The Anatomy of the Register File in a Multiscalar Processor

Scott E. Breach T.N. Vijaykumar Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
{breach, vijay, sohi}@cs.wisc.edu

Abstract

This paper presents the operation of the register file in the Multiscalar architecture. The register file provides the appearance of a logically centralized register file, yet is implemented as physically decentralized register files, queues, and control logic in a Multiscalar processor. We address the key issues of storage, communication, and synchronization required for a successful design and discuss the complications that arise in the face of speculation. In particular, the hardware required to implement the register file is detailed, and software support to streamline the operation of the register file is described. Illustrative examples detailing important aspects of the operation of the register file and an evaluation of its effectiveness are provided.

1 Introduction

The Multiscalar architecture is a novel architecture for exploiting instruction-level parallelism (ILP) [1] [2] that speculatively executes multiple operations in parallel, yet provides the semblance of sequential execution. An implementation of this architecture, a Multiscalar processor, is a collection of execution engines which share a common register namespace. Each execution engine, called a *stage*, views the register namespace as a logically centralized register file, though it is actually comprised of physically decentralized register files, queues, and control logic. This approach to the design of the register file allows the processor to exploit communication locality within a single stage and to recover the precise architectural state among multiple stages in an efficient manner.

A successful design of such a register file must provide correct and efficient mechanisms to address the key issues of storage, communication, and synchronization. The *storage* mechanism must provide a means to bind register values to storage and to distinguish between multiple values created for the same register. The *communication* mechanism must be able to identify the correct value for a register and to deliver this value to the appropriate storage. The *synchronization* mechanism must coordinate the concurrent production and consumption of values bound to registers. Each of the issues is relatively straightforward in normal execution, but becomes complex in the presence of unconstrained speculation.

The remainder of this paper focuses on the precise details of how the register file in a Multiscalar processor manages the issues identified above. Section 2 provides background information on the Multiscalar architecture. Section 3 describes the basic function of the register file and the interaction of its components. In Section 4, we explain the role the compiler may play to support the operation of the register file. Section 5 provides examples to illustrate the working of the register file. Section 6 provides an evaluation of the register file. Section 7 offers a summary of this work.

2 Multiscalar Execution Model

To support the Multiscalar execution model, the stages of a Multiscalar processor are organized in the form of a circular queue (see Figure 1). The processing element of each stage executes, in either sequential or parallel fashion, the instructions within a subgraph of the control flow graph of the program, called a *task*. (Herein, we do not detail how tasks are selected in a program, as a proper treatment of the subject is beyond the scope of this paper.) Intra-task communication is localized within the stage. Inter-task communication is structured to proceed in one direction (from predecessor to successor stages) around the circular queue via point-to-point connections between stages. The control flow order of the active tasks in execution on the stages, from least to most recent in the dynamic instruction sequence, is maintained by means of *head* and *tail* pointers to the circular queue. All control and data dependences between tasks are enforced via hardware mechanisms to guarantee correct behavior.

Each cycle that an idle (i.e. not in execution) stage is available, a speculation (control flow prediction) is made to *invoke* the next task in a connected dynamic sequence of tasks. The speculated task represents the continuation of the most recent task, and as such, it executes on the stage which is the immediate successor to the last stage invoked, indicated by the tail pointer. Upon a successful invocation,

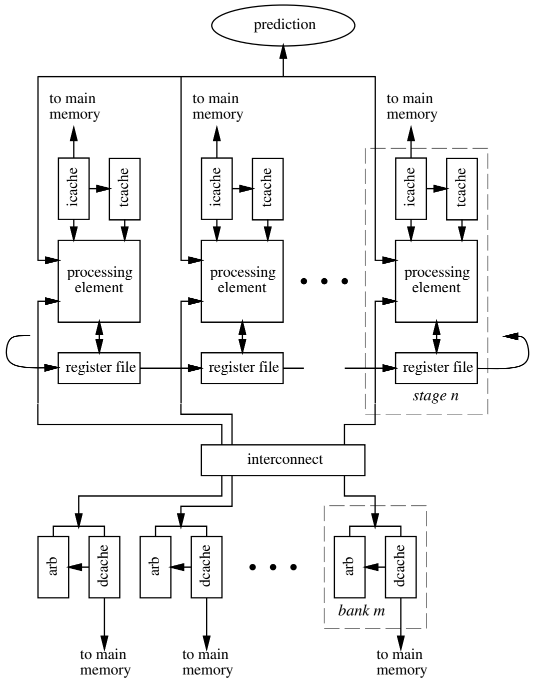

Figure 1: Organization of a Multiscalar processor.

the tail pointer is advanced to the next stage to activate the new task (and busy the stage). A stage executes the instructions that comprise its task until the task is complete.

A stage that has completed its task is ready to update the processor state to reflect the effects of the task; that is, to *commit* the instructions of the task. However, all stages except the head are in execution via a chain of unverified speculations (control flow to the next task is not resolved until the last dynamic instruction of a task). For this reason, only the task at the head is able to commit its instructions. Upon a successful commit, the head pointer is advanced to the next stage to end the task (and idle the stage). In the event an incorrect execution (due to speculation) is detected, all stages between the point of the incorrect execution and the tail are recovered in what is known as a *squash*.

A squash recovers the correct state of the stages involved in the incorrect execution in a manner that is transparent to the program in progress. In the cycle following the detection of the erroneous execution, all stages between the point of detection and the tail are marked as squashed, and the tail is adjusted to resume invocation at the point of detection. The correct state of each squashed stage is recovered at the time it is invoked (again). The actions that are necessary to restore the stages take place in the background of the corrected execution.

To illustrate the execution model, consider Figure 2 which shows basic Multiscalar operation on a 4-stage processor. Relevant portions of the control flow graph of a sample program are provided, with each task represented by a lettered box. Task A is followed by either task B or task C, each of which is followed by task D. Task D either loops back to itself, or proceeds to some other portion of the program. Assume prior to the code fragment shown, registers r2 and r3 were created by some task. Register r1 is created in task A, neither created nor used in tasks B or C, and used in task D. Register r2 is used and created in task B or task C and subsequently used in task D. Register r3 is neither created nor used in tasks A, B, or C, but is used in task D.

