Appears in MICRO-38, November 2005

How to Fake 1000 Registers

David W. Oehmke[†], Nathan L. Binkert, Trevor Mudge, Steven K. Reinhardt

Advanced Computer Architecture Lab
University of Michigan, Ann Arbor
{doehmke, binkertn, tnm, stever}@eecs.umich.edu

Abstract

Large numbers of logical registers can improve performance by allowing fast access to multiple subroutine contexts (register windows) and multiple thread contexts (multithreading). Support for both of these together requires a multiplicative number of registers that quickly becomes prohibitive. We overcome this limitation with the virtual context architecture (VCA), a new register-file architecture that virtualizes logical register contexts. VCA works by treating the physical registers as a cache of a much larger memory-mapped logical register space. Complete contexts, whether activation records or threads, are no longer required to reside in their entirety in the physical register file. A VCA implementation of register windows on a single-threaded machine reduces data cache accesses by 20%, providing the same performance as a conventional machine while requiring one fewer cache port. Using VCA to support multithreading enables a four-thread machine to use half as many physical registers without a significant performance loss. VCA naturally extends to support both multithreading and register windows, providing higher performance with significantly fewer registers than a conventional machine.

1. Introduction

Twenty-five years ago architects expected VLSI trends to enable processors with thousands of registers, and they devised schemes such as register windows and multithreading to take advantage of this resource [25]. Unfortunately, the realities of wire delay and power consumption, along with the large number of ports needed to support wide-issue superscalar pipelines, made these extremely large register files impractical. Nevertheless, those original ideas have merit. This paper shows how architects can achieve the benefits of large logical register files using a standard physical register file.

Registers are a central component of both instruction-set architectures (ISAs) and processor microarchitectures. From the ISA's perspective, a small register namespace allows the encoding of multiple operands in an instruction of reasonable size. Registers also provide a simple, unambiguous specification of data dependences, because—unlike memory locations—they are specified directly in the instruction and cannot be aliased. From the microarchitecture's point of view, registers comprise a set of high-speed, high-bandwidth storage locations that are integrated into the datapath more tightly than a data cache, and are thus far more capable of keeping up with a modern superscalar execution core.

As with many architectural features, these two purposes of registers can conflict. For example, the dependence specification encoded by the register assignment is adequate for the ISA's sequential execution semantics. However, out-of-order execution requires that these logical registers be renamed into an alternate, larger physical register space to eliminate false dependencies.

This paper addresses a different conflict between the abstraction of registers and its implementation: that of *context*. A logical register identifier is meaningful only in the context of a particular procedure instance (activation record) in a particular thread of execution. From the ISA's perspective, the processor supports exactly one context at any point in time. However, a processor designer may wish for an implementation to support multiple contexts for several reasons: to support multithreading, to reduce context switch overhead, or to reduce procedure call/return overhead (e.g., using register windows) [15, 27, 29, 5, 20, 25]. Conventional designs require that each active context be present in its entirety; thus each additional context adds directly to the size of the register file. Unfortunately, larger register files are inherently slower to access, leading to a slower

† Currently at Cray Inc.

cycle time or additional cycles of register access latency, either of which reduces overall performance. This problem is further compounded by the additional rename registers necessary to support out-of-order execution.

We seek to bypass this trade-off between multiple context support and register file size by decoupling the logical register requirements of active contexts from the contents of the physical register file. Just as caches and virtual memory allow a processor to give the illusion of numerous multi-gigabyte address spaces with an average access time approaching that of several kilobytes of SRAM, we propose an architecture that gives the illusion of numerous active contexts with an average access time approaching that of a conventionally sized register file. Our design treats the physical register file as a cache of a practically unlimited memory-backed logical register space. We call this scheme the *virtual context architecture* (VCA).

In VCA, an individual instruction needs only its source operands and its destination register to be present in the register file to execute. Inactive register values are automatically saved to memory as needed, and restored to the register file on demand. The architecture modifies the rename stage of the pipeline to trigger the movement of register values between the physical register file and the data cache. Furthermore, a thread can change its register context simply by changing a base pointer—either to another register window on a call or return, or to an entirely different software thread context. Compared to prior proposals (see Section 5), VCA:

- unifies support for both multiple independent threads and register windowing within each thread;
- completely decouples the physical register file size from the number of logical registers by using memory as a backing store;
- enables the physical register file to hold just the most active subset of logical register values, instead of the complete register contexts, by allowing the hardware to spill and fill registers on demand;
- is backward compatible with existing ISAs at the application level for multithreaded contexts, and requires only minimal ISA changes for register windowing;
- requires no changes to the physical register file design and the performance-critical schedule/execute/writeback loop;
- is orthogonal to the other common techniques for reducing the latency of the register file—register banking and register caching;
- does not involve speculation or prediction, avoiding the need for recovery mechanisms.

The virtual context architecture enables a near optimal implementation of register windows, improving performance and greatly reducing traffic to the data cache (by up to 10% and 20%, in our simulations). VCA naturally supports simultaneous multithreading (SMT) as well, allowing large numbers of threads to be multiplexed on relatively small physical register files. Our results show that a VCA SMT machine can achieve speedups comparable to a conventional SMT implementation with twice as many registers.

