B bits in the row can tell us if a branch preceding the instruction is still in flight. When an instruction is issued, executed and written back, it needs to clear its corresponding line to stop demanding its sources. Branches, load predicates and stores will additionally clear the entire column to allow any check waiting on these dependencies to resolve. The OR result of the entire column is the required input to wake up unexecuted instructions.

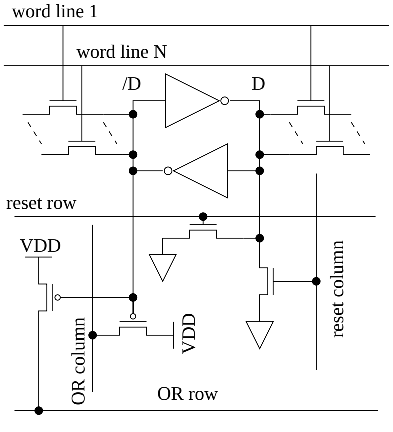


Figure 8: Matrix Cell

One possible implementation of the matrix cell is shown in Figure 8. In addition to an N write port SRAM design, the matrix cell incorporates 2 extra rows and 2 extra columns. The 2 extra rows are used to reset entire rows and OR the last B bits (value used in experiments for B is 16) in the rows which represent the control dependency data. The 2 extra columns are used to reset the columns and OR entire columns.

Each matrix cell has the same length and height. We also need to add some empty area around the cell for routing purposes. If a matrix cell has W write ports, for each cell we have to add $W+2$ word line area and $W+2$ bit line area for routing. W word lines and bit lines are used to write to the matrix cell and the 2 extra word lines are used to reset the entire row and OR the last B bits of the row. The extra two bit lines have a similar function: they are used to reset the entire column and OR the entire column. Using all this information, we can calculate the total area of the matrix and get the load capacitance of each line based on their length and number of MOSFETs they connect. All the formulas are listed below.

$$C_{wordLine} = 2 * \#Cols * C_{gate} + C_{metal} * Length_{row} + C_{driver}$$
$$C_{bitLine} = 2 * \#Rows * C_{drain} + C_{metal} * Length_{column} + C_{driver}$$
$$C_{orRow} = \#Cols * C_{drain} + C_{metal} * Length_{row} + C_{senseAmp}$$
$$C_{orCol} = \#Rows * C_{drain} + C_{metal} * Length_{column} + C_{senseAmp}$$
$$C_{resetRow} = \#Cols * C_{gate} + C_{metal} * Length_{row} + C_{driver}$$
$$C_{resetCol} = \#Rows * C_{gate} + C_{metal} * Length_{column} + C_{driver}$$

Since many instructions only depend on a few others, the matrix is sparse. Therefore, most of the cells contain 0. When we evaluate the power consumption for the matrix, we only activate the bit lines which we need to set to 1, which can save a significant amount of power.

7. Experiments

We simulated a typical superscalar processor as our baseline and LaZy Superscalar. Simulators were automatically synthesized from descriptions written in ADL processor description language [19]. Both simulators are cycle accurate and their ADL implementations respect timing at the RTL level. Baseline processor shown in Figure 9 uses centralized scheduling using broadcasting and wake-up/select is completed in a single cycle. Load and store instructions are issued directly to memory units since address computation is done via splitting the computation into another instruction.

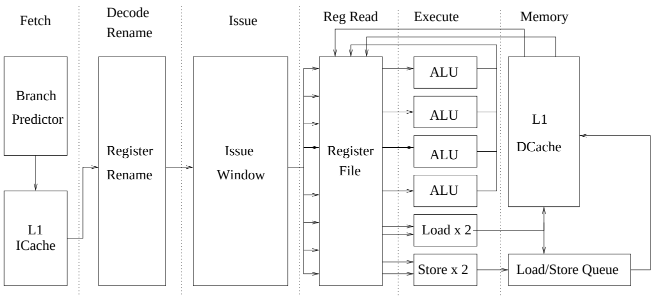


Figure 9: Baseline Superscalar Pipeline

We kept the processors as identical as possible. Both processors use identical fetch engines, fetch, decode and execute the same number of instructions and have identical execution units. It should be noted that both processors also have the same number of read and write ports on their register file. If LaZy Superscalar attempts to execute a fused instruction and