To begin execution, task A is invoked on stage 0. A speculation is made that task B will be executed, hence it is invoked on stage 1. As the only successor to task B, task D is invoked on stage 2. A speculation is made that control will loop back and execute task D again, so another instance of task D is invoked on stage 3. At this point, all stages are active and busy executing instructions from their respective tasks. During execution, it is detected that an earlier speculation incorrectly invoked task B. Accordingly, stages 1, 2, and 3 are squashed, and the correct task, task C, is invoked on stage 1. As the only successor to task C, task D is invoked on stage 2, while task A is committed at the same time. Again, the loop back is speculated, and another instance of task D is invoked on stage 3.

As execution unfolds with tasks A, B, D, and D in stages 0, 1, 2, and 3 respectively, stage 0 creates a value for register r1. This value must be propagated to stages 1, 2, and 3 to ensure that all stages have the latest (and correct) value for register r1. Likewise, stage 1 creates a value for register r2 which must be propagated to stages 2 and 3. At the point task B is squashed, the former value of register r2 must be recovered in stage 1. In addition, stages 2 and 3 must recognize that the value of register r2 that propagated is not correct and that tasks in execution on these stages must not use the incorrect value of register r2. The value of register r1 created and propagated by stage 0, however, can be used in stages 2 and 3, as it is indeed a correct value. Since none of the active tasks create a value for register r3, the value propagated to all stages is correct.

In the following section, Section 3, we show how the basic functionality of a logically centralized register file can be implemented as a physically decentralized collection of register files, queues, and control logic. In Section 4, we show how the compiler may be involved in the operation of the register file by providing information that is readily available from a simple analysis of the program control flow graph.

3 Distributed Register File Structure

3.1 Basic Organization

The basic components of the register file in a Multiscalar processor are a modified register storage, a register queue, and a collection of register control bit masks connected to a pipeline core (see Figure 3). The register storage serves as a repository for register values. It is comprised of two register banks to maintain past register state (of the execution of predecessor tasks) and present register state (of the execution of the current task). The queue serves as the mechanism for communication and harbors register values whose communication to successor tasks is imminent. The register control is responsible for coordination of communication and synchronization. It performs this function via simple logical operations on a collection of bit masks: create mask, accum mask, recv mask, sent mask, recover mask, and squash mask (the following subsections illustrate the use of these masks).

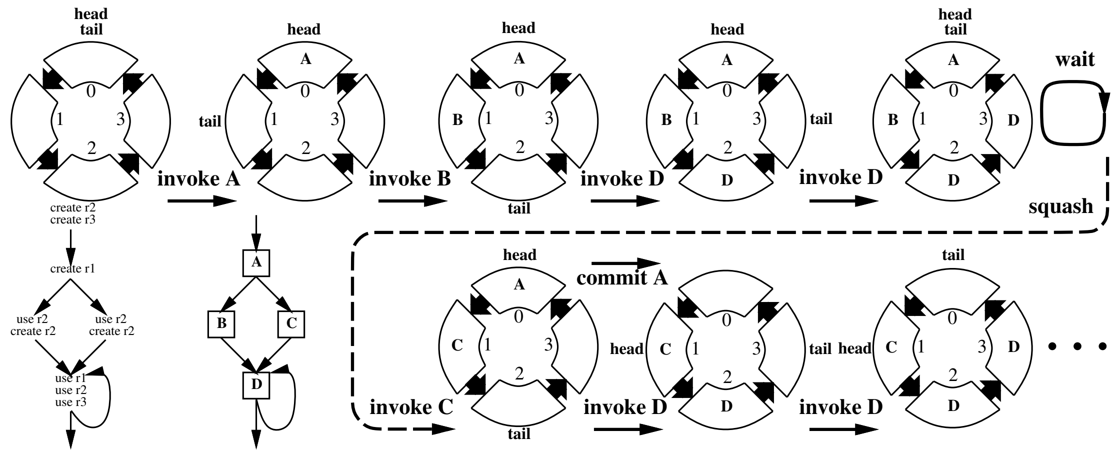

Figure 2: Illustration of the execution model of the Multiscalar architecture.

3.2 Storage

A register may be updated with a value from a predecessor task or a value from the current task. Conceptually, only a single set of register storage is required per stage, as the value produced by the current task logically displaces the value produced by the predecessor task. The nature of speculation is such that incorrect execution can occur at any time in any active stage except the head. Hence, it must be possible to recover at any time correct values for a stage whose register file is updated with incorrect values. The recovery may be performed by having unaffected stages propagate the correct register values to the squashed stages. Unfortunately, a naive approach to this solution contains several pitfalls. If it is not possible to rely on squashed stages to supply the precise identity of the registers to be recovered, the unaffected stages must by default send all registers. As squashes need not be rare, the register communication bandwidth required in this case is prohibitive. Indeed, it may be possible to rely on the squashed stages to identify the registers to be recovered. However, bidirectional communication between stages is required instead of unidirectional communication, which complicates the overall control structure.

A solution to the performance and complexity problems caused by one register set is to have two register sets, a *past* register set and a *present* register set. The past register set contains register values that have been created by predecessor tasks, and the present register set provides working storage for the register values of the current task. Logically, each register set has its own distinct storage; one bank of registers is the storage of the past set, and one bank of registers is the storage of the present set. Physically, if the past and present sets are separated into disjoint register banks, an expensive copy operation (of present to past) is required each time a task is committed. To avoid this multiple register copy operation, we maintain a collection of pointers using additional control bits to provide the illusion of two distinct past and present register banks, somewhat similar to approaches proposed to implement boosting [3] and precise interrupts [4].