Implementing both register windows and multithreading in the same processor requires a multiplicative number of contexts. The one such machine of which we are aware (Sun's Niagara [12]) has 640 registers per core—even though it is an in-order pipeline, it does not require renaming registers. VCA provides unified support for both techniques in an out-of-order pipeline without the multiplicative increase in register count normally required. We show that VCA can achieve near-peak performance on an out-of-order pipeline using register windows and four threads (as in Niagara) with only 192 physical registers.

We describe VCA in more detail in Section 2. Section 3 discusses our simulation-based evaluation methodology. Section 4 evaluates a VCA-based implementations of register windows, SMT, and a combination of both techniques. Section 5 presents previous work. Section 6 concludes and discusses future work.

2. The Virtual Context Architecture

This section presents the virtual context architecture in two phases. First, we outline the basic changes to a conventional out-of-order processor required to support VCA. Second, we describe a handful of implementation optimizations that make VCA more efficient.

2.1. Basic VCA implementation

The fundamental operation of VCA lies in mapping logical registers to physical registers. VCA builds on the register renaming logic already found in a modern dynamically scheduled processor. A key feature of VCA is that it has minimal impact outside of the rename stage. Figure 1 illustrates the blocks that are changed. We describe the basic operation of VCA by first discussing the VCA renaming process, then detailing the states and state transitions of physical registers. We then discuss the implications for branch recovery, and the requirements for supporting multithreading and register windows.

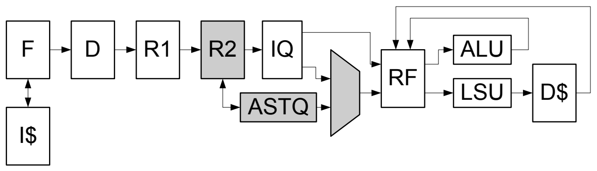

Figure 1: Pipeline diagram.
Shaded regions depict pipeline changes for VCA.
R2 represents the extra rename stage necessary.
ASTQ is the architectural state transfer queue, described in Section 2.2.2.

2.1.1. VCA renaming. Renaming in VCA is a two-stage process. First, the source and destination register indices in the machine instruction are combined with the thread's context base pointer to generate memory addresses. In the simplest case, each register index is simply added to the base pointer to form a logical register memory address.

The second stage of renaming maps the logical register memory addresses to physical registers. Because VCA maps registers from a large, sparse space, the rename table must have tags like a cache or TLB. To avoid deadlock, the rename table must be able to concurrently map all of an instruction's source operands; thus the table must either guarantee that no two registers from the same context can conflict or have an associativity at least equal to the maximum number of source register operands (typically two). Although higher associativity provides higher performance by reducing rename-table conflicts, we find that a four-way set associative table provides good performance and use this configuration in our simulations. Section 2.2 describes further rename-table optimizations that allow a significant reduction in the number of tag bits required for each entry.

The key difference between VCA and a conventional architecture is that the rename table lookup may miss, i.e., there may be no physical register mapping for the desired logical register. For destination registers, a miss is not a problem, as the previous value of the logical register is not needed. The lookup of the destination register is only to determine which physical register, if any, to mark as free when the current instruction is committed. For sources, however, the logical register must be allocated to a free physical register and the register value must be read in from memory. We call this operation a *fill*.

If there are no free physical registers, a physical register already allocated to a different logical register must be freed. If the value in the physical register is modified with respect to the logical register's memory location, the value must first be written back to memory; we call this operation a *spill*. (Further details of physical register allocation are discussed in the following section.)

In the simplest incarnation, fills and spills can be accomplished by injecting load and store operations, respectively, into the processor's instruction queue. These operations can use the same register ports, datapaths, and cache ports as regular load and store instructions, but are simpler in some ways. Because the full memory address is generated in rename, the execution stage need not read a base address register or perform an effective address calculation. The operations are not entered into the reorder buffer and do not go through the commit stage because they are not part of the actual program flow.

Once physical registers have been identified for the instruction's source and destination values, the instruction is dispatched to the instruction queue and reorder buffer using these physical register indices, as in a conventional machine. By creating appropriate dependences between any required spills and fills and the instruction itself, this dispatch need not wait for the spills and fills to complete. The instruction then passes through remainder of the pipeline (schedule, execute, writeback, and commit) just as it would in a conventional out-of-order processor.

2.1.2. VCA physical register states. This section provides a more thorough description of VCA operation by detailing the possible physical register states and state transitions. Because the VCA register file serves as both a rename register file and a cache, physical register management combines aspects of conventional register free-list management and cache replacement.

Conceptually, VCA associates four pieces of state with each physical register: the corresponding logical register memory address (if any), a reference count, a *committed* bit (C), and a *dirty* bit (D). The reference count tracks the number of uses of the register by instructions in the execution portion of the pipeline, as in previous work [18, 1]. A register's reference count is incremented when an instruction using the register passes through rename, and decremented when the instruction commits or is flushed due to an exception or misspeculation. The committed bit, if set, indicates that the register holds a valid, non-speculative register value. A set dirty bit indicates that the value is more up-to-date than the corresponding memory address, and needs to be written back before being replaced.