Simulation Architecture

Parameter	Baseline	LaZy
Front End Width	8 wide	
Commit Width	16 wide	
Issue Width	8 wide	
Issue Window Size	128 entries	N/A
Load Predicates	N/A	32
Execution Units	4 int/fp units	
Memory Units	1 Load/1 Store - 2 Load/2 Store	
Rename Registers	128 registers	
Reg. File Ports	16 read, 8 writes	
In Flight Branches	16	
Load/Store Queue	64 entries	N/A
Store Queue	N/A	32 entries
Reorder Buffer	176 entries	
L1 Data Cache	32KB 2-way, 1 cycle lat.	
L1 Inst Cache	32KB 2-way, 1 cycle lat.	
L2 Unified Cache	512KB 8-way, 12 cycle lat.	
Main Memory	80 Cycle lat	
PHT Size	16KB	
Branch Prediction	GShare with 4KB BTB	
Mispred. Recovery	4 cycles	

Table 1: Architectural Parameters Used in Experiments

there are not enough ports to write both results during that cycle, it will stall. In order to have a fair comparison in terms of issue capability, LaZy Superscalar is provided with a 32 entry store buffer and 32 load predicates as it does not have a load queue and the baseline is given a 64 entry load-store queue. The matrix implementation faithfully implements the operation of the matrix at the bit level. Experiment parameters are summarized in Table 1. Baseline superscalar enjoys full age based scheduling (oldest ready instruction in the window always schedules first). LaZy Superscalar issues instructions which are dependent on no branches with priority. The matrix in LaZy already provides a single bit output that is 1 if an instruction is dependent on any branch. This issue policy lets LaZy approximate age based scheduling. Otherwise, instructions are issued based on their location in the matrix.

We executed Spec2006 integer benchmarks which were compiled using gcc version 4.3 with the highest optimization setting (-O3). The software environment used is Binutils version 2.22. Binaries were compiled to MIPS instruction set for Linux kernel 2.6. O/S kernel was not simulated but C library code was included in the simulation. uClibC version 0.9.33 was used to link the benchmarks. We ran the ref inputs for the given benchmarks for 500 million instructions for cache and branch predictor warm up, then for an additional 1 billion instructions to gather performance and other data.

Fused instruction distribution over the benchmarks and the performance data are shown in Figure 10. The data has been collected by running all input sets for a particular benchmark and taking the average of each run. As can be seen, a large fraction of total instructions are fused successfully using the implemented LaZy scheduling algorithm. However, as expected, fusing a large number of instructions does not necessarily lead to improved performance. Bzip2 is an exception showing a wide range of performance depending on the input set (e.g., chicken.jpg: 19.63%, liberty.jpg: -10.18 %). Our investigation yielded that the simulation parameters of warming-up for 500,000M and simulating 1B instructions is not a good fit for this benchmark as it cannot finish loading the liberty.jpg in 1.5B instructions, therefore it still is in its initialization phase. Gcc, mcf and perlbench do not show a commensurate increase in performance to that of number of fused instructions. On the other hand, some benchmarks show a larger than expected performance increase, given the number of fused instructions. This is due to the fact that in these benchmarks, majority of fused instructions are on the critical path. Collapsing such dependencies enables available parallelism to be harvested significantly earlier in the program flow.

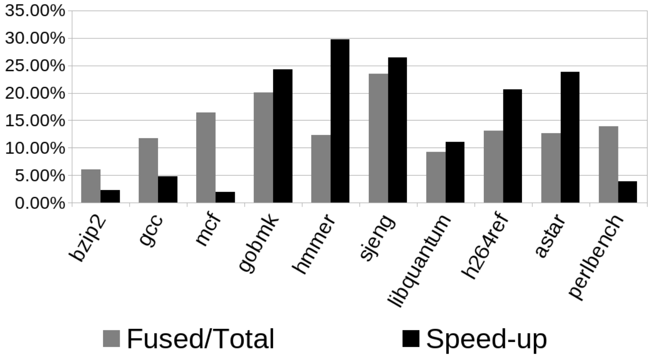


Figure 10: Fused Instructions as Fraction of Total and LaZy Superscalar Speed-up

Excerpt from P7Viterbi in Hmmer

Instruction	Baseline Ex Cycle	LaZy Ex Cycle
`i01 lw   $2,36($fp)`	1	1
`i02 sll  $2,$2,2`	2	2
`i03 lw   $3,80($fp)`	1	1
`i04 addu $2,$3,$2`	3	2 (fused to i02)
`i05 lw   $4,36($fp)`	2	2
`i06 li   $3,1073676288`	1	1
`i07 ori  $3,$3,0xffff`	2	1 (fused to i06)
`i08 addu $3,$4,$3`	3	2
`i09 sll  $3,$3,2`	4	2 (fused to i08)
`i10 lw   $4,92($fp)`	2	2
`i11 addu $3,$4,$3`	5	3
`i12 lw   $4,0($3)`	6	4