The control bits consist of two bit masks, the *past mask* and the *present mask*. The physical registers are divided into two banks, bank 0 and bank 1. The bit corresponding to register r_i in the past or present mask determines the physical bank which contains the corresponding instance of register r_i. At the time a task is invoked, the past and present masks are identical. As the current task executes, the past and present masks are inspected and manipulated to coordinate register accesses. A register read inspects the present mask to access the appropriate register bank. A register write inspects the past mask (and chooses the other bank) to access the appropriate register bank and updates the present mask to reflect the bank which contains the present value of the register. In this way, register values from the past and present are kept distinct. At the time a task is committed, the past mask is updated with the present mask. In the event a task is squashed, the present mask is updated with the past mask, effectively discarding the (incorrect) present registers and retaining the (correct) past registers.

3.3 Communication

As a task executes, it produces register values. The *create mask* of a task identifies all registers for which values may be created by the execution of a task. Some of the register values need to be forwarded to succeeding tasks (and stages). In particular, the last instruction that writes into a register which is live outside the current task needs to propagate the register value. This approach requires an indication of which instruction is the last one to update a register in a task. (An earlier instruction, for example any instruction that writes a register specified in the create mask, could also forward a register value, but this increases the chances of a later task being squashed from the use of an incorrect register value.) Nonetheless, the last instruction in a task to update a register can be determined in a straightforward manner from the control flow graph, as is discussed in Section 4. (If no indication is available regarding when a register value should be propagated, the task must wait until the end of its execution to propagate the values of registers specified in the create mask.)

The register values in flight pass between stages via queues. As a value propagates, it is written into the past register set

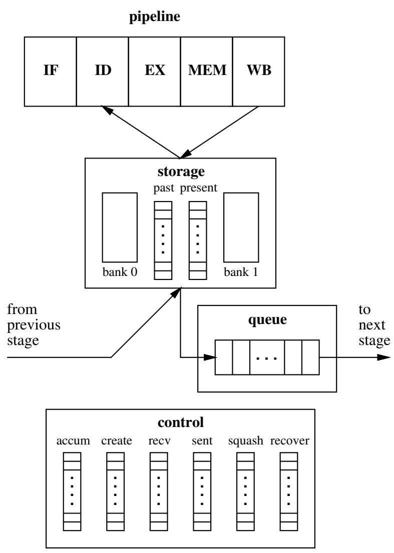

Figure 3: Basic components of the Multiscalar register file.

of the current stage and is propagated to the next stage as determined by the task in execution. The passage of registers between stages is recorded on the *recv mask* and the *sent mask* of a task. The appropriate bit in the recv mask is set to indicate that the register has been received by the current stage; the appropriate bit in the sent mask is set to indicate that the register has been sent to the next stage. A register value need not be propagated by a stage if the task in execution creates another instance of the same register. As per sequential execution semantics, the more recent value is the one that is to be used by successor stages. If the passage of a register is not blocked by a stage, it propagates around the ring until it reaches the stage that created it. Nevertheless, a register value can only propagate among the active stages in the processor, which implies that register values must wait at the tail (until such time as the tail advances). Accordingly, each register value propagates at most once around the ring, although most register values propagate far less, depending upon the characteristics of the tasks [5].

3.4 Synchronization

To provide synchronization between tasks, the control in each stage must identify if and when the correct register values for an instruction to execute are available. In particular, the control must (i) recognize what register values are produced by all predecessor tasks, (ii) determine if these register values are still in flight, and (iii) wait until such time as these register values are received.

The create mask of a task is not sufficient to coordinate the production and consumption of register values among all (possibly non-adjacent) tasks. As a task may require register values from any one of its predecessors, only the combined create masks of all predecessors supplies the required information. This amalgam of create masks is assembled into the *accum mask* (or cumulative create mask) of a task. The accum mask is provided to a task being invoked by its immediate predecessor. To assemble the accum mask (for the next task invoked), a stage performs the bitwise OR of the accum mask and the create mask of the current task. The accum mask synchronizes the consumption of registers in the current task with the production of registers in the predecessor tasks. Of the registers indicated on the accum mask, any that have not yet been received (indicated by the recv mask), are busy. A read of a busy register may require a stall in the stage; a write of a busy register never stalls, as it produces, rather than consumes a register value.

An important consideration with regard to the mechanism defined above is that it provides a means to set bits in the accum mask, but no means to clear bits in the accum mask. In short order, all bits of the accum mask would become set. As such, any task invoked would expect to wait in all cases for all registers. To deliver the desired semantics of this strategy, any bit of the accum mask that corresponds to a register instance that has made a complete cycle back to its creator must be cleared at the next task invocation. In order to provide this functionality, each bit of the accum mask must be tagged with its corresponding creator stage. The drawback of such an approach is that $\log_2(n)$, where n is the number of stages, times as many bits are required to represent the accum mask, and hence must be communicated between stages at task invocation. An alternative to this technique is to produce the accum mask by combining create masks unchecked in the hardware, but relying on the compiler to explicitly remove dead registers via a *kill mask* for each task. (This alternative is under investigation.)

3.5 Recovery from Incorrect Execution

In the event of incorrect execution due to speculation, a mechanism must be in place to recover the proper register state in each stage. In the most basic terms, any register which has been created in a stage that is a part of the incorrect execution must be recovered. The recovery actions are simple for incorrect register values that have been contained within a stage; the two bank register storage allows all incorrect registers to be discarded from the present storage with the past storage preserving the correct values. As might be expected, the recovery actions are more complex for incorrect register values that have been propagated from a stage (to other stages). In any event, the goal is to recover the minimal amount of register state to afford correct execution.

The difficulty that arises in recovery is not identifying which registers must be recovered; rather, it is discovering precisely which successor stages have been modified by incorrect register values. Without analyzing the aggregate register state information from all stages at the same time, optimal recovery is not possible. In general, though, correct recovery does not require this level of detail. Instead, recovery can be performed, not all at once, but distributed over time, as each squashed stage is invoked again. To approach the problem, it is necessary to identify the extent of the incorrect execution. The extent may be broken down into what stages have been affected and into what registers within stages have been affected.

