Integrating Superscalar Processor Components to Implement Register Caching

Matthew Postiff, David Greene, Steven Raasch, and Trevor Mudge
Advanced Computer Architecture Laboratory, University of Michigan
1301 Beal Ave., Ann Arbor, MI 48109-2122
{postiffm, greened, sraasch, tnm}@eecs.umich.edu

ABSTRACT

A large logical register file is important to allow effective compiler transformations or to provide a windowed space of registers to allow fast function calls. Unfortunately, a large logical register file can be slow, particularly in the context of a wide-issue processor which requires an even larger physical register file, and many read and write ports. Previous work has suggested that a register cache can be used to address this problem. This paper proposes a new register caching mechanism in which a number of good features from previous approaches are combined with existing out-of-order processor hardware to implement a register cache for a large logical register file. It does so by separating the logical register file from the physical register file and using a modified form of register renaming to make the cache easy to implement. The physical register file in this configuration contains fewer entries than the logical register file and is designed so that the physical register file acts as a cache for the logical register file, which is the backing store. The tag information in this caching technique is kept in the register alias table and the physical register file. It is found that the caching mechanism improves IPC up to 20% over an un-cached large logical register file and has performance near to that of a logical register file that is both large and fast.

1. BACKGROUND AND MOTIVATION

A large logical register file is a very important aspect of an instruction set architecture because it allows significant opportunity for compiler optimizations. Such optimizations have been shown to eliminate memory operations and speed program execution. Specifically, a logical register file of 64 or more entries is desirable to house locals, optimization temporaries, and global variables [14]. Recent commercial architectures have underscored the importance of a large logical register file as well [6]. More logical registers can also enhance ILP by eliminating memory cache operations, thus freeing the cache for more critical memory operations. The elimination of memory instruction reduces instruction fetch and decode time. Other techniques such as register windowing require a large logical register file in order to eliminate spill/reload traffic [12].

Unfortunately, any large register file with many read and write ports is not practically implementable at clock speeds which are marketable even if the performance advantage of the large file is compelling at slower clock speeds. There have been a number of proposals to circumvent this problem for large register files: either by physically splitting the register file or by providing a cache of the most frequently used registers and having a large backing store for the full *logical* set of registers [5, 20, 21, 17]. The primary observation that these caching proposals rely on is that register values have temporal and spatial locality. This is the same principle that makes memory caches work.

A slightly different observation drives this work. Locality in the rename register reference stream in an out-of-order microprocessor is different than in the logical register reference stream because renaming turns the incoming instructions into a single-assignment program. No register is written more than once; this means that a write breaks the locality for a given architected name. Instead, the observation that we rely on for this work is that most register values are produced and then consumed shortly thereafter: around 50% of values are used so quickly that they can be obtained from a bypass path instead of from the register file. Of the values not obtained from a bypass path, many of them are available within the instruction window, i.e. within the speculative storage of the processor. Such values rarely even need to be committed to the architected state because they will never be used again. Such commits are termed “useless” by previous work, which reports that this phenomenon occurs for 90-95% of values [8].

These facts suggested that we investigate how to implement a large logical register file efficiently in the context of a superscalar processor–a problem that previous work does not specifically address. The result of our investigation is the subject of this paper. The logical register file that we want to implement – 256 registers – is so large that it is actually larger than most rename storage in aggressive superscalar processors built today. The rename storage is responsible for maintaining the storage of speculative values as well as mappings that determine where the architected state of the processor can be found. This immediately suggested that we view the implementation’s smaller rename storage as a cache for the larger architected register file, which would act as a backing store. The goal of this design is to achieve the software performance of a large and fast architected register file without having to pay the hardware cost of implementing it. The cache is meant to reduce the performance hit of the large logical register file.

Before continuing with the explanation of our implementation, we must review two other pieces of background information: *register renaming* and what we call *register architecture*. Section 2 and Section 3 then describe the physical register cache mechanism in more detail. Section 4 enumerates several of the advantages and disadvantages of the proposal. Section 5 evaluates the proposed mechanism by comparing it to a lower and upper performance bound which have no register caching. Section 6 compares our work to previous work in the area of register caching, and Section 7 concludes.

1.1. Register Renaming

In out-of-order superscalar processors, register renaming decouples the logical register file of the instruction set architecture from the implementation of the processor chip. The instruction set may have 32 registers while the microarchitecture implements 80 “rename registers” in order to allow it to exploit instruction-level parallelism by simultaneous examination of a large window of instructions which have been transformed into a single-assignment language to remove anti-dependencies and output dependencies. These rename registers contain state which is speculative (because of speculated branches, loads, etc.).

ICS ’01 Sorrento, Italy
© ACM 2001 1-58113-410-x/01/06…$5.00

Register renaming is implemented in several different ways in commercial microprocessors. These designs are surveyed in detail elsewhere [16] but we describe them briefly here.

One mechanism is called the merged register file, used in the MIPS R10000 and Alpha 21264 processors [11, 19]. In this design, the architected state and rename state are mingled in a single large register file which we will call the physical register file. Both speculative and non-speculative state share the same storage structure in this design. The register renaming and register release mechanisms must be designed so that architected state is maintained in a precise way.

The second implementation of register renaming is the split register file. The architected state is kept separate from the speculative state; each have their own register file and are updated appropriately. This approach is used in the PowerPC 6XX and PA 8000 processors.

The third approach is similar to second in that the architected state is separate from the speculative state, but the speculative state is stored in the reorder buffer. This technique is used in the P6 (Pentium II and III) microarchitecture.

Though renaming decouples the rename storage from the logical view of the architecture, the merged file approach is constrained in that it must implement more rename storage than there are logical registers. We denote this condition by saying that NPR > NLR must hold. Here, NPR is the number of physical (rename) storage locations, and NLR is the number of logical registers in the instruction set architecture. This constraint is most simply explained by noting that the rename storage must have enough registers to contain all of the architected state plus some number of registers to support speculative execution (the result of running ahead of the architected state using branch prediction, etc.). Thus the merged file approach does not take advantage of a complete decoupling of the logical storage from the rename storage.

This difficulty constrains the logical file to be smaller than the physical file, a condition contrary to our initial desire, and prompted us to consider a design which allows more complete decoupling of the physical storage from the logical storage by splitting the logical and physical value storage instead of merging them. The next subsection explains another consideration that makes this conclusion attractive.

1.2. Register Architecture

By the term *register architecture*, we mean the number and configuration of registers need to support program execution. The register architecture has two facets – one facet is the logical register architecture, i.e. the number and configuration of registers supported in the instruction set. The other facet is the physical register architecture, i.e. the number and configuration of registers in the implementation.

The logical register file should be as large as desired by the software that runs on the machine. Compiler optimizations have a significant impact on the logical register architecture [15]. The number of storage locations in the implementation, on the other hand, is related to implementation technology, design complexity, and desired machine capacity, factors which are decided long after the instruction set has been fixed. The physical register file must be matched to the characteristics of the design in order for the design to be balanced, instead of being matched to the instruction set architecture. These different requirements mean that strictly decoupling their designs is advantageous to allow the designer to maximize the performance of both the compiler and hardware implementations.