Table 2: Execution Profile Fragment from P7Viterbi in Hmmer

We show an example fragment from the P7Viterbi function in the hmmer benchmark in Table 2. In the fragment, 3 out of 12 instructions are fused (25%), which yields a 50% speed-up. Fusing shortens a dependence chain of length 4 ($i06 \rightsquigarrow i07 \rightsquigarrow i08 \rightsquigarrow i09$) to 2. Hmmer spends 99% of it's execution in this function, and P7Viterbi contains many similar sequences back to back and in loops.

Since LaZy only fuses integer instructions, we focus on the integer benchmarks in the Spec2006 set. For completeness, we also evaluated the floating point benchmarks. FP benchmarks get an average performance improvement of 9.06%. Performance profiles are similar to the integer benchmark set.

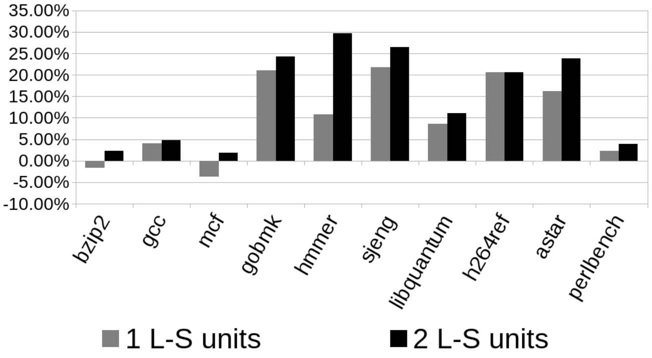


Figure 11: LaZy Superscalar Speed-up with Different Load/Store Units

As LaZy execution is a technique which improves *available parallelism*, it cannot improve the performance of an architecture which cannot harvest what is already available. Improving the available parallelism is not always free due to the delay inherent in our demand-driven model and program periods with largely varying resource requirements. If the improved available parallelism cannot be harvested, LaZy will clearly do poorer. To illustrate the point, we varied the number of load-store units between 1-2. The result is shown in Figure 11. With one load and one store unit, which may not be able to harvest the available parallelism in most of these benchmarks, LaZy actually loses performance in two of the benchmarks whereas increasing the number of load and store units in both the LaZy and the baseline superscalar yields results in which LaZy is clearly superior across the entire suite.

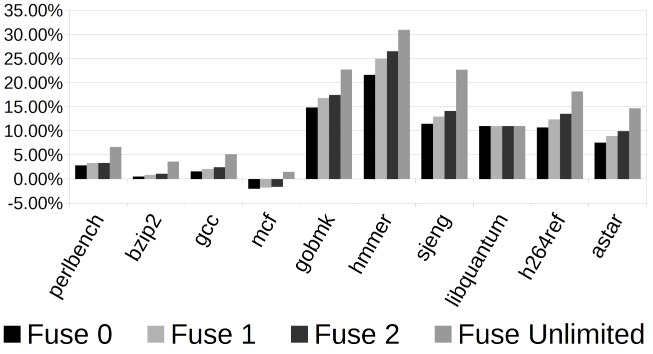


Figure 12: LaZy Superscalar Speed-up with Fusing Over N Branches

We also evaluated a limited version of LaZy Superscalar by disabling instruction fusion when there is a branch between the two instructions (Fuse 0), when there are no more than a single branch (Fuse 1), no more than two branches (Fuse 2) and unlimited number of branches (Fuse Unlimited) with the goal of showing the significance of fusing beyond control dependencies (Figure 12). While there are benchmarks which can benefit from fusing in a single fetch block, without exception all benchmarks benefit from fusion across branches, validating our motivation of designing a general mechanism.