Physical registers with non-zero reference counts are considered *pinned* (P) and cannot be reallocated to other logical registers. Pinning guarantees that all the physical register indices an instruction needs to exe-

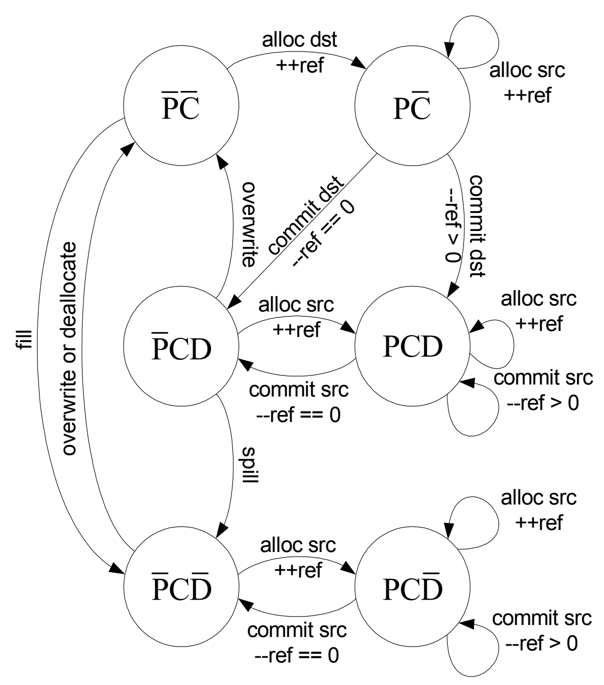

Figure 2: Physical register state machine.

cute will remain valid from the time the instruction receives them at rename until that instruction commits.

The complete register state diagram is shown in Figure 2. All physical registers start out in the unpinned, uncommitted state ($\overline{P}\overline{C}$), which we refer to as *free*. In this state (and this state only), a physical register has no association with any logical register. A physical register leaves the free state in one of two ways. It may be allocated as a destination register for a new instruction, in which case it is associated with the logical register's memory location and becomes pinned (the reference count is incremented) but remains uncommitted (state $P\overline{C}$). As long as the value is uncommitted, it should not be written back, so the value of the dirty bit is irrelevant. When the producer instruction commits, the physical register's value becomes the committed value for the corresponding logical register. The committed and dirty bits are both set. The instruction may remain pinned or become unpinned (state PCD or $\overline{P}CD$) depending on whether other instructions in the pipeline are referencing the register as a source operand.

A free register may also be allocated to hold an existing logical register value that is needed as a source operand but is not already in the physical register file. In this case, a fill operation will bring the value in from memory. The reference count will also be incremented, placing the register in the $PC\overline{D}$ state. For clarity, Figure 2 shows this transition as requiring two steps (through state $\overline{P}C\overline{D}$), but from the implementation perspective it is atomic.

Committed values remain allocated to physical registers even after they become unpinned (i.e., their reference counts decrement to zero), so that the values are available to future instructions without requiring a fill. These cached unpinned values correspond to states $\overline{P}CD$ and $\overline{P}C\overline{D}$ (depending on whether the value was generated by an instruction or brought in by a fill, respectively). This behavior provides the physical register file's caching effect. These committed registers can become free in two ways. First, the logical register being cached can be overwritten when a later instruction with the same logical destination commits. This transition corresponds to the freeing of a physical register in a conventional speculative out-of-order processor. Second, the register can be reallocated to a different logical register. This transition corresponds to a cache replacement. When there are no free registers available, the rename stage selects an unpinned register for replacement using an LRU algorithm. If the selected register is dirty, it must be spilled before it can be reallocated. To avoid spilling physical registers that are about to be overwritten, registers for which an overwriting instruction has been dispatched are given the lowest priority for replacement. From the implementation's perspective, a reallocation takes a physical register directly from $\overline{P}CD$ or $\overline{P}C\overline{D}$ to $PC\overline{D}$ or $P\overline{C}$ (depending on whether the register is reallocated as a source or a destination). For clarity, Figure 2 breaks these transitions down into multiple independent transitions passing through the free state ($\overline{P}\overline{C}$).

The rename stage may be forced to stall if there are no free or unpinned registers. Forward progress is guaranteed, however, because all instructions that have passed rename are guaranteed to be able to commit (due to register pinning). As these instructions commit, they will decrement the register reference counts, eventually unpinning physical registers. (If rename stalls indefinitely, all physical registers will be unpinned once all renamed instructions have committed.)

2.1.3. Branch recovery. In processors with merged physical register files, the processor must revert its register map to an earlier state when a misspeculated branch is encountered. Some processors, such as the Compaq Alpha 21264 [9], checkpoint the rename table at each branch and restore the appropriate checkpoint on a misprediction. Others, such as the Intel Pentium 4, maintain a separate rename table that is updated as instructions commit, thus tracking the machine's committed architectural state. This scheme recovers from a mispredicted branch at the head of the reorder buffer

(ROB) by copying the commit-stage rename table to the rename stage. The processor need not wait for the mispredicted branch to reach the head of the ROB: on a detected misprediction, the processor can iterate from the head of the ROB up to the mispredicted branch, updating the commit-stage table in the process, to generate the appropriate rename table state. Due to the size and complexity of VCA's rename table, the checkpointing approach is likely to be infeasible. We use the Pentium 4 approach to provide misspeculation recovery in VCA.