We consider the split register file model of register renaming. This decision was based on three factors: 1) the logical file is larger than the physical file and thus has a natural backing-store to cache relationship; 2) a merged approach to register renaming cannot have more logical registers than physical registers (otherwise a *register-release deadlock* could occur; 3) the design of the logical file should be decoupled from the design of the physical storage to allow the designer the most freedom to optimize each individually. Advantages of this selection will be presented later.

Table 1: Summary of terminology.

Term	Meaning
LRF	Logical (Architected) Register File
NLR	Number of Logical (Architected) Registers
PRF	Physical Register File (the cache)
NPR	Number of Physical Registers
RAT	Register Alias Table; maps logical registers to virtual registers
VRN	Virtual Register Number
NVR	Number of Virtual Registers
PRFV	Physical Register Free Vector
Architected State	Committed, in-order, non-speculative state of the processor, visible at the instruction set architecture interface.

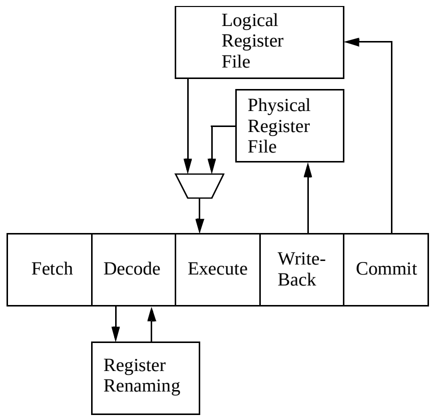

Figure 1: The machine model considered in this paper.

1.3. The Physical Register Cache

The new register caching technique that is introduced in this paper allows a large logical register file to be implemented at a realistic cost. It integrates a number of out-of-order processor hardware elements and combines a number of features of previous designs to arrive at a novel solution to the problem of building a large logical register file. The generic model is shown in Figure 1. It consists of the physical register file (PRF) which contains speculative results and some non-speculative results (this is the rename storage) and a logical register file (LRF) which contains precise architected state at all times. The PRF will have as many registers and ports as required in order to have a balanced execution engine; the LRF will have as many ports as can be sustained within the desired cycle time. Note that though the LRF may be much larger, its access time may not be much worse than the PRF because it will have fewer read and write ports. The other terminology that we will use is described in Table 1.

Instruction values committed by the processor to architected state are committed from the final stage in the pipeline. This is to avoid having to read the value out of the physical register file at commit and is why there is no direct path in the diagram from the PRF to the LRF. Alternatively, read ports could be added to the PRF to allow committed values to be read from it and sent to the LRF.

By design, the PRF is smaller than the LRF to allow for a fast cycle time. The PRF caches recently computed results and maintains those values as long as possible. The cache contains the values most recently defined by the processor. In this way the architected file can be large and somewhat slower while the smaller physical file can have many ports to supply operands to the function units quickly.

The logical registers are mapped to the physical registers through a third (larger) set of "registers" called the virtual register numbers (VRNs). There is no storage associated with these VRNs, which are used to avoid the register-release deadlock, to allow the PRF to be directly indexed instead of associatively indexed, and to allow the PRF (cache) to maintain the values after they are committed to the LRF.

2. REGISTER CACHE DESIGN

The innovation in this paper is the combination of four mechanisms: separate logical and physical register files, a physical register file that is smaller than the logical file, renaming through a larger set of virtual registers, and a simple indexing scheme that maps the virtual numbers to physical registers. The combination of these techniques are used to achieve several goals: to provide a large logical register file that does not impact the speed of the processor's critical path, to avoid deadlock conditions in register assignment that are problems in previous work, and to provide an efficient mapping from virtual number to physical register.

2.1. Microarchitecture Components

The major components in the microarchitecture are:

1. A large logical register file (LRF) which contains precise architected state at all times. There is storage associated with logical registers: there are NLR entries in the LRF. Values are written to the logical register file at the time an instruction commits to architected state.
2. A set of virtual register numbers (VRNs). There is no storage associated with virtual registers: they are just numbers that track data dependences and the location of data (physical or logical register file) [3]. There are NVR virtual registers, where NVR > NLR. The virtual registers are assigned such that the low bits of the identifier index directly into the physical register file and the remaining high bits are called the *check tag*. The purpose of the check tag will be explained later. A VRN is allocated and deallocated as in the merged renaming approach, described in Section 1.1.
3. A physical register file (PRF) which contains speculative values eventually destined for the logical register file. The PRF also has value storage: it has NPR <= NLR entries. Values are written to the physical register file after an instruction computes its result. A result register's value is retained after its producing instruction is committed to architected state until a new value overwrites it. The PRF is directly indexed, unlike in some previous work [2]. The physical register file also contains tag bits (from the virtual register number check tag) to verify that the requested value is present; otherwise the value can be found in the logical register file because it was already committed and overwritten by a later producer instruction.
4. A virtual number free list (VNFL) which contains the numbers of all virtual registers that are currently available for use.
5. A physical register free vector (PRFV) of NPR entries where each bit represents whether the physical register is free to be allocated.
6. A rename table (RAT) of NLR entries each of which contains the virtual register number for the corresponding logical register.
7. A busy bit table of NLR entries which contains a bit for each logical register indicating whether it is presently being written by an instruction in the pipeline. If the bit is clear, then the value can be found in the logical register file. If set, there is a producer instruction in the pipeline which will produce the value at some future point.
8. A set of reservation stations. Each contains the following fields, assuming two source operands for convenience of explanation: a) dest virtual register; b) src1 ready bit; c) src1 source virtual register; d) src1 logical register number; e) src2 fields as for src1.

Table 2: Register cache parameters used in this study.

Param	Value	Comment
NLR	256	The instruction set architecture allows 8 bits for each register specifier.
NPR	64	From the machine capacity.
NVR	512	Since NVR > NLR must be true (constraint 2), we specify 9 bits of virtual register number. The low 6 bits of this are used to index into the 64 physical registers. The remaining 3 bits are used as the check tag.

2.2. Design Constraints

There are several constraints that must be met by the design:

1. NPR < NLR. This is the basic assumption of the cache design of the NPR.
2. NLR < NVR. This ensures no deadlock condition in renaming since the rename register set is larger than the logical register set [11].
3. Number of In-flight Instructions <= NPR. We limit the number of instructions to no more than the number of physical registers. This ensures that each instruction has a unique slot in the physical register file for its result. No two uncommitted instructions can have the same physical index. In other words, the number of instructions in flight cannot exceed the machine's capacity.

3. REGISTER CACHE OPERATION

This section describes the detailed operation of the caching mechanism we propose. For this study, we chose the parameters shown in Table 2.

When an instruction arrives at the dispatch stage, the source register operands are renamed based on the contents of the RAT as in conventional renaming. Both the virtual register number from the RAT and the logical register identifier are carried with the instruction into the reservation station. No values are read from any register file at this time. In this way, values are stored centrally in either the logical or physical register file and are not duplicated in the reservation station entries.