The affected stages are those stages from the point of the squash to the tail. The stage at the point of the squash and the stage at the tail are recorded in the prediction hardware as the *squash head* and *squash tail* respectively. The actual head remains as it is, and the actual tail is changed to the stage at the point of the squash (as this stage is where task invocation resumes). Thus, the stages between the squash head and the squash tail are to be recovered. No other stages participate in recovery because no other stages could have been affected by the incorrect execution.

The determination of which registers to recover can be made via simple logical operations on the register control bit masks. For each squashed stage, all registers which have been produced and propagated from the corresponding task are placed on a *recover mask*. That is, the recover mask indicates the registers for which the stage must propagate correct values to replace the incorrect ones. The stage assumes the role of creator for each register to ensure that all stages see the corrective update. A *squash mask* is provided to synchronize the recovered values. The squash mask is the combined recover masks of all predecessor stages involved in the recovery process (the analog to the accum mask for the create masks).

The squash mask is transferred along with the accum mask to each stage that invokes a task in the midst of an outstanding squash. As each squashed stage is invoked again, it receives a squash mask from its immediate predecessor stage and removes any registers in the squash mask from its recv mask. To assemble the squash mask (for the next stage to be invoked again), the stage produces a recover mask (of its prior incorrect execution) and performs the bitwise OR of the squash mask and this recover mask. The registers in the squash mask of the product are removed from the sent mask of the stage. The earlier difficulty with respect to a means to set bits, but no means to clear bits, is not an issue for the squash mask because its scope is confined to the stages between the squash head and squash tail.

In the event another squash occurs before an outstanding squash has been entirely recovered, it is not the case that the recovery information for the new squash is stacked upon the recovery information for prior (possibly multiple) outstanding squashes. Such an approach may require the assemblage of an unbounded quantity of state. Instead, the recovery information for the new squash is integrated directly into the recovery information for the prior outstanding squashes. Any registers which have been recovered need not be recovered again, unless required by the new squash and are added to the existing recover mask.

4 Compiler Support

To improve the efficiency of the register file, the compiler may be of assistance by providing information that would otherwise not be available (delaying register file actions) or otherwise have to be computed at run-time (complicating register file design). We limit the discussion of compiler support in this paper to (i) a determination of which register values a task may produce (that is, the create mask) and (ii) an indication of which instructions are the last updates of the registers. This information is readily available from a simple analysis of the control flow graph.

If a single path of control is maintained throughout a task, the analysis to be performed is straightforward. The create mask is determined by noting the destination registers of all instructions within the task. The create mask can be optimized to include only register values that are live outside the task with a live register analysis. With only a single path of control, it is simple to determine which is the last instruction in a task to update a register. This instruction is tagged with a *forward bit* so that when the instruction is executed, the value created for its destination register is propagated to successor stages. Tagging an instruction with a forward bit can be accomplished in a variety of ways. The two ways that appear most promising are (i) using special opcodes for such instructions (which we call *op-and-send* instructions) and (ii) using a specially inserted bit mask, similar to the GUARD instructions proposed in [6].

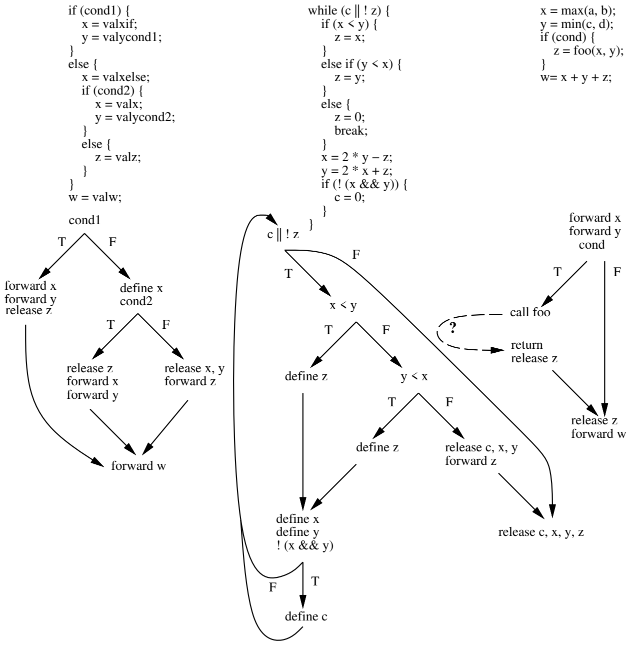

Figure 4: Identification of forward and release for tasks with simple control flow, loop, and procedure call.

If the path of control in a task is complex, the situation becomes more involved. The instructions may be tagged with forward bits as before, but the determination of the create mask may be somewhat problematic. As the dynamic path through the task is unknown, the create mask must allow for all possibilities and must reflect the union of registers created on all possible paths through the task. Depending upon the actual dynamic path of execution through the task, some register values indicated in the create mask may never be produced (due to the conservative nature of the create mask). Unfortunately, it is not possible to determine which registers fall into this category until the end of the task is reached at run-time. As the registers on the create mask imply the need for synchronization with successor tasks, waiting until the end of the task to fulfill the obligations on the reserved registers could significantly delay the execution of successor tasks.

The performance penalty caused by the conservative nature of the create mask can be overcome with the use of a *release instruction*. The function of this instruction is to release the reservations made on registers by the conservative create mask. The release instructions are inserted at appropriate points in the code where it is known that a particular register value cannot be created by any subsequent execution of the task (because execution has proceeded down a path that cannot create the register value), even though the create masks indicates that the task might create the value. The execution of the release instruction causes the propagation of the specified registers. Thereby, successor tasks which might be waiting on the released register value are able to proceed without further delay.

A task that contains a procedure call requires special handling. The compiler may conservatively assume that all registers are created, causing increase in the register traffic. Alternatively, the compiler may compute the exact create mask, requiring inter-procedural dependence analysis. Fortunately, most compilers (including the Multiscalar compiler) follow caller/callee save conventions for procedure calls. This practice motivates a simple, yet effective, solution. The register values that are created inside the procedure are not live after the procedure call returns. The only exception to this rule are values returned in a register and global variables allocated to a register. The create mask would necessarily include any values returned and any global variables modified by the callee procedure.