2.1.4. Multithreading support. A conventional pipeline requires that each thread have its own rename table and sufficient physical registers to hold its complete architectural state. VCA requires neither of these; it requires only that each thread have a separate context base pointer. A machine instruction's register index combined with the thread's base pointer creates a global memory address that uniquely identifies the logical register across all contexts. The rename table takes these global addresses as input, and thus a single table suffices for all threads. A large number of threads could create more conflicts in the rename table, so a heavily multithreaded machine could require a larger or more associative rename table for optimum performance, but the minimum configuration for functional correctness is unchanged.

Because VCA requires only the registers referenced by in-flight instructions to be in the physical register file, there is also no requirement to increase the physical register size when increasing the level of multithreading (though again, performance concerns may make a larger register file more desirable).

2.1.5. Register window support. The logical register context for a thread can be modified instantaneously by updating its context base pointer value. Because the rename map and all in-flight instructions use memory addresses to identify logical registers, this update does not require a pipeline flush or any other state update or synchronization. Using the basic VCA as described thus far to support fully windowed registers for procedure calls and returns is as simple as allocating a stack of contexts and updating the base pointer appropriately on calls and returns. Updating the base pointer by less than the total size of the architectural register context provides overlapping windows for parameter passing.

Commercial ISAs that support register windows (i.e., SPARC and IA-64) partition the logical registers into windowed and non-windowed subsets. Registers in the latter set do not change on procedure calls and returns, and are used to hold global constants. Supporting this partitioning on VCA requires that each thread have two base pointers, one for each class of registers. The base pointer for windowed registers is updated on procedure calls and returns, while the base pointer for non-windowed registers is modified only on context switches. The architectural register index selects the appropriate base pointer before being summed to form the logical register address.

2.2. VCA optimizations

The virtual context architecture as described in the previous section is complete and functionally correct. However, the straightforward implementation detailed there suffers from some potential inefficiencies. This section discusses two optimizations we apply to the base VCA. First, we add a level of indirection to the rename table to reduce the size of that structure. Second, we add a new structure to the pipeline, the architectural state transfer queue (ASTQ), to handle spill and fill operations more efficiently.

2.2.1. Rename table. VCA requires a cache-like rename table with tags to map arbitrary addresses to physical registers. Adding a full address tag to each entry in the table significantly increases its size. Because the rename table will likely be replicated to implement the required number of ports, this size increase may be costly. We can take advantage of register address locality to drastically reduce the tag size with only a modest increase in complexity. Although any memory address could be used as a base pointer, in practice only a relatively small number of addresses will be used as base pointers within a given period of time. Register memory addresses also exhibit spatial locality around each base pointer address.

To exploit these characteristics, we introduce an additional translation table that maps the upper bits of each register memory address to a much smaller *register space identifier (RSID)*. The concatenation of the RSID and the remaining low-order memory address bits (the *register space offset*) are then used for the rename table lookup. Figure 3 illustrates this process. In the example shown there, the upper 48 bits of the 64-bit register memory address are mapped to a 4-bit RSID. As a result, the tag on each rename table entry is only 11 bits rather than the 55 bits required without RSIDs. In this case, the translation table could be implemented with a small 16-entry fully associative buffer where the RSID is simply the index of the matching entry.

If a register memory address does not find a match in the RSID table, it must allocate a new table entry,

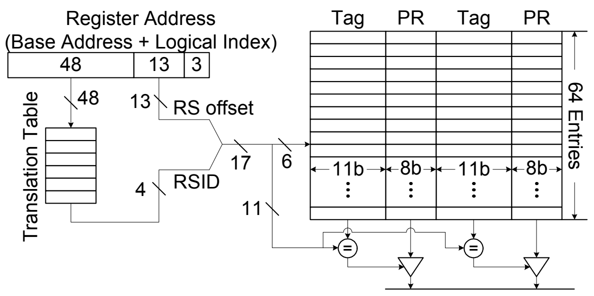

Figure 3: Optimized rename table.
(See Section 2.2.1.)

potentially replacing an existing valid table entry. In this (rare) situation, any physical registers using the current RSID must be flushed to memory before the RSID can be reused. It may be desirable to associate reference counters with each RSID so that unused RSIDs can be identified and reused without requiring a flush operation. RSIDs can be managed in hardware (or low-level software, such as PAL code), and thus the very existence of RSIDs is hidden from the operating system and user-level code.

We can reduce the overhead of the translation process by caching the RSID associated with each base pointer, accessing the RSID table only when a base pointer is updated (due to a context switch or a register-window call or return). This optimization requires a modest alignment restriction on the base pointer so that a logical register offset cannot generate a memory address outside the range of the current RSID. A similar scheme could be used to cache the physical addresses associated with each RSID, eliminating the need for TLB lookups (and the possibility of page faults) on spill and fill operations.

Note that the optimal configuration of the RSID scheme (both the number of RSIDs and the division of the register memory address bits between the RSID table lookup and the register space offset) will depend on the expected usage model. The number of RSIDs should clearly be at least as large as the number of base pointers, and perhaps larger to enable caching of registers across context switches. For a system that does not support register windows, the register space offset need only be large enough to map a single logical register file, while a register-windowed machine would perform best with larger register spaces that can map the working set of an active register stack.