Each destination register in the instruction is assigned a virtual register number from the free list as in conventional register renaming. There is one difference: the physical register free vector is also queried. No virtual register number whose (six) physical index bits are currently in use can be chosen for allocation. This additional constraint is necessary to ensure that no two instructions share a physical index and is a necessary side-effect of the simple mapping policy that is used to index the physical register file (described later). This works without deadlock since the number of virtual registers is larger than the number of logical registers. Once a physical register number meeting this constraint is chosen, its free bit in the PRFV is cleared to indicate that this physical register has been "pre-allocated". A register that is pre-allocated is marked as being in use as a destination register. Our scheme allows the value currently in that register to still be used by consumer instructions until the value is over-written. The relevant structures are overviewed in Figure 2.

Two bookkeeping structures (not shown in the figure) allow the processor to select an appropriate register. The physical register free vector knows about all the free physical storage locations in the system. Similarly, the virtual register free list does the same for virtual registers. An autonomous register selection circuit can examine this information, determine which virtual-physical pairs

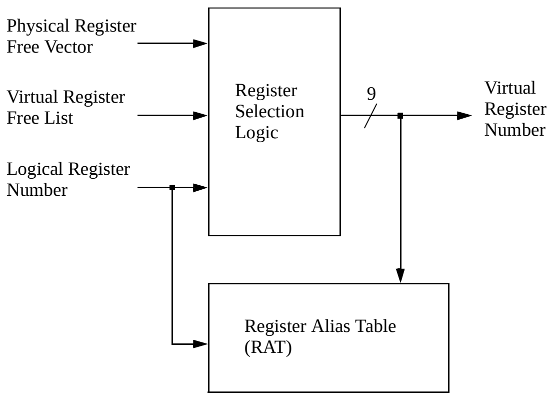
Physical Register Free Vector
Virtual Register Free List
Logical Register Number
Register Selection Logic
9
Virtual Register Number
Register Alias Table (RAT)

Figure 2: The mechanism for renaming a destination register.

are available for allocation, and put them onto a third list which the processor then pulls from, in order, to rename the destination of an incoming instruction. This list is not shown in Figure 2 but is inside the register selection logic box. Essentially, the circuit is looking for a virtual register whose six index bits describe a free physical register. In our system, there are 64 physical registers and 512 virtual tags, so that for any physical register there are 8 possible virtual registers that can meet this criterion. The register selection circuit tries to find pairings where both are free. The register selection circuit has flexibility to choose the registers according to any policy that it likes and can effect different caching policies "offline". If there is no virtual register that qualifies for renaming, the frontend of the processor stalls until one becomes available.

The newly renamed instruction is then dispatched and waits in a reservation station until its operands become ready. Readiness is determined when a producer instruction completes and broadcasts its virtual register number to the reservation stations. Each station compares its unready source VRN with the virtual register number broadcasted. If there is a match the source is marked ready.

At some point all the instruction's operands are ready and it is scheduled (selected) for execution. The low 6 bits of its source operand VRN are used to directly index into the 64-entry physical register file. This simple indexing scheme constrains the initial selection of the VRN (in the renaming pipeline stage) but greatly simplifies the register access at this point. No associative search is necessary.

The upper 3 bits of the VRN are used as the 3-bit check tag whose function is to verify that the value currently in the physical register comes from the correct producer. If the PRF entry has a matching 3-bit check tag, then the value in the physical register is taken as the source operand. If the tag does not match, the value no longer resides in the PRF (like a cache miss) and must be fetched from the LRF. This means that it was committed to architected state some time ago and was evicted from the physical register set by some other instruction with the same low 6 bits in its virtual number. In the case where the value is not available from the physical register file, an extra penalty is incurred during which the backing store (logical register file) is accessed. Our indexing scheme does not allocate back into the cache upon a miss because the value that would be allocated no longer has a VRN (it was committed).

When the instruction issues to a function unit, it picks up the necessary source operands, some from the LRF and some from the PRF. The LRF access can be started in parallel with the PRF access if there are enough ports on the LRF to support this. This is shown in Figure 3, though this approach would require as many ports on the LRF as on the PRF. Alternatively, the LRF can be accessed the cycle after it is determined that the PRF did not contain the value. This latter approach is used in our simulations. Since each physical register could be accessed by multiple consumers in a single cycle, multiple ports are required on the PRF, but this is no different than

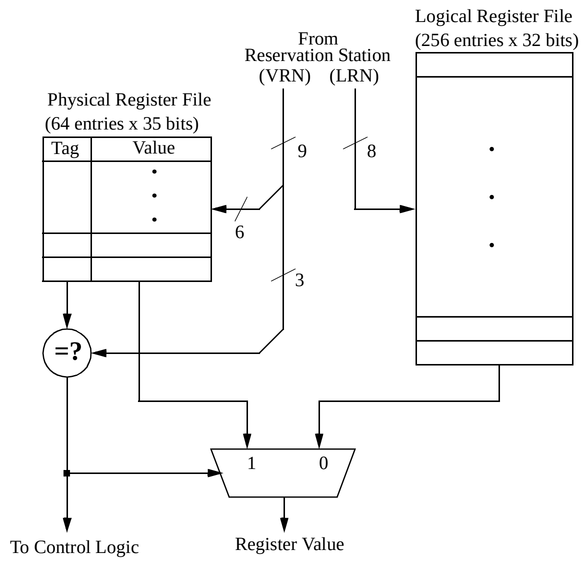

Figure 3: Mechanism to access a register value from the cache or backing store.

other register file designs. Special scheduling logic, such as exists on the Alpha 21264 to handle remote register reads, is necessary in our system to handle the timing when a cache miss occurs.

Immediately upon completion of execution, the (speculative) data is written to the physical register file. It is written to the index specified by the destination virtual register number. The check tag at that location in the PRF is also updated with the 3-bit check tag from the current instruction's virtual register number. This completes the allocation of the physical register for the current instruction. Any previous value that happened to be there is now overwritten and its value must be accessed from the LRF. We ensure that we do not overwrite a value which has not been committed yet because we require that no other in-flight instruction share the same 6 bits in its virtual register number (by the way the tags were selected). A write to the PRF always "write-allocates" its result this way and never misses the cache because it is a first-time write of a speculative value. It cannot be written to the LRF until it is proven to be on the correct execution path.

The instruction then broadcasts its VRN to each reservation station entry to indicate that the value is ready so that other instructions can be readied for execution (as in conventional systems). The physical register file is updated and the result is forwarded immediately to any consumers that require it. The virtual register number allocation algorithm ensures that the instruction retains its physical register at least until it commits to architected state.

Finally, the result of the instruction is carried down the pipeline into the reorder buffer. If this were not done, then the PRF would need more read ports in order to read out values at the time they are committed to the LRF.