	Baseline			
	Total	**Inst Win**	**ALU**	**RegFile**
bzip2	24.3419	5.3863	1.6016	1.846
gcc	26.739	6.1104	2.0164	2.0082
mcf	18.0578	3.7762	1.1973	1.2332
gobmk	22.6356	4.8681	1.59	1.7178
hmmer	26.3296	5.6902	2.0607	2.2308
sjeng	35.6064	7.4474	2.5745	2.4457
libquantum	35.6022	7.8922	2.5969	2.8325
h264ref	27.0862	6.0292	2.0526	2.2135
astar	30.517	6.9825	2.1313	2.5826
perlbench	22.656	5.4913	1.825	1.7021
	LaZy Superscalar			
	Total	**Inst Win**	**ALU**	**RegFile**
bzip2	23.7724	2.95	2.6083	0.8778
gcc	24.7298	2.85	2.8366	0.8912
gobmk	24.5608	3.01	2.836	0.8543
mcf	18.2716	3.012	1.7223	0.5534
hmmer	28.359	2.89	3.5401	1.1101
sjeng	34.1901	2.845	4.0962	1.1526
libquantum	43.0629	3.01	5.4218	1.9866
h264ref	28.2634	2.802	3.4117	1.1523
astar	33.5478	2.92	3.9513	1.3287
perlbench	20.8651	2.665	2.4505	0.7612

Table 3: Power Analysis (watts)

7.1. Power Analysis

We incorporated power models and estimated the power consumption for both LaZy Superscalar and the baseline. Power values have been obtained by adapting Wattch[2] to the ADL simulator framework. The power results have been validated against the McPAT[15] tool tested with a very similar superscalar pipeline to ensure correctness. The breakdown of power consumption is shown in Table 3 in watts. Within the ten reported benchmarks, LaZy consumes less total power in four. In general, LaZy consumes more power for ALU operations as expected, but makes up for it through reduced power of the matrix implementation. This information agrees with Safi et. al.[25]'s work on the physical characteristics of a matrix scheduler. LaZy appears to consume more power when performance increase is high, with mcf and sjeng being exceptions. Libquantum consumes more power in LaZy due to increased number of data cache accesses.

The calculated energy delay product (EDP) of LaZy over the baseline implementation is 0.92 on average. Assuming both machines can be implemented at the same clock speeds, we believe LaZy promises to be a better approach than conventional eager scheduling with its lower EDP.

8. Related Work

We are not aware of any publications which evaluate a demand-driven scheduler for superscalar processors. We are also not aware of any technique which dynamically fuses instructions and eliminates partially dead instructions automatically as proposed by LaZy Superscalar architecture. However, there are many techniques from which we have benefited in our design. We present an incomplete list of these publications.

There are a number of publications which target the delay of the instruction scheduling logic by using matrix schedulers. One of the earliest publications we could find is the work of Masahiro et.al. [8] which uses a matrix scheduler. This work uses two matrices one for each operand. It reads a column and bitwise ORs it with a ready vector for both Left and Right operands. The paper reports energy savings as well as minor IPC degradation. There are several related patents filed by Intel Corporation. The work of Merchant et. al. [17] uses a dependency matrix for scheduling intel IA32 uops but not branches or stores. A later patent application by Alexander Henstrom [9] incorporates store and branch instruction support. A matrix scheduler similar to the one we are using was evaluated for the physical-level characteristics of a matrix instruction scheduler by Safi et. al. [25]. It shows the power usage on a matrix read or write based on the number of rows in the matrix as well as the delay of the matrix circuit. Techniques developed in this work can also be incorporated into our scheduler to improve delays. Sasaki et. al. [26] group instructions in a traditional CAM scheduling window by splitting the scheduling window into two. Fusion is done between renaming and insertion into the windows by scanning the queues feeding the windows. Unlike our work, an ALU instruction and a load instruction may be fused. Work by Sassone et. al. [27] uses a mapping table between register names and matrix entries. It also incorporates a loose age tracking technique. Our technique can benefit from these ideas as well.