2.2.2. ASTQ. The basic VCA implementation described above inserts standard load and store operations into the pipeline to implement fills and spills, respectively. However, fill and spill operations have simpler requirements than program loads and stores, so they can be supported by a streamlined hardware path. Adding simpler, dedicated resources for fills and spills allows them to bypass the instruction queue and load/store queue, preserving these more precious resources for regular program instructions.

There are three key differences between fill/spill and load/store operations. First, fills and spills do not require effective address calculation; fill addresses are calculated in rename, and spill addresses are obtained via table lookup. Second, fills and spills do not require memory disambiguation with respect to program loads and stores. VCA does not provide coherence between the register file and backing memory, so the system provides no guarantees on the results of regular program accesses to the memory-mapped register space. Third, fills and spills do not have data dependences on regular instructions. A regular instruction may have a dependence on a fill, but fill/spill addresses do not come from registers (as mentioned) and spills are never generated unless the target physical register has no outstanding references. Spills are always ready to execute as soon as they are generated in rename, and the only dependence a fill may have is on a prior spill that is freeing its physical register target.

As a result, spill and fill operations can be scheduled independently from all other program instructions using a simple FIFO buffer. We call this buffer the *architectural state transfer queue (ASTQ)*. The location of the ASTQ in the pipeline is illustrated in Figure 1. ASTQ operations share issue ports with memory operations so that additional register file and cache ports are not required. If there are fewer ready load or store operations than memory issue ports, the entry at the head of the ASTQ (if any) is issued to a free port. We found that only four entries are required in the ASTQ to provide maximum benefit.

3. Methodology

We used simulation to compare VCA to a conventional architecture. We begin with common features, then discuss details related specifically to register windows and SMT, respectively.

We simulated VCA using M5, a detailed execution-driven architecture simulator [3]. We used an aggressive but realistic four-issue superscalar processor as our baseline. Table 1 lists the simulated baseline system parameters. We chose a short pipeline depth (8 cycles) to exaggerate the impact of the added rename cycle we assume for VCA (see Figure 1). We also chose a conservative 3-cycle data cache hit latency to reflect realistic spill and fill overheads.

Table 1: Baseline processor parameters.

Machine Width	4
Instruction Queue	128
Reorder Buffer	192
Pipeline depth (fetch to exec)	8 cycles
DL1 Cache Ports	2 R/W
DL1 Cache Size	64K 4-Way 3 cycle hit
IL1 Cache Size	64K 4-Way 1 cycle hit
L2 Cache Size	1M 4-Way 15 cycle hit
Memory Latency	250 cycles
Branch Predictor	Hybrid

The size of the VCA rename table varies based on the number of threads supported. The table is set-associative with 64 entries per way, with associativity of 3, 5, or 6 (192, 320, or 384 entries) for one, two, and four threads, respectively. The VCA rename table has only 8 ports (compared with 12 on the baseline) to further reduce its complexity. Reads of the same register are combined and use a single port. At most two spill or fill operations each cycle can be written into the ASTQ, which has four entries. If the rename table or ASTQ ports are exhausted, one or more instructions will be delayed in rename until the following cycle.

We evaluated the designs using the SPEC CPU2000 benchmarks [10] (except for the four Fortran 90 benchmarks, which the GNU compiler suite could not compile). The benchmarks were compiled with gcc 3.3.3 at -O3 optimization, which includes function inlining. We generated the best single SimPoint [24] for each binary running the reference input. In the case of benchmarks with more than one input, the input closest to the average IPC of all the inputs was selected. Each program was simulated for 100 million instructions after a 5 million instruction warm-up period.

3.1. Register windows

Comparing executions both with and without register windows requires extra effort. Windowed registers visibly change the ISA semantics, requiring recompilation to eliminate the stores and loads used to explicitly spill and fill registers on a non-windowed machine. Furthermore, the elimination of these loads and stores changes the program path length, meaning that IPC is no longer a valid metric for comparison.

Because our simulation environment supports only the Alpha ISA [26], we defined a variant of Alpha that includes windowed registers. To maintain compatibility with pre-existing binaries, we define any register used to communicate values across a function call (in either direction) is treated as non-windowed (does not change). All other registers are treated as windowed (changes on function calls and returns). We overload the call and return instructions to allocate and deallocate register windows respectively. We modified the GNU compiler suite (gcc 3.3.3) [8] to support the new architecture and ABI. The GNU standard C library (glibc 2.3.2) was also recompiled using the new ABI for linking with the windowed binaries.

Table 2: Path length ratio
(register window to baseline).

Benchmark	Ratio	Benchmark	Ratio
bzip2_graphic	0.92	twolf	0.99
crafty	0.93	vortex_2	0.82
eon_rushmeier	0.94	vpr_route	0.90
gap	0.91	ammp (FP)	0.98
gcc_expr	0.92	equake (FP)	0.94
gzip_graphic	0.92	mesa (FP)	0.92
parser	0.92	wupwise (FP)	0.93
perlbmk_535	0.85	**Average**	**0.92**

To deal with the differing instruction path length between the windowed and non-windowed binaries, we measured the number of instructions required to execute both versions of each benchmark to completion using fast functional simulation. We estimate execution time as the product of the CPI from the detailed SimPoint simulation and the complete benchmark's dynamic instruction count. Total cache accesses are calculated similarly, by multiplying the rate at which the cache is accessed (per committed instruction) by the total dynamic instruction count. The ratio between the path lengths is shown in Table 2. Because register windows only affect performance when there is a reasonably high frequency of function calls, we use only those benchmarks that make a function call at least once every 500 instructions.