When the instruction reaches the head of the instruction window and is able to commit, its value is written to the LRF (architected state) and the instruction is officially committed. The physical register is marked as free for use by later instructions. This is done by resetting the bit in the physical register free vector. The value in the physical register file, however, remains until it is absolutely necessary to overwrite it. This means that later consumers can read from the physical register for some time until the physical register is allocated to some other instruction. This eviction scheme is not simply "least recently defined" but is instead determined by when the next instruction that needs that register completes execution.

Virtual register numbers are released in the same manner as rename registers are released in a conventional, R10K style processor [19]. The virtual register number for logical register R1, for example, can be released when another virtual register number is assigned to R1 (at the next definition of R1) and that definition is committed. We call this the *free-at-remap-commit* condition. Virtual register numbers have no associated storage, so we can specify as many as needed in order to avoid stalling issue due to lack of them. No early release mechanism is considered.

Because the LRF maintains precise architected register state between each instruction in the program, recovery from exceptions and branch mispredictions is simple. Instructions which are younger than the excepting instruction are cleared from the machine. Older instructions are retained. The logical to virtual mappings are maintained as in previous work [19]. No entries from the physical register file need to be destroyed because any consumers that would have consumed bogus values have been cleared from the machine, and the bogus values will just be overwritten at some future point anyway. This has the advantage that useful values are retained in the physical file (even those that have already committed); i.e. the cache is not destroyed even for a mispredicted branch. Were this not the case, then the LRF would have to supply all values initially after a branch misprediction; this would be slow because the LRF will not have very many ports.

4. ADVANTAGES AND DISADVANTAGES

This design has many advantages, some of which are found in previous work but integrated here into one system. These advantages all arise from the main features of the design, namely the split LRF and PRF, the large set of virtual register numbers, and the way virtual numbers are mapped to physical registers using a simple indexing scheme (which pre-allocates physical registers). The advantages will be discussed in these categories.

4.1. Advantages of Split LRF and PRF

The split design allows the LRF to be large while keeping the physical file small. This is the key to the cache-like behavior of the PRF. At any point in time the LRF contains precise architected state, making state maintenance easy, particular at points of misspeculation or other exceptional events. The logical register file needs fewer ports because it only supplies values that have been committed to architected state and that do not still reside in the cache. It need not provide values that are supplied from the bypass paths nor those from the (smaller and faster) PRF.

The split design also extends naturally to a register windowed architecture where there can be a large number of logical registers (hundreds) but where the PRF is desired to be smaller to speed execution. This is feasible since only a subset of the logical registers (one or two windows), are ever active at any one time.

4.2. Advantages of Virtual Register Numbers

The mapping of logical register to physical register through the virtual register numbers has a number of inherent advantages. First, the approach avoids deadlock conditions that would exist in a merged register logical/physical file where NLR >= NPR. The "up-rename" from logical to virtual number has no deadlock problem. The subsequent "down-rename" does not have a deadlock problem because the split physical and logical register files allow physical registers to be freed as soon as their values are committed, rather than waiting for future instructions to enter the machine.

The use of VRNs also mean that dependency tracking is separated from physical value storage. Virtual numbers are used to track dependencies while a separate PRF contains the values. This advantage was first proposed in previous work [3, 4, 10]. The PRF is sized according to the capacity of the machine, independent of the size of the architected register file.

Virtual registers can be allocated in any order, and the order that is selected can implement a caching policy by keeping certain values in the physical register file longer after they commit than other values. This means that a trade-off can be made between machine capacity and value caching. In other words, some physical registers can be tied down to specific logical values so that reads can be satisfied out of the cache. This reduces the number of physical registers available for renaming but the increased PRF hit rate may more than outweigh this problem, given that mispredictions and other non-idealities effectively reduce the exploitable window size anyway. Such a trade-off would obviously be more applicable to a machine with a large number of physical registers, i.e. a machine that can more easily afford such a reduction in physical registers.

4.3. Advantages of Direct-Mapped PRF

The simple mapping scheme from virtual to physical register number also has a number of advantages. Extra cycles are not required to access the physical register file. Previous work has resorted to using a fully-associative physical register file, which we believe is not feasible [2]. The only disadvantage of this approach is that the physical register selection mechanism in the renaming pipeline stage is somewhat complicated since it needs to make sure that it assigns a virtual register whose physical register is free. Thus it needs to make a lookup in both the virtual register free list and the physical register free list.

Additionally, physical registers are pre-allocated as soon as the instruction enters the machine to avoid complex mechanisms to steal registers or reserve registers when machine capacity is overrun [3, 4, 10].

Pre-allocation of physical registers is performed without overwriting the previous value assigned to that physical register. Actual allocation is not performed until the instruction completes and writes back to the physical register file. This provides the opportunity for older values to remain in the cache, satisfying consuming instructions faster than had the value been forced to reside exclusively in the LRF after its own commit. The late allocation feature of our approach also reduces the pressure on the physical file.

Physical registers can be freed as soon as the value contained is committed to architected state, unlike in other renaming configurations where a later write to the same logical register is required in order to free the register (like in the free-at-remap-commit mechanism). Releasing a physical register is decoupled from the number of consumer instructions in the processor because a physical register can be freed and the consumer can still access the value from the LRF. Previous techniques hang on to the physical registers longer than this [11, 3] and thus could reduce the number of in-flight instructions because of lack of available registers.

Even though the physical registers can be freed early, the PRF can retain committed values until they are overwritten. This means that, for example, branch mispredictions can be handled nicely because values produced before the mispredict could still be valid.

4.4. Disadvantages

There are a number of features of the design which could be problematic. We list them in this section.

1. The number of write ports on the LRF must match the commit bandwidth of the machine in order to keep up with the machine's instruction graduation rate.
2. The RAT has as many entries as the (large) LRF. It must have enough read and write ports to keep pace with instruction decode. Each entry is only 9 bits wide and the structure is direct mapped, which somewhat simplifies the task of making it fast.
3. Our scheme always writes the value to the physical register file upon completion instead of selectively caching it as in previous work [2]. Values cannot be written to the backing store (LRF) until the instruction is committed and we do not consider any way to avoid caching values which are used on the bypass network. We leave this for future work.
4. Before instruction execution can complete, the 3-bit physical register check tag must be compared against the 3 upper bits in the source VRN specifier. If the bits match, then the value requested is the one actually residing in the PRF and execution can proceed uninterrupted. If there is not a match, then an extra lookup must be made into the LRF to get the value, which was previously committed. This adds extra complexity