Figure 4 shows three examples of forwards and releases. In the figure, only the last update of a value down a path is a *forward*, other updates are indicated as *define*. The *release* indicates the affected values. All the variables are assumed to be register allocated.

The first example shows forwards and releases in the presence of acyclic control flow in the task. If *cond1* is true, then x and y are forwarded by the instructions that compute them. Since z is not defined down this path, it is released. The same actions are taken if *cond2* is true. Note that the earlier define of x (if *cond1* is false) does not forward it. If *cond2* is false, then the instruction that computes z forwards it, and x and y are released. Since w is independent of control flow, it is always forwarded.

The second example shows a task which contains an entire loop. All iterations of the loop are performed within the task. The registers that are created in the loop can only be forwarded/released when the loop is exited. This behavior is manifest because a guarantee of no more defines of the registers can be given only at the exit points of the loop. In the example, there are two exit points from the loop, the loop condition ($c \,\|\, !z$) being found false or the break being executed. All of x, y, z and c are defined in the loop. If the loop is exited via the break, then z is forwarded, and x, y and c are released. If the loop is exited due to the fall through of the loop, then all are released. Note that since the two exit paths merge, the hardware may encounter releases for the registers, even though it had already forwarded/released them in the break. In this case, the redundant releases are ignored.

The third example shows a task which contains a procedure call. That is, all the instructions of *foo* are performed within the task. Assuming no global register allocation in the procedure *foo*, the return value, z, is released after the call returns. All the registers created within the procedure *foo* are completely "hidden" from other tasks because they are neither included in the create mask of the task nor forwarded/released from within *foo*.

5 Working Examples

Herein, we provide two examples to illustrate some of the basic concepts of the register file. The examples show the behavior of the register file in the case of normal and squash situations. The normal situation example (see Figure 5) walks through the synchronization and communication of registers among the stages in a 4-stage Multiscalar processor. The squash situation example (see Figure 6) shows the process of recovery unfold following a squash in a 4-stage Multiscalar processor.

In the normal situation example, we show the execution of a sequence of tasks with the accompanying manipulation of the control bit masks that orchestrate synchronization and communication among the stages. The sequence begins as stage 0 invokes the prologue of the loop, continues as stages 1 and 2 each invoke a body of the loop, and ends as stage 3 invokes the epilogue of the loop. Note that the <f> following an instruction indicates that the instruction forwards its result. In addition to following the sequence of instructions for each stage, we have supplemented the control information with some additional bit masks to improve the navigability of the register state for the reader. The supplemental bit masks are a *busy mask* to identify registers on which the task must wait, a *forward mask* to indicate registers that have been forwarded by the task, and a *release mask* to show registers that have been released by the task. (The supplemental bit masks are not part of the actual register control.)

For this example, we have assumed perfect prediction of the loop. In addition, we make use of the ability of the compiler to use a register as a local temporary by not including it in the create mask (in the case of r4). It should be noted that this example is not a cycle-by-cycle record of events. Instead, it only provides enough detail to follow the general operation of the synchronization and communication mechanism. Notice that once it has been determined that a register is to be forwarded or released, it is propagated at the next available opportunity. In a situation where multiple registers are ready, we employ the heuristic of giving priority to a register created within the stage over a register created from a previous stage. Observe that the loop constant (size in r3) is simply passed through the stages, as it is not on the loop body create mask. The loop induction variable (s in r2) is created and propagated for each stage, whereas the loop dependent variable (val in r1) is created and propagated for stage 1, but passed for stage 2.

In the squash situation example, we show the execution of a sequence of tasks which results in a squash and follow the subsequent recovery. We employ the same format as the previous example to assist the reader in following the progression of events. Although this example is similar to the previous one, it does not have the benefit of perfect prediction of the loop. The sequence begins as stage 0 invokes the prologue of the loop and continues as stages 1, 2, and 3 each invoke a body of the loop. However, in this instance, the loop ought to have executed only once. Hence, a squash occurs at stage 2. Observe that all stages from the stage at the point of the squash to the stage at the tail must take recovery action, as indicated by the squash head and squash tail.