We simulated the windowed binaries from execution points corresponding to the SimPoints in the baseline binaries. We identified these by counting dynamic conditional control instructions, which remained constant (within 0.001%) between the windowed and non-windowed binaries.

3.2. Simultaneous multithreading

We used a scheme similar to that of Raasch [22] to generate representative multi-thread workloads. We simulated all the 253 possible two-thread workloads using the baseline architecture. We then generated a vector of 14 statistics (IPC, cache miss rate, etc.) for each workload and ran a linkage-based clustering algorithm to identify workloads with similar characteristics

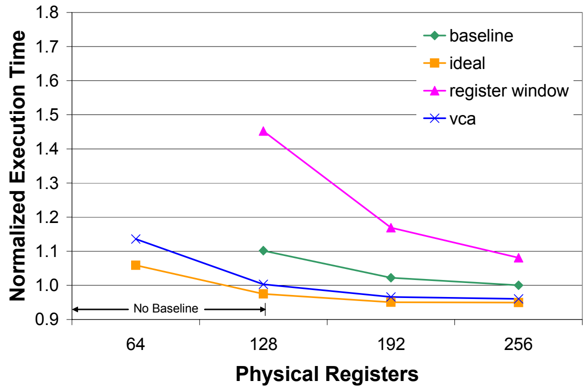

Figure 4: Register window execution time.
Normalized to baseline with 256 phys. regs.

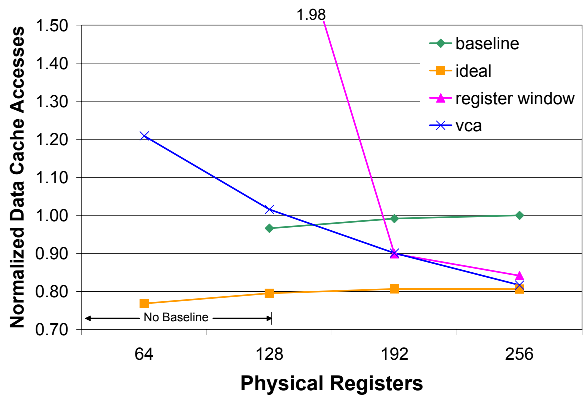

Figure 5: Register window cache accesses.
Normalized to baseline with 256 phys. regs.

(after reducing the dimensionality using principle components analysis). We identified 43 workload clusters, and selected the workload nearest the centroid of each cluster for our experiments. We repeated this process on all pairs of two-thread workloads to generate a set of 127 four-thread workloads.

Each workload is warmed up until one thread reached 5 million instructions, then run until one thread commits 100 million instructions. Two statistics are used to measure performance. The first is weighted execution time. This metric is calculated by summing the relative execution time of all threads—the execution time of each thread in the SMT workload, divided by the execution time of the same benchmark running as a single thread. The execution time was calculated by multiplying the cycles per instruction (CPI) by the dynamic path length of the benchmark. The second statistic is weighted cache accesses. It is calculated similar to weighted speedup, but using data cache accesses per instruction instead of execution time.

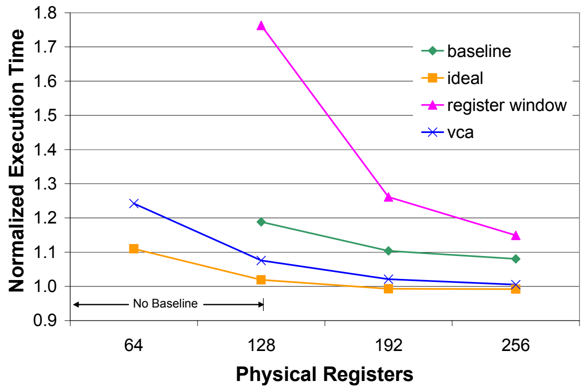

Figure 6: Single cache port execution time.
Normalized to dual-port baseline
with 256 phys. regs.

4. Results

We present three sets of results. First, we examine the performance of register windows and SMT individually, each implemented using VCA. We then look at the behavior of combining both SMT and register windows on VCA.

4.1. Register windows

This section compares register windows implemented using the virtual context architecture to three other architectures. We perform this comparison across various physical register file sizes. We start with 64 physical registers, equal to the number of architectural registers—32 integer and 32 FP. We go up to 256 registers, equal to the number of architectural registers plus the size of the reorder buffer. Performance does not improve beyond this point as the reorder buffer fills before any additional physical registers can be used.

In addition to our non-windowed baseline, we model a conventional register window implementation in which the number of logical registers is expanded to hold multiple contiguous register windows—specifically, the maximum number of windows that can be fit in the physical register file while leaving at least 64 rename registers available. When a register window overflow or underflow occurs, the pipeline delays for 10 cycles to model the time needed for trapping to the operating system's overflow/underflow handler. After the delay, load or store instructions are inserted into the pipeline to either fill a new window on an underflow, or to save all the dirty registers in the departing window on an overflow.

We also include results for an idealized register-window implementation. This model provides an lower bound on execution time by handling spills and fills instantaneously and without accessing the data cache.