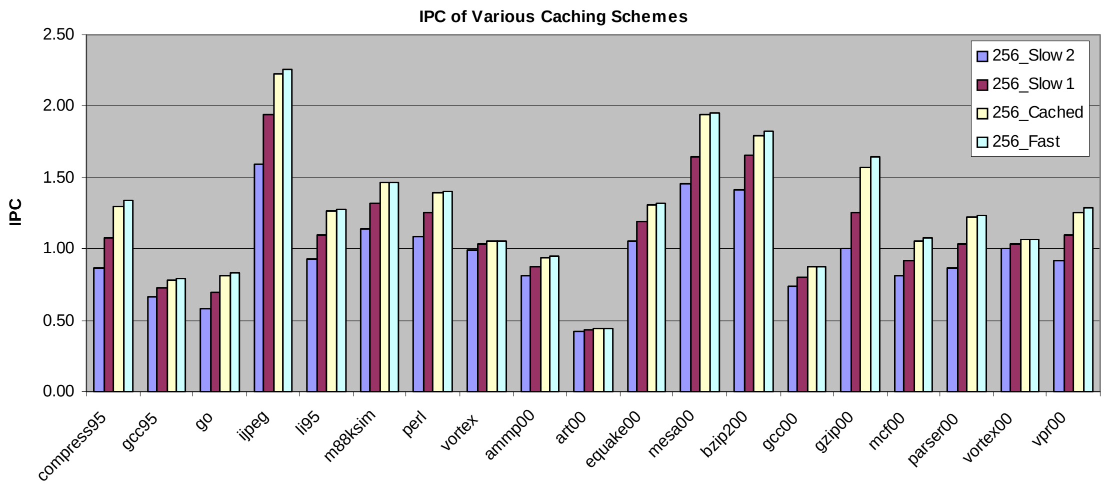

Figure 4: IPC of the four register configurations studied.

to the supply of operands to instructions, but the check can be made in parallel with instruction execution.

5. EXPERIMENTAL EVALUATION

5.1. Experiment Setup

All the benchmarks used in this study were compiled with the MIRV C compiler. We ran variants of the SPEC training inputs in order to keep simulation time reasonable. A description of MIRV, our compilation methodology, and benchmark inputs is presented in our technical report [13].

All simulations were done using the SimpleScalar 3.0/PISA simulation toolset [1]. We have modified the toolset (simulators, assembler, and disassembler) to support up to 256 registers. Registers 0-31 are used as defined in the MIPS System V ABI [18] in order to maintain compatibility with pre-compiled libraries. Registers 32-255 are used either as additional registers for global variables or additional registers for local caller/callee save variables.

All simulations were run on a subset of the SPEC95 and SPEC2000 binaries compiled with inlining, register promotion, global variable register allocation, and other aggressive optimizations for a machine with 256 registers, half set aside for locals and half for globals.

We have implemented our register caching scheme on a variant of the `sim-outorder` simulator. The rename logic is duplicated for the integer and floating point files, so each of the descriptions below applies for each. The register cache simulator is a stand-alone module which is hooked in several places into the `sim-outorder` simulator.

Table 3 lists the register cache configurations used in our simulations. The latency numbers in the tables are the additional delay (on top of whatever delay is simulated in the `sim-outorder` simulator). All simulations use the register cache code with different parameters to simulate the various configurations of interest. The cached simulation is the one of interest in this work. It has a 256-entry LRF coupled with a 64-entry PRF. The LRF has an additional 1 cycle penalty for accessing it. The PRF has no additional penalty. The other three configurations (Fast, Slow1, and Slow2) are provided for comparison purposes. These are *simulated* with a 512-entry PRF so that the VRN to PR mapping is direct, with no check bits, and an appropriate latency. In effect, each of these three models always hits in the PRF and never needs to use the LRF. “Fast” is a model of a 256 logical register file machine which has no extra delay to access the registers. “Slow1” and “Slow2” are similar but add 1 and 2 extra cycles, respectively, to the register access to simulate the slower register file.

The remainder of the simulation parameters are common across all simulations. The machine simulated is a 4-issue superscalar with 16KB caches; the parameters are used from `sim-outorder` defaults except for two integer multipliers and 64 RUU entries. The default parameters are described in our technical report [13]. The initial simulations assume an infinite number of ports on each register file. Section 5.5 explores port constraints.

Table 3: Register cache simulation configurations

Name	NLR	LR Lat	NVR	NPR	PR Lat
Fast	256	0	512	512	0
Cached	256	1	512	64	0
Slow1	256	0	512	512	1
Slow2	256	0	512	512	2

5.2. Results

This section describes the results of simulating our configurations. Figure 4 shows the IPC for the 4 register configurations mentioned above. The cached configuration approaches the performance of the fast configuration in most cases. The Fast configuration averages about 12% faster than the Slow1 while the cached configuration is 11% faster. Thus the caching strategy is able to recover most of the performance of the fastest configuration while maintaining a small physical register file.

The first numeric column of Table 4 shows the hit rate of the integer-side physical register file. In every case, the hit rate is 80% to 95%. Even the small cache is able to capture most of the register activity. This is due to effects that have been reported previously, namely that values are very frequently consumed very shortly after they are produced. The hit rate includes all values that would be bypassed or provided by the physical register file: in any case, these are register references that do not have to access the large LRF, so we deem them to be “hits.” Even if 50 out of 100 values are provided by bypasses, 30 to 45 of the remaining 50 are provided by our cache–which is still a 60 to 90% hit rate. The floating point hit rate is a bit higher, 85% to 95%.

The register cache allows values to hang around until they are overwritten by some later instruction; while the later instruction has pre-allocated the register, the old value remains. This is advantageous because the physical register is occupied with useful data for a longer period of time than if the value were “erased” as soon as it was committed to the LRF. All of the simulations presented so far use this policy, which we call *delayed allocation*. We simulated the `go` benchmark with and without this policy to model what would happen if registers were allocated in a manner similar to

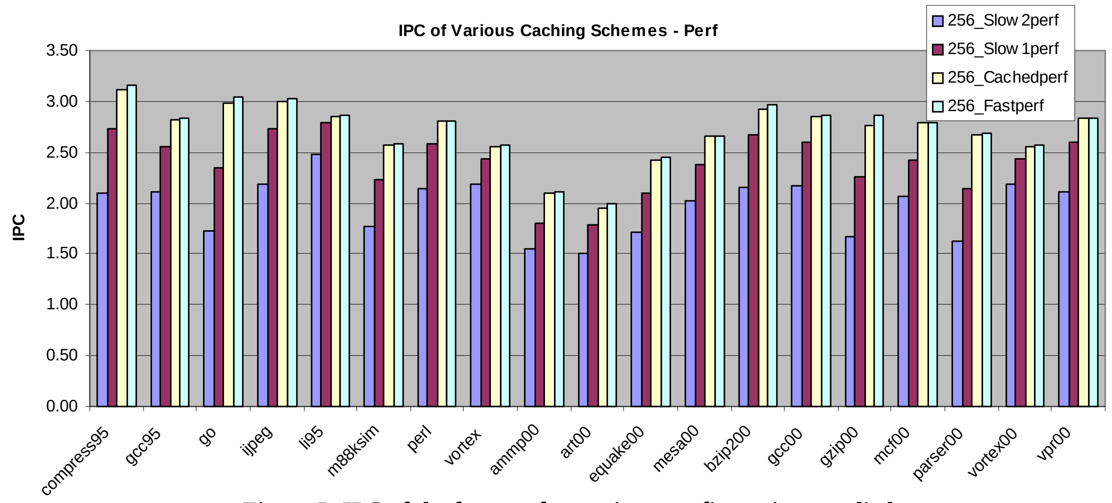

Figure 5: IPC of the four perfect register configurations studied

Table 4: Hit rate of the integer physical register file (cache) in both normal and perfect configurations.