There are several techniques which performed instruction fusing. Work of Hu et. al. [11] accomplishes fusing through binary translation and a virtual machine. This technique fuses 80 % of fusible instructions. Hu et. al. improve on this work to generate more efficient fused code by software improvements and convert the entire pipeline to execute these fused instructions[10]. The patent by Ronen et. al. [24] is a collaborative technique where the compiler generates fused instructions statically and the micro-architecture directs them to fused instruction execution units. Another patent by Gocman et. al. [7] fuses micro-operations generated by a single macro-instruction in the hardware. Most micro-architecture techniques which attempt to use the performance of the wake-up/select logic utilize dependence information. Gochman et. al.'s [6] description of the Pentium M processor includes instruction fusing on some memory operation micro-ops and some test and branch pairs, which is mainly used to virtually improve reorder buffer and issue queue capacity as well as allowing branches to resolve earlier in the pipeline. Kim et. al. [14] shows an in-processor macro-op fusion where instructions with single cycle latency are systematically removed from the pipeline and combined into macro-ops, which are then scheduled nonspeculatively in the pipeline as multi-cycle operations. This technique also focuses on increasing the effective size of the instruction window. Sassone et. al. [28] group instruction sets called strands, which are linear chains of dependent instructions without intermediate fan-out, into macro operations, thus freeing up reorder buffer and issue queue space. Additionally, the paper explores the optimization of these strands. Sassone et. al. improve on their previous work by proposing the usage of static strands [29], which are dependent chains without fan out which are exposed by a compiler pass. These instruction chains can be manipulated to improve the processor's power usage and increase the pipeline resources. Bracy et. al. describe a technique called handle prefix outlining [1], which is a hybrid technique combining PRISC [23], static strands and CCA-subgraphs to aggregate instructions. Clark [4] takes on the problem of adding general purpose accelerators to microprocessors. This becomes relevant to instruction fusion when we note that computation accelerators collapse portions of an application's data flow graph, which is what fusing instructions aim to achieve.

9. Conclusion

We have presented a novel architecture which schedules instructions based on demand signals from other instructions. The architecture naturally eliminates dead code and fuses instructions which can be fused together to remove delays from the critical path of the program. There are several contributions we make: (1) We contribute an alternative superscalar pipeline which implements a general dependency tracking mechanism for all instruction types and we show that such a mechanism is viable for superscalar processors. This general dependency tracking mechanism allows unification of dependence checking for all instruction types and results in a cleaner layout of the pipeline. In this organization, memory operations can be treated very much like any other instruction and can be combined into the main scheduler. (2) We contribute a fine-grain state handling mechanism which permits instructions to be retired in any order and over a large time span. The fine-grain state mechanism should work with other novel approaches in which in-program order of instruction retire may not be possible. (3) We show that aggressive instruction scheduling is not a must for good performance and show that a lazy architecture can be effective in improving performance, particularly with those benchmarks with highly dependent code. (4) We show that combination of shallow pipelines with operation fusing is another dimension of processor optimization, as opposed to removing delays through deeper pipelining.

Our future work includes more thorough experimental evaluation of the paradigm, modification of the selection logic so that an execution slot is not wasted if there are idle execution units, extension of fusibility to other instruction types and implementation of previously proposed dependence height cutting techniques, such as speculative and non-speculative memory cloaking and bypassing [18, 21]. We are also planning to extend the capabilities of the architecture so that memory dependence speculation is integrated into the pipeline structure of LaZy Superscalar.

References

[1] A. Bracy and A. Roth, "Encoding mini-graphs with handle prefix outlining," University of Pennsylvania, Tech. Rep. MS-CIS-08-36, January 2008.

[2] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. ACM, May 2000, pp. 83–94.

[3] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ser. ISCA '98. IEEE Computer Society, June 1998, pp. 142–153.

[4] N. T. Clark, "Customizing the computation capabilities of microprocessors," Ph.D. dissertation, UNIVERSITY OF MICHIGAN, 2008.

[5] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA '01. ACM, May 2001, pp. 74–85.

[6] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. Valentine, "The intel pentium m processor: Microarchitecture and performance," *Intel Technology Journal*, vol. 7, no. 2, pp. 21–36, 2003.

[7] S. Gochman, I. Anati, Z. Sperber, and R. Valentine, "Fusion of processor micro-operations," February 19 2004, *US Patent* 2004/0034757 A1.

[8] M. Goshima, K. Nishino, T. Kitamura, Y. Nakashima, S. Tomita, and S.-i. Mori, "A high-speed dynamic instruction scheduling scheme for superscalar processors," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 34. IEEE Computer Society, 2001, pp. 225–236.