Figure 4 presents execution times, normalized to the baseline architecture with 256 physical registers. Note that the baseline architecture cannot run with only 64 physical registers; there are 64 architectural registers, so this size does not provide any rename registers needed for out-of-order execution.

VCA performs very close to the ideal architecture—within 1% with 256 physical registers. With fewer physical registers, VCA is forced to generate many more spills and fills, increasing the gap between it and ideal.

VCA with register windows provides a performance improvement over the baseline non-windowed architecture at all register file sizes. VCA's performance advantage increases as the number of physical registers decreases, from 4% at 256 registers to 9% at 128 registers. In the conventional architecture, fewer physical registers means fewer rename registers, which reduces the effective instruction scheduling window size. In contrast, VCA can move infrequently used architectural state out to memory to free more physical registers for renaming.

By eliminating the loads and stores needed to fill and spill registers, register windows not only reduce the number of instructions executed but also the number of data cache accesses. Figure 5 presents the number of data cache accesses generated by the four architectures over the same range of physical register file sizes. The results are again normalized to the baseline binary with 256 physical registers.

Note that some architectures experience more data cache accesses as the number of physical registers is decreased, while others experience the opposite effect. In all cases, having fewer physical registers throttles misspeculation, resulting in a slight decrease in cache accesses. For the baseline and ideal register window architectures, only explicit loads and stores in the binary generate non-speculative data cache accesses, so the misspeculation effect is the only source of variation.

For VCA and the conventional register window architecture, smaller physical register files cause significant increases in window fills and spills, dwarfing the impact of reduced misspeculation. Note that this traffic increases at a much slower rate with VCA than with the conventional window architecture. The conventional register-window scheme saves and restores entire windows on each overflow or underflow—including dead and unused registers—while VCA spills and fills values to memory at the granularity of individual registers. Even when the conventional register window architecture provides nearly the same data cache access savings as VCA, the performance of VCA is much better. The incremental single-register spills and fills generated by VCA have much less effect on pipeline performance than the bursty sequences of loads and stores generated by the conventional implementation. VCA also avoids the underflow/overflow trap overhead seen in conventional register window implementations.

We can take advantage of VCA's reduced cache traffic by reducing the number of data-cache ports, saving cache area, latency, and power. Figure 6 shows the execution time of our four architectures modified to have only a single data-cache port, normalized to the two-port baseline architecture. The results are qualitatively similar to Figure 4. However, VCA's lower cache traffic provides an even larger performance advantage on this pipeline, especially at 256 physical registers. The VCA configuration is almost 7% faster than the baseline in the single-port pipeline versus only 4% faster in the dual-port pipeline. Furthermore, at 256 registers, the single-port VCA provides effectively the same performance as the dual-port baseline (0.5% slowdown). Comparing the VCA data points in Figure 6 with the baseline data points in Figure 4, we see that with 128 registers, VCA with one cache port is nearly 2.5% faster than the dual-port baseline.

4.2. Simultaneous multithreading

This section examines the effectiveness of VCA in supporting multiple thread contexts for SMT (without register windows). Figure 7 presents weighted speedups for two- and four-thread workloads on both VCA and a conventional SMT architecture. The speedups are relative to single-threaded execution on the baseline architecture with 256 physical registers.

For these runs, the number of physical registers is varied from 64 to 448. For the two-thread runs, there is no performance benefit past 320 physical registers, since this size is large enough to hold two copies of the architectural state (128 regs) plus one rename register for each ROB entry (192 regs). In addition, the conventional architecture cannot operate unless the number of physical registers is strictly greater than the number of architectural registers needed (128 for two threads or 256 for four threads).

The virtual context architecture is able to maintain its performance with a much smaller physical register file size. With two threads and 192 physical registers, VCA is able to achieve 97% of the performance of the baseline with a full set of 320 physical registers, while the baseline only achieves 88% of this level. The four

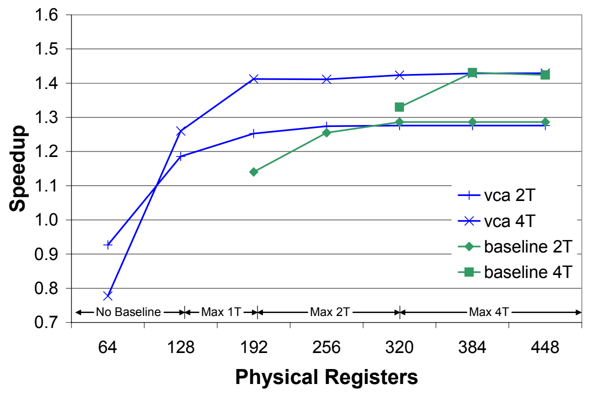

Figure 7: SMT performance.
Weighted speedup vs. single-thread baseline with 256 phys. regs.