The recovery begins as stage 2 is invoked again with a

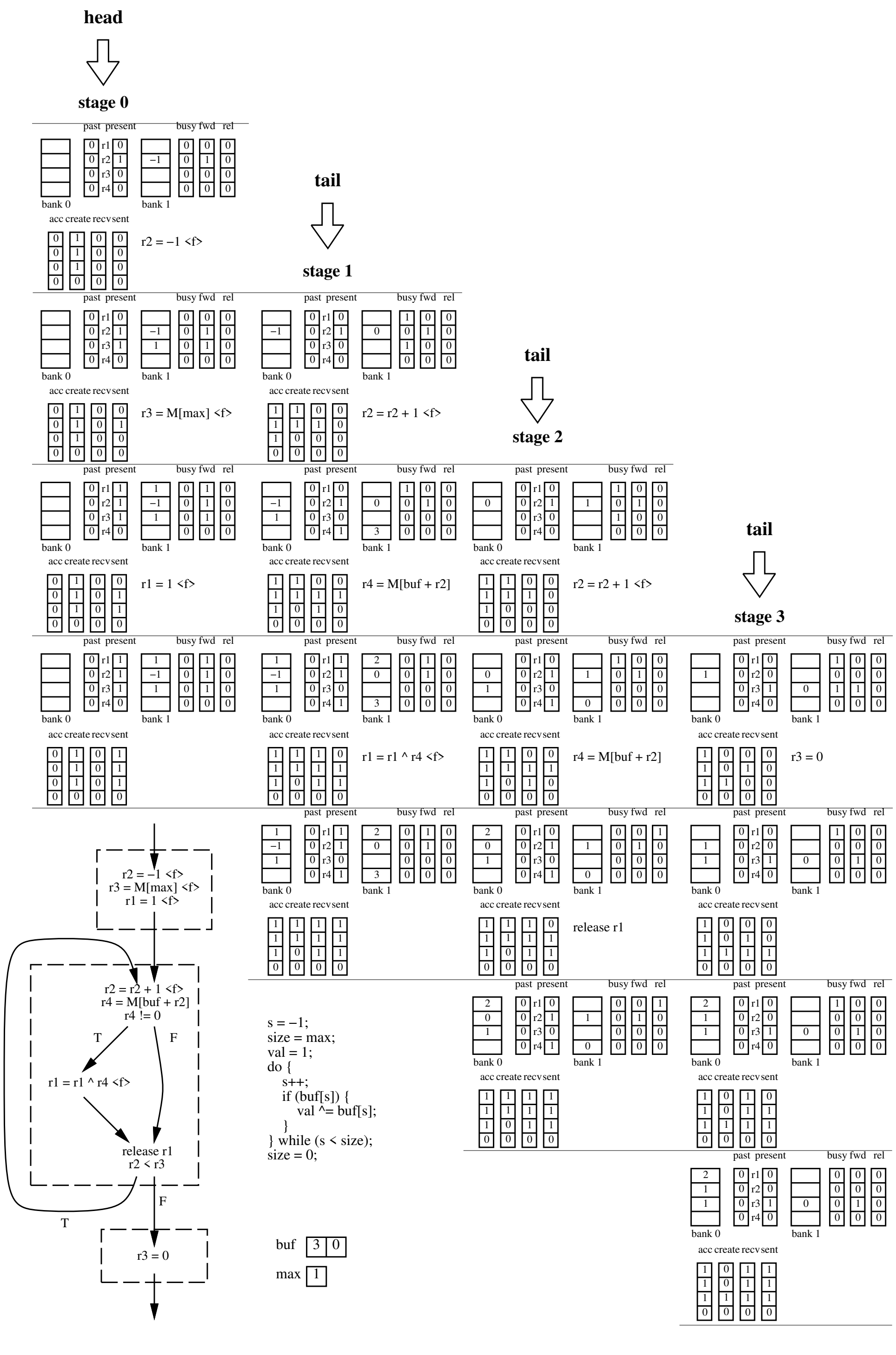

Figure 5: Synchronization and communication among stages.

new task. Notice that the loop induction variable (s in r2) is placed in the recover mask since it was created and propagated during the prior incorrect execution; the register is removed from the sent mask because its earlier propagation was incorrect. The recovery continues as stage 3 is invoked again with a new task. Although the loop induction variable was created during the prior incorrect execution, the value was not propagated (as seen from the sent mask), so the register is not added to the recover mask. Nevertheless, the register is on the squash mask provided by stage 2, so it is removed from the recv mask and returned to the busy mask to ensure that stage 3 waits until the correct register value is propagated.

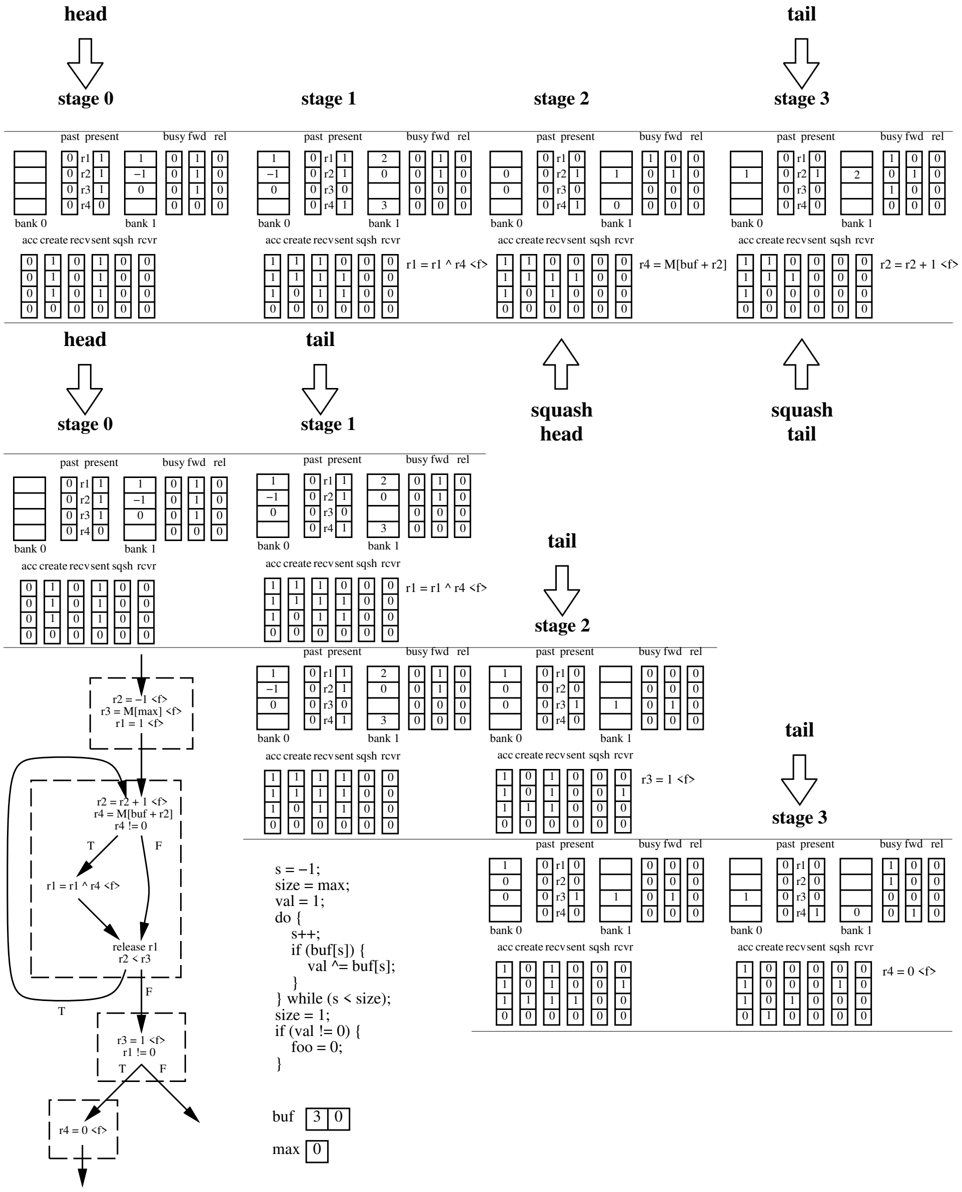

Figure 6: Recovery following a squash.