Benchmark	Hit Rate	Hit Rate (perf)
compress95	82%	83%
gcc95	91%	91%
go	82%	83%
ijpeg	90%	90%
li95	96%	95%
m88ksim	84%	83%
perl	93%	93%
vortex	91%	91%
ammp00	85%	86%
art00	60%	47%
equake00	85%	86%
mesa00	88%	88%
bzip200	82%	82%
gcc00	90%	90%
gzip00	86%	86%
mcf00	83%	82%
parser00	91%	91%
vortex00	91%	91%
vpr00	87%	88%

current superscalar processors. This modification extends the physical register busy time on the front end (before execution) because it makes the register busy until the value is produced, even though no useful value resides in that register. Turning off delayed allocation increases the useless occupation time of the physical register.

We found that when the delayed allocation was turned off, the hit rate of the integer PRF decreased by about 2.6%. There was also a slight decrease in IPC. This shows the delayed allocation policy is making a difference though it is very slight. The primary reason for this is that most values are consumed quickly and the extra time the value hangs around in the cache is not very profitable. It is probably easier to allow delayed allocation than to aggressively kill values out of the register file, and since the latter has no performance benefit.

On the back end (after commit), our scheme releases the register as soon as the commit is completed. We cannot extend the register busy time until the commit of the next value writing the particular architected register because we would run out of physical registers (that is, as we said before, we cannot use a merged renaming approach like the R10000).

5.3. Perfect Prediction, Caching, and TLBs

The absolute performance of the configurations in the previous section is somewhat attenuated by the heavy penalty of branch prediction and cache misses. This section removes those constraints by simulating all the configurations with perfect caches, perfect branch prediction, and perfect TLBs (denoted by the “perf” suffix on the configuration names in the following graph). Our goal is to determine what happens to the caching scheme as improvements are made in other areas of the system. The figures in this section show the results.

Figure 5 shows the IPC of the four perfect configurations. It can be seen that the IPCs have increased from the 1-1.5 range up to the 2-3 range. The performance of `go`, for example, has tripled. More interesting than the absolute performance increases produced by the more ideal system is the trend in performance from the Slow2 to Fast configurations of the register cache. Since a number of the performance-attenuating features have been eliminated from the microarchitecture, the gap between Slow2 and Fast has increased. For example, whereas in the non-perfect simulations Fast was 44% faster than Slow2, the perfect Fast configuration is 76% faster than Slow2. This points out the (somewhat obvious) conclusion that as the other bottlenecks are removed from the system, the register configuration makes a significant difference.

The second numeric column of Table 4 shows the hit rate of the integer physical register file for the perfect configuration. These and the floating point numbers are essentially unchanged from the non-perfect simulations, showing that the mechanism is robust under different configurations. The only significant difference is found in the `art` benchmark, where it attains a 47% hit rate as opposed to a 60% hit rate in the imperfect configuration.

5.4. Varying Cache Size

In our design, the PRF capacity is intimately related to the other parameters in the machine, such that changing it requires changing a number of other parameters as well. The cache size should be determined by the desired machine capacity and performance. For this reason and because of the large number of simulations that would be required to examine a variety of cache sizes for all of our benchmarks, we have limited the discussion to the `go` benchmark. For each simulation, we changed the size of the cache (for the cached configurations) and reduced the size of the instruc-

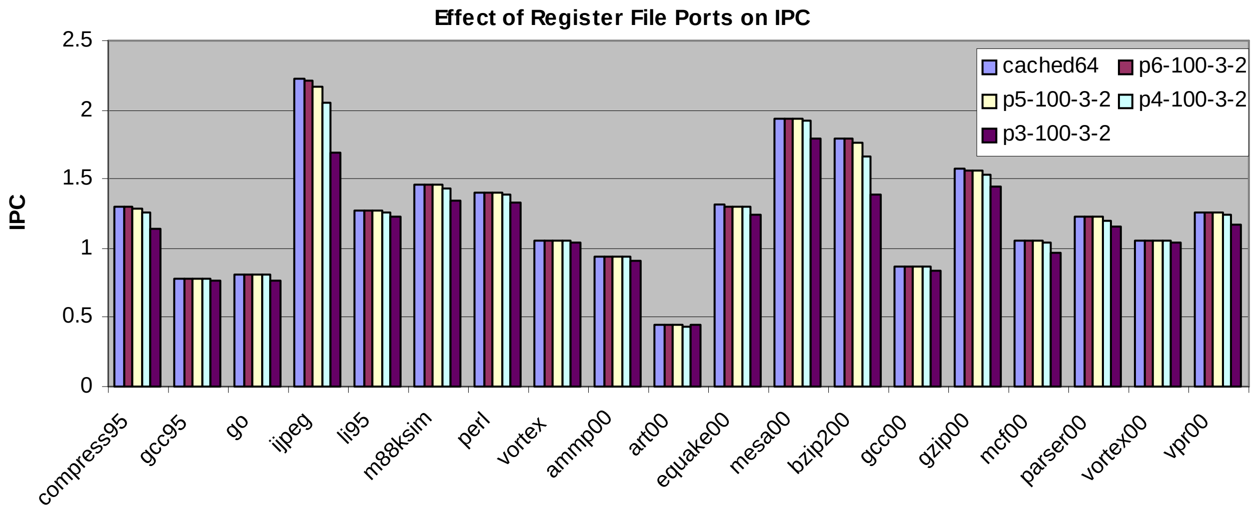

Figure 6: The IPC of limited port configurations for SPEC2000.

tion window (for all the configurations) to match the capacity of the cache.

We simulated five cache sizes: 8, 16, 32, 64, and 128 entries, and correspondingly-sized instruction windows. Generally speaking, the Fast configurations are always better than the Cached configurations which are in turn better than the Slow1, etc. However, there are a couple of exceptions. The cached8 configuration is slightly slower than the slow1-128, and the same as the slow1-64 configuration. This shows that more slow registers and a larger instruction window is better than too few. Similarly, the fast8 configuration is slower than all cached configurations except cached8. Fast16 is much better. The indication in both of these exceptional cases is that an 8-register cache and window size is simply insufficient to provide the best performance on `go`. The PRF hit rate trends upward as the cache increases in size, from 65% up to 85%. From the perspective of experimental design, this data does not tell us much because too many parameters in the machine are changed: it is difficult to determ ine the effect that each has on overall performance. The tight integration of our caching model with the rest of the superscalar hardware makes it impossible to untangle these different parameters.

5.5. Varying Available Register Ports

The results above assume an infinite number of ports to both register files. In this section we demonstrate what happens with several different configurations of ports on the register files.