[9] A. Henstrom, "Scheduling operations using a dependency matrix," April 29 2003, *US Patent* 6,557,095.

[10] S. Hu, I. Kim, M. H. Lipasti, and J. E. Smith, "An approach for implementing efficient superscalar cisc processors," in *Proceedings of the 12th annual International Symposium on High-Performance Computer Architecture*, ser. HPCA 12. IEEE, February 2006, pp. 41–52.

[11] S. Hu and J. E. Smith, "Using dynamic binary translation to fuse dependent instructions," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ser. CGO '04. IEEE Computer Society, March 2004, p. 213.

[12] M. Johnson, *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[13] R. Kessler, "The alpha 21264 microprocessor," *Micro, IEEE*, vol. 19, no. 2, pp. 24–36, 1999.

[14] I. Kim and M. H. Lipasti, "Macro-op scheduling: Relaxing scheduling loop constraints," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. IEEE, December 2003, pp. 277–288.

[15] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. IEEE, December 2009, pp. 469–480.

[16] N. Malik, R. J. Eickemeyer, and S. Vassiliadis, "Interlock collapsing alu for increased instruction-level parallelism," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, ser. MICRO 25. IEEE, December 1992, pp. 149–157.

[17] A. A. Merchant and D. J. Sager, "Scheduling operations using a dependency matrix," December 25 2001, *US Patent* 6,334,182 B2.

[18] A. Moshovos and G. S. Sohi, "Speculative memory cloaking and bypassing," *Int. J. Parallel Program.*, vol. 27, no. 6, pp. 427–456, December 1999. Available: http://dx.doi.org/10.1023/A:1018776132598

[19] S. Önder and R. Gupta, "Automatic generation of microarchitecture simulators," in *Proceedings of the 1998 International Conference on Computer Languages*, ser. ICCL '98. Chicago: IEEE Computer Society, May 1998, pp. 80–89.

[20] S. Önder and R. Gupta, "Instruction wake-up in wide issue superscalars," in *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, ser. Euro-Par '01. Manchester, UK, LNCS 2150: Springer-Verlag, August 2001, pp. 418–427.

[21] S. Önder and R. Gupta, "Load and store reuse using register file contents," in *Proceedings of the 15th International Conference on Supercomputing*, ser. ICS '01. Sorrento, Italy: ACM, June 2001, pp. 289–302.

[22] S. Palacharla, N. P. Jouppi, and J. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. ACM, June 1997, pp. 206–218.

[23] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, ser. MICRO 27. ACM, December 1994, pp. 172–180.

[24] R. Ronen, A. Peleg, and N. Hoffman, "System and method for fusing instructions," January 6 2004, *US Patent* 6,675,376 B2.

[25] E. Safi, A. Moshovos, and A. Veneris, "A physical-level study of the compacted matrix instruction scheduler for dynamically-scheduled superscalar processors," in *Proceedings of the 9th International Conference on Systems, Architectures, Modeling and Simulation*, ser. SAMOS'09. IEEE Press, July 2009, pp. 41–48.

[26] H. Sasaki, M. Kondo, and H. Nakamura, "Energy-efficient dynamic instruction scheduling logic through instruction grouping," in *Proceedings of the 2006 International Symposium on Low Power Electronics and Design*, ser. ISLPED '06. ACM, October 2006, pp. 43–48.

[27] P. G. Sassone, J. Rupley II, E. Brekelbaum, G. H. Loh, and B. Black, "Matrix scheduler reloaded," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 335–346, May 2007.

[28] P. G. Sassone and D. S. Wills, "Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 37. IEEE Computer Society, December 2004, pp. 7–17.

[29] P. G. Sassone, D. S. Wills, and G. H. Loh, "Static strands: safely collapsing dependence chains for increasing embedded power efficiency," *ACM SIGPLAN NOTICES*, vol. 40, no. 7, p. 127, June 2005.

[30] J. Stark, M. D. Brown, and Y. N. Patt, "On pipelining dynamic instruction scheduling logic," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 33. ACM, 2000, pp. 57–66.

[31] N. Zaidi, G. Hammond, K. Shoemaker, and J. Baxter, "Dependency matrix," May 16 2000, *US Patent* 6,065,105.