6 Evaluation

All of the results in this paper have been collected on a simulator that faithfully represents the Multiscalar architecture. The simulator accepts annotated big endian MIPS instruction set binaries (without architected delay slots of any kind) produced by the Multiscalar compiler, a modified version of GCC 2.5.8. In order to provide results which accurately reflect reality, the simulator performs all of the operations of a Multiscalar processor and executes all of the program code except system calls on a cycle-by-cycle basis.

6.1 Machine Model

The machine model that we consider in this paper is based on the organization in Figure 1. We have limited the evaluation to 4-stage and 8-stage Multiscalar processors of this organization. Each stage has been fixed in the configuration of 16 kbytes of direct mapped instruction cache in 64 byte blocks and 3 kbytes of direct mapped task cache in 48 byte blocks (each block is a task header consisting of the target addresses, the return addresses, and the create mask of some task). A full crossbar interconnects the stages to twice as many banks of interleaved data cache. Each bank has been fixed in the configuration of 8 kbytes of direct mapped data cache in 64 byte blocks and 256 address resolution entries [1] (which constitutes a total data cache of 64 kbytes and 128 kbytes for 4-stage and 8-stage processors respectively). The control flow prediction has been fixed in a PAs configuration [7] with 4 targets per prediction and 6 outcome histories. The prediction storage is composed of a first level history table that contains 64 entries of 12 bits each (2 bits for each out-

Int Function	Latency	FP Function	Latency
add/sub	1	sp add/sub	2
shift/logic	1	sp mult	4
multiply	4	sp divide	12
divide	12	dp add/sub	2
memory load	2	dp mult	5
memory store	1	dp divide	18
branch	1		

Table 1: Instruction latencies.

come due to 4 targets) and second level pattern tables that contain 4096 entries of 3 bits each (1 bit target taken/not taken and 2 bits target number). The control flow prediction is supplemented by a 16 entry return address stack.

The processing element of the machine model is a traditional 5 stage pipeline (IF/ID/EX/MEM/WB) with 1-way issue that drains between task invocations. The pipeline stalls on all WAW hazards and on any RAW hazards that cannot be handled via bypassing. Any instruction discontinuity (procedure call, taken branch, jump, etc) incurs an unschedulable 1 cycle instruction refill penalty. Instruction cache hits require 1 cycle to return 4 words. Data cache hits require 2 cycles to return 1 word (2 load delay slots). Any read access that misses in the instruction or data cache stalls the entire pipeline at the point of the access. Any write access that misses in the (write allocate) data cache is handled in the background. All memory requests are handled by a single 4 word memory bus that queues in FIFO order. Each memory access requires a 10 cycle access latency for the first 4 words and 1 cycle for each additional 4 words. Arithmetic instructions are serviced by pipelined functional units, one for integer computation and one for floating point computation, with the latencies indicated in Table 1.

6.2 Benchmarks

Benchmark	Useful Instructions
eqntott	1338.8 M
espresso	733.5 M
cmp	1.4 M
wc	1.3 M
tomcatv	2111.2 M

Table 2: Benchmark dynamic instruction counts.

To serve as benchmarks, we have used *eqntott* with input *int_pri_3.eqn* and *espresso* with input *ti.in* from the SPEC92 integer suite; 2 common Unix utilities, *cmp* from the GNU diffutils 2.6 with input *cccp.c* and *wc* from the GNU texutils 1.9 with input *cccp.c* (Note: these are the same utilities, albeit from different source code, with the same inputs as used by the IMPACT group in [8]); and a f2c translation of *tomcatv* from the SPEC92 floating point suite. All have been compiled with the Multiscalar compiler at optimization level -O2. All benchmarks have been simulated to completion and verified instruction-by-instruction during simulation. Dynamic instruction counts for the benchmark are given in Table 2.

	Useful Instructions Per Cycle					
	1 Cycle			2 Cycle		
Benchmark	1	2	∞	1	2	∞
eqntott	1.85	1.88	1.88	1.83	1.86	1.86
espresso	1.26	1.28	1.28	1.24	1.26	1.26
cmp	2.97	2.97	2.97	2.97	2.97	2.97
wc	2.06	2.06	2.06	2.04	2.04	2.04
tomcatv	1.73	1.73	1.73	1.73	1.73	1.73

Table 3: Instruction Per Cycle (IPC) on 4-stage processor for 1 and 2 cycle latency with 1, 2, and ∞ bandwidth.

	Useful Instructions Per Cycle					
	1 Cycle			2 Cycle		
Benchmark	1	2	∞	1	2	∞
eqntott	2.86	2.90	2.90	2.74	2.77	2.77
espresso	1.39	1.40	1.40	1.37	1.38	1.38
cmp	5.31	5.32	5.32	5.31	5.31	5.31
wc	3.74	3.74	3.74	3.58	3.59	3.59
tomcatv	2.57	2.57	2.57	2.57	2.57	2.57

Table 4: Instruction Per Cycle (IPC) on 8-stage processor for 1 and 2 cycle latency with 1, 2, and ∞ bandwidth.