For these simulations, the number of read and write ports on each register file restrict the flexibility of several portions of the pipeline model. The read ports of both the LRF and PRF guide the instruction scheduler so that it does not issue more instructions in a cycle than there are register ports to supply operands. The scheduler is optimistic in that it examines all ready instructions and starts selecting them for execution as long as the number of read ports from the appropriate register file is not exceeded. It continues through all ready instructions, perhaps skipping some that cannot be issued due to high port utilization, until it issues as many as it can, up to the issue width limit. The LRF write ports are used in the commit stage of the pipeline, where if an instruction cannot commit because of lack of LRF write ports, commit is stopped and the remainder of the retiring instructions must be committed the next cycle (or later). The PRF write ports are used in the writeback stage of the pipeline to write results from the function units. Our simulator assumes that the PRF must be able to sustain the writeback bandwidth of as many instructions that can complete per cycle. Therefore we do not restrict the PRF write ports.

There are minimum port requirements on each of the register files. Both the PRF and LRF must have at least three read ports each, since our simulator will only read the sources for an instruction in a single given cycle, and there are some instructions with three source operands. This could be solved by taking several cycles to read the operands, but we deemed it not necessary to simulate fewer than 3 read ports since most PRF designs should have at least that many. Similarly, the simulator requires at least 2 write ports on each register file since some instructions have two destination registers.

Figure 6 shows the results for all of the benchmarks studied in this paper. The configurations are labeled with 4 numbers: the number of PRF read and write ports and the number of LRF read and write ports, respectively. Since we did not model limited write ports on the PRF, we set them to 100. There is usually not much performance degradation going from infinite ports to the minimum number of ports on the LRF; however reducing the number of ports on the PRF to the minimum does affect performance. From the infinite port configuration to the most limited, performance is reduced 2% up to 24%. The `ijpeg` and `bzip` benchmarks perform the worst with the limited port configuration. This is not surprising since those two benchmarks have the highest average port requirements (simulation results not shown). The `art` benchmark produces the only unexpected result. This has been a difficult benchmark through all of the studies because it has such terrible overall performance. This is due to the very high data cache miss rates–the L1 data cache misses 42% of the time; the unified L2 cache misses 48% of the time. These misses cause major backups in the instruction window, so that it is full over 90% of the time. The most limited cache configuration slightly changes the order that instructions are run from the earlier configurations and thus it is not surprising that there is a small perturbation in the performance; in this case it is in the upward direction. The IPC is low enough that the PRF is easily able to sustain the average requirements (1/2 an instruction per cycle can easily be accommodated by 3 read ports on the PRF).

This data demonstrates that our technique is not hampered by a limited number of ports on the LRF. This is because data values are produced and then consumed shortly thereafter so that the cache or bypassing logic can supply the values. Furthermore, the out-of-order engine can tolerate the extra cycle incurred by a PRF miss. The PRF port limited studies show that performance does not really begin to degrade until the number of read ports is reduced to 5 or 4. In addition, we investigated the `go` benchmark over a larger number of port configurations. Perfomance did not degrade until the number of read ports on the PRF was reduced below 5.

6. COMPARISON TO PREVIOUS WORK

The register cache is a hardware controlled mechanism for making using of temporal locality of register reference [20, 21]. The register file is organized as a hierarchy with the operands supplied from the uppermost (smallest and fastest) level. The lower levels constitute backing storage for the full set of registers, not all of which will simultaneously fit into the small upper level. Motion between files is performed by the hardware based on recent usage patterns.

This strategy is used in a recent proposal, called the "multiple-banked register file," where a mutli-level caching structure with special caching and prefetching policies is used to implement a large number of physical registers [2]. This work attempts to cache the physical registers of a dynamically renamed microprocessor in the context of a merged-file renaming mechanism, despite the seeming lack of locality in the physical register reference stream (because of the "random" selection of physical registers from the free list).

In that work, all values produced by the function units are written to the physical register backing store; some of them are written to the cache as well based on caching policies. There is no dirty-writeback path from the cache to the backing store, so all values produced by the function units must be written to the backing store during instruction writeback. In our work, only committed values need to be written to the logical register file. This will require somewhat less bandwidth than the previous approach since the number of instructions committed is less than the number written back.

Another disadvantage of the multiple-banked research is that the physical register file, though small at 16 entries, requires a fully associative lookup on all ports. Our work eliminates this inefficiency by using a clever virtual-to-physical register indexing scheme to allow the physical file to be direct mapped.

Hierarchical register files have also been proposed to allow efficient implementation of large logical register sets. The register file is separated into several regions, each in turn containing more registers and having slower access time than the previous region [17]. Placement of data is performed by the compiler with respect to frequency of access while motion between files is explicitly coded by the compiler.

The problem of large register files has been addressed in commercial machines such as the Alpha 21264, which split its physical register file into two copies, with a 1-cycle delay for updates from one copy to another [5].

Other work attempts to reduce the number of physical registers required for a given instruction window size so that caching techniques will not be necessary. One example is the virtual-physical register research which makes use of the observation that physical register lifetimes do not begin until instruction completion, so that storage need not be allocated until late in the pipeline [3, 4, 10]. The delayed allocation of physical registers introduces a potential deadlock where there may not be a physical register available by the time an instruction commits. This is corrected by reserving a number of physical registers for the oldest instructions and sending younger instructions back for later re-execution if they try to use one of the reserved registers. The technique was later changed to handle this deadlock better by implementing a physical register stealing approach [10]. Our approach avoids this deadlock by separating the logical register file from the physical register file.

Our work is like the virtual-physical register approach in that we actually allocate the physical register at instruction writeback, though pre-allocation happens earlier. This proposal also frees the physical register as soon as the instruction commits, which is the earliest that any technique can do so. No merged file mechanism can do this because the value storage must be retained until it is certain that the value will never be needed again. Once the physical register is freed and the value is placed in the logical file, any future consumers can access it from there. Even though the register is freed, the value can sit in the physical register until some other writer destroys it. This allows the value to sit in the cache for some time after commit. Some previous work has considered earlier deallocation of physical registers by using dead value information which exploits the fact that the last use of a register can be used for a deallocation marker instead of waiting for the next redefinition [9, 7].

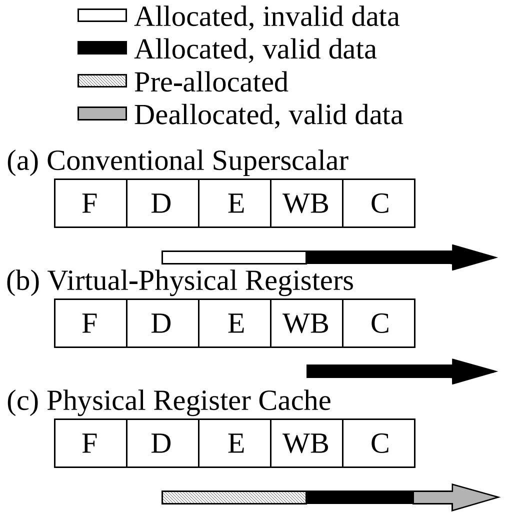

Figure 7: The lifetimes of physical registers in various schemes

Figure 7 shows these differences in pictorial form. The clear bar represents regions where the physical register is allocated but does not contain a valid value. The black bar shows where the physical register is allocated and must contain valid data (until a new value is committed to architected state). The checked bar shows the region where the physical register is pre-allocated but does not contain data for the present instruction; it may contain valid data from an older instruction. Finally the shaded arrow represents the region where the physical register is free to be allocated to another instruction, yet it contains valid data from the previous producer instruction, which consumers are free to use. Therefore, the deallocate region overlaps the pre-allocate region for a later instruction that will use the same physical storage location.

Our work differs from previous research in that it proposes to use the physical register file itself as a cache for a large logical file using a new register-renaming technique. Previous work is mainly concerned with implementation of large physical register files whereas we are mainly interested in implementing a large logical register file.

The rename storage in previous superscalar designs could be considered as a cache for the logical register file. However, if the rename storage is larger than the architected storage, as it is in many modern superscalar processors, the "cache" is bigger than the backing store. In any case, our system is designed specifically to cache a large set of registers provided in the ISA to the compiler.

7. CONCLUSIONS

We have presented an implementation of a large and fast logical register file by integrating register renaming with a physical register file smaller than the logical one. This physical register file serves as a cache for the logical register file as well as storage for in-flight instructions. The renaming scheme is unique in several ways. First, the physical register file is smaller than the logical file. Second, the renaming scheme renames the logical registers to physical registers through an intermediate set of virtual registers. Third, the mapping function is constrained in such a way as to ensure that the physical register file is direct-mapped instead of fully associative as in previous approaches. This technique avoids the register-release deadlock problem and also deadlock problems of earlier virtual tagging schemes which had to reserve or steal registers to ensure that the program makes forward progress. The caching mechanism provides an improvement of up to 20% in IPC over an un-cached large logical register file with conventional register renaming. The hit rate of the physical register file on most benchmarks is 80% or better.

The caching mechanism proposed here can be extended in a number of directions. First, the proposal is amenable to modifications to effect caching policies on the physical registers through careful allocation of virtual registers. Second, the caching mechanism is a natural fit to be integrated with register windows for a SPARC-like architecture. Another way our approach could be used is to build an inexpensive superscalar implementation of a conventional (32-register) logical file with an even smaller number of physical registers, say 16, while using a direct-indexed physical file instead of the associative ROB lookups used in earlier designs.

Finally, this approach could be useful on simultaneous multithreaded processors which require very large logical register files to house the contents of the multiple thread contexts that are simultaneously live in the machine. Previous research has used a merged register renaming scheme [7], which means that the physical register file (which contains both architected and speculative state) must be extremely large. For example, for 4 threads at 32 registers each, the PRF would need to be larger than 128, and in particular it would be 128 plus the maximum number of in-flight instructions. Our technique could be used to implement a much smaller physical register file.

8. REFERENCES

[1] Douglas C. Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. University of Wisconsin, Madison Tech. Report. June, 1997.

[2] Jose-Lorenzo Cruz, Antonio Gonzalez, Mateo Valero and Nigel P. Topham. Multiple-Banked Register File Architectures. Proc. 27th Intl. Symp. Computer Architecture, pp. 316-325, June 2000.

[3] Antonio Gonzalez, Mateo Valero, Jose Gonzalez and T. Monreal. Virtual Registers. Proc. Intl. Conf. High-Performance Computing, pp. 364-369, 1997.

[4] Antonio Gonzalez, Jose Gonzalez and Mateo Valero. Virtual-Physical Registers. Proc. 4th Intl. Symp. High-Performance Computer Architecture (HPCA-4), pp. 175-184, Feb. 1998.

[5] Linley Gwenapp. Digital 21264 Sets New Standard. Microprocessor Report, Vol. 10, No. 14. October 28, 1996, pp. 11-16.

[6] Intel IA-64 Application Developer’s Architecture Guide. May 1999. Order Number: 245188-001. Available at http://developer.intel.com/design/ia64/devinfo.htm.

[7] Jack L. Lo, Sujay S. Parekh, Susan J. Eggers, Heny M. Levy, Dean M. Tullsen. Software-Directed Register Deallocation for Simultaneous Multithreaded Processors. IEEE Transactions on Parallel and Distributed Systems, Vol. 10, No. 9, September 1999, pp. 922-933.

[8] Luis A. Lozano C. and Guang R. Gao. Exploiting Short-Lived Variables in Superscalar Processors. Proc. 28th Intl. Symp. Microarchitecture, pp. 292-302, Nov. 1995.

[9] Milo M. Martin, Amir Roth, and Charles N. Fischer. Exploiting Dead Value Information. Proc. 30th Intl. Symp. Microarchitecture (MICRO’97), Dec. 1997.

[10] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez and V. Vinals. Delaying Physical Register Allocation through Virtual-Physical Registers. Proc. 32nd Intl. Symp. Microarchitecture, pp. 186-192, Nov. 1999.

[11] M. Moudgill, K. Pingali and S. Vassiliadis. Register Renaming and Dynamic Speculation: An Alternative Approach. Proc. 26th Intl. Symp. Microarchitecture (MICRO’93), pp. 202-213, Dec. 1993.

[12] David A. Patterson and Carlo H. Sequin. RISC I: A Reduced Instruction Set VLSI Computer. Proc. 8th Intl. Symp. Computer Architecture, Vol. 32 No. CS-93-63, pp. 443-457. Nov. 1981.

[13] Matthew Postiff, David Greene, Charles Lefurgy, Dave Helder, Trevor Mudge. The MIRV SimpleScalar/PISA Compiler. University of Michigan CSE Technical Report CSE-TR-421-00, April 2000. Available at [22].

[14] Matthew Postiff, David Greene, and Trevor Mudge. Exploiting Large Register Files in General Purpose Code. University of Michigan Technical Report CSE-TR-434-00, October 2000. Available at [22].

[15] Matthew Postiff. Compiler and Microarchitecture Mechanisms for Exploiting Registers to Improve Memory Performance. Ph.D. Dissertation, University of Michigan. March 2001.

[16] Dezso Sima. The Design Space of Register Renaming Techniques. IEEE Micro, Vol. 20 No. 5, pp. 70-83. Sep/Oct 2000.

[17] John A. Swenson and Yale N. Patt. Hierarchical Registers for Scientific Computers. Proc. Intl. Conf. Supercomputing, pp. 346-353, July 1988.

[18] UNIX System Laboratories Inc. System V Application Binary Interface: MIPS Processor Supplement. Unix Press/Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[19] Kenneth C. Yeager. The MIPS R10000 superscalar microprocessor. IEEE Micro, Vol. 16 No. 2, pp. 28-40. April, 1996.

[20] Robert Yung and Neil C. Wilhelm. Caching Processor General Registers. Intl. Conf. Computer Design, pp. 307-312, Oct, 1995.

[21] Robert Yung and Neil C. Wilhelm. Caching Processor General Registers. Sun Microsystems Laboratories Tech. Report. June, 1995.

[22] http://www.eecs.umich.edu/mirv.