6.3 Results

One measure of execution performance is the number of useful instructions completed per cycle, or IPC. In these simulations, useful instructions are all instructions executed, except release instructions inserted by the compiler (the release instructions still consume execution cycles, but are not included in the dynamic instruction counts). The results for instructions per cycle are presented in Table 3 and Table 4 for a 4-stage and a 8-stage Multiscalar processor respectively. The cycle counts for a particular run may be computed by dividing the instruction count in Table 2 by the IPC value for the corresponding benchmark. (For example, eqntott on a 4-stage processor with 1 register bandwidth and 1 cycle latency runs for $1338.8M\ insts/1.85\ insts\ per\ cycle = 723.7M\ cycles$).

To evaluate the effect of the register file on performance, we vary the register communication bandwidth and latency. We compare configurations for communication bandwidth between adjacent stages of 1 and 2 registers per cycle to an ideal configuration of infinite registers per cycle. We consider pipelined communication latencies between adjacent stages of 1 and 2 cycles respectively.

The instructions per cycle figures shown are somewhat pessimistic due to the non-aggressive processing element design under consideration. At present, within a stage we employ only 1-way issue and force the pipeline to drain between task invocations to simplify simulation. In addition all cache miss and data hazards incur stall cycles. Any instruction discontinuity (procedure call, taken branch, etc) incurs a 1 cycle pipeline refill stall.

In this context, the results indicate that a modest communication bandwidth between adjacent stages of 1 register per cycle performs nearly as well as infinite communication bandwidth. Likewise, the small increase in communication latency between adjacent stages from 1 to 2 cycles has only a small effect on performance in terms of instructions per

Benchmark	Avg Reg In Flight	% Total Reg Bandwidth	Avg Reg Queued All	Avg Reg Queued Tail
eqntott	0.92	23.0%	0.84	1.88
espresso	0.56	14.1%	1.29	3.48
cmp	0.39	9.8%	0.67	1.74
wc	1.14	28.6%	0.88	1.61
tomcatv	0.47	11.8%	1.92	8.13

Table 5: Register communication bandwidth on 4-stage Multiscalar processor

Benchmark	Avg Reg In Flight	% Total Reg Bandwidth	Avg Reg Queued All	Avg Reg Queued Tail
eqntott	1.53	19.1%	0.44	1.40
espresso	0.87	10.8%	0.73	3.17
cmp	0.71	8.8%	0.40	1.68
wc	2.08	26.0%	0.48	1.20
tomcatv	0.72	9.0%	0.93	6.01

Table 6: Register communication bandwidth on 8-stage Multiscalar processor

cycle.

We present statistics for the utilization of the register communication hardware in Table 5 and Table 6 for a 4-stage and a 8-stage Multiscalar processor respectively. We limit our consideration to the configuration of 1 register per cycle between adjacent stages with a latency of 1 cycle. Such a configuration is capable of supporting a bandwidth of n registers per cycle in the communication ring, where n is the number of stages. We see from the figures that generally less than a fourth of the available bandwidth is actually used. Moreover, we find that the queues among the stages contain no more than a single register most of the time.

Of particular interest, the queue at the tail stage contains many more registers than the queues of all stages on average. This outcome comes as no surprise, as no register may propagate past the tail. Even so, in absolute terms the number of registers queued at the tail stage is usually quite small. In the cases of espresso and tomcatv, the disparate occupancy of the queue at the tail stage is attributable to the high number of average dynamic instructions per task for these benchmarks. Specifically, loop bodies which execute many instructions tend to occupy stages for many cycles, thereby preventing the registers at the tail from propagation.

Concern for this behavior may be unfounded for two reasons. First, the occupancy of the queues (from the tables above) includes registers that have propagated from predecessor stages, but which cannot be passed due to register reservations, as well as registers that have been propagated by the current stage, yet cannot be sent. Only the registers from the current stage occupy queue storage; the registers from predecessor stages occupy the past register storage. Second, it should be pointed out that the figures are pessimistic since we do not yet use compiler analysis to suppress the propagation of registers which are not live across the task boundary. In fact, most of the registers queued are, in fact, dead registers. At present, we are exploring strategies to prevent the propagation of dead registers from a task.

7 Summary

This paper described the register file in a Multiscalar processor. As this component of the architecture is the primary focus of all inter-instruction communication, its proper organization is a critical issue to avoid a potential performance impasse. We addressed the key issues of storage, communication, and synchronization that are central to a successful design and discussed the complications that arise in the presence of speculation. We provided details of the hardware required to implement the register file, and discussed how software may assist in streamlining the operation of the register file. We evaluated the register file in terms of the overall execution performance and considered the utilization of several configurations on a number of benchmarks. The results indicate that the performance of modest configurations rivals that of ideal configurations. In this ongoing research, we continue to pursue hardware and software alternatives to further improve performance.

Acknowledgements

This work was supported in part by NSF grant CCR-9303030 and by ONR grant N00014-93-1-0465. We would like to thank the anonymous reviewers and the computer architecture group at the University of Wisconsin-Madison for their helpful comments. In addition, we would like to express our appreciation to the IMPACT group in the CRHC at the University of Illinois for assistance with benchmarks.

References

[1] M. Franklin. *The Multiscalar Architecture.* PhD thesis, University of Wisconsin-Madison, Tech Report 1196, November 1993.

[2] M. Franklin and G.S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proc. of ISCA 19*, pages 58–67, May 1992.

[3] M.D. Smith, M.S. Lam, and M.A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proc. of ISCA 17*, pages 344–354, May 1990.

[4] J.E. Smith and A.R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[5] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proc. of MICRO-25*, pages 236–245, December 1992.

[6] D.N. Pnevmatikatos and G.S. Sohi. Guarded execution and branch prediction in dynamic ilp processors. In *Proc. of ISCA 21*, pages 120–129, April 1994.

[7] T. Yeh and Y.N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proc. of ISCA 20*, pages 257–266, May 1993.

[8] R.E. Hank, S.A. Mahlke, R.A. Bringmann, J.C. Gyllenhaal, and W.W. Hwu. Superblock formation using static program analysis. In *Proc. of MICRO-26*, pages 247–255, December 1993.