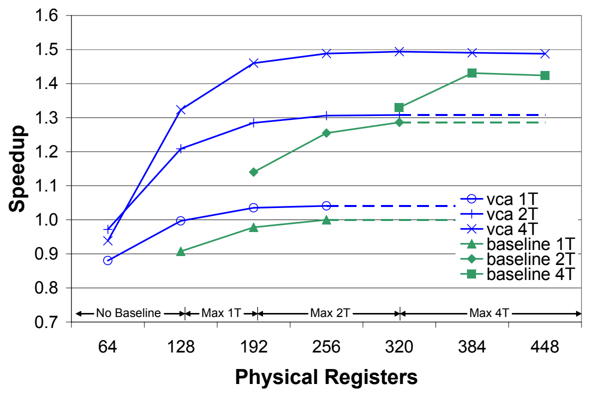

Figure 8: SMT + register window performance.
Weighted speedup vs. single-thread baseline with 256 phys. regs.

thread results are even more persuasive. With 192 physical registers, VCA is able to achieve a 98.7% of the performance of the baseline with a full set of 448 physical registers. VCA achieves this speedup with fewer physical registers than logical registers, a point where the conventional architecture is unable to operate. The baseline requires double the number of physical registers to achieve a comparable speedup.

4.3. SMT with register windows

In a conventional architecture, register windows and SMT each require a significant increase in the size of the physical register file. Worse yet, combining the two techniques has a multiplicative impact. This section shows that VCA can easily accommodate both SMT and register windows at the same time with a conventionally sized physical register file.

Figure 8 presents weighted speedups for VCA and the baseline architecture with one, two, and four threads. As before, speedups are relative to the single-thread baseline with 256 physical registers. As in the previous section, the ROB size places an upper bound on the effective register file size for one and two threads, while the conventional architecture cannot operate without more physical than logical registers.

By combining the efficiencies of register windows and SMT, while enabling SMT on smaller register files, the virtual context architecture provides a higher speedup at every size of physical register than the baseline architecture. In fact, VCA achieves 98% of its peak performance using four threads on only 192 registers. At this same register file size, the conventional architecture can support only two threads, resulting in 22% lower speedup.

The ability of the virtual context architecture to support more threads on a smaller register file comes at the cost of additional data cache accesses. A key benefit from combining register windows and SMT is that the reduction in cache accesses from register windows counters the increase due to SMT. For example, although the four-thread non-windowed VCA with 192 physical registers provides 98% of the performance of the 448-register baseline, it requires 24% more cache accesses. Adding register windows to the same VCA configuration reduces its cache accesses by 23%, resulting in 5% fewer cache accesses than the non-windowed baseline.

5. Related work

The idea of using memory to provide a backing store to the register file has been explored before. Ditzel et al. [7] describe the C machine stack cache, where the register file is replaced by a large circular buffer that is mapped contiguously onto the stack. Huguet and Lang [11] use a table of memory addresses and compiler support to allow the hardware to perform background saves/restores. Nuth et al. [19] proposed the Named State Register File, which like VCA treats the physical register file as a cache of memory-mapped logical registers. Unlike our design, the Named State Register File requires a content-associative lookup on the entire register file on every access, which would likely impact the processor's cycle time.

Register caches [2, 21, 30] and banking [2, 6] have been proposed to improve the access latency of large register files. In these designs, the full architectural state is still kept in the register file, so context switches and register window over- and underflows still require explicit copying of many (often unused) registers to and from memory, and die area requirements continue to scale with the product of the number of thread contexts and the number of windows per context. Monreal et al. [17] improve physical register utilization by delaying the allocation of physical registers until write-back. The physical register file implementation is orthogonal to VCA, so these techniques may still be useful underneath the VCA scheme.

One of the major costs of SMT is the larger physical register file needed to accommodate additional architectural state. Earlier proposals sought to reduce this burden by dividing the logical registers among multiple threads [28, 23]. These designs require extensive compiler and operating system support.

Martin et al. [16] and Lo et al. [14] propose software identification of dead register values to reduce save/restore traffic and to free physical registers for other SMT threads, respectively. VCA achieves similar goals in part by moving dead values out of the register file into memory. VCA does not require software annotations, and also handles values that are not dead but have not been accessed recently. Dead-value annotations would be a useful addition to VCA, allowing it to avoid spilling dead values to memory and to reclaim dead registers preferentially over live but inactive ones. We hope to explore this extension in future work.

Two commercial architectures that use register windows are SPARC [29] and Itanium [5]. Overflow and underflow conditions are handled by trapping to the operating system on SPARC processors, while Itanium designs use a hardware engine. In either case, the handling of an underflow or overflow halts the execution of the program to copy entire windows, adding a significant amount of overhead.

Previous work has also proposed directing stack accesses to a separate pipeline and cache [4, 13]. These designs reduce data cache bandwidth demand, but only by diverting this demand to a separate cache.

6. Conclusions and future work

The virtual context architecture (VCA) uses a novel mapping scheme to effectively decouple the number of supported logical registers from physical register storage requirements. VCA enables the physical register file to hold just the most active subset of logical register values by spilling and filling individual registers on demand in hardware. The removal of the logical register storage requirement allows architects to support register-hungry schemes such as register windows and simultaneous multithreading, while choosing the physical register file size based on performance considerations alone.

A VCA-based implementation of register windows in an out-of-order processor reduces execution time by 4% while reducing data cache accesses by nearly 20% compared to a non-windowed machine, with an even larger performance advantage over a conventional register-window implementation. VCA's data cache traffic reduction is large enough that it can achieve the same performance with one cache port as an otherwise similar conventional machine would with two cache ports.

VCA is also able to manage thread contexts efficiently, enabling effective implementation of simultaneous multithreading (SMT) using as few as half the registers of a standard architecture. Furthermore, VCA handles thread and function contexts at the same time in a unified way, allowing SMT to be combined with register windows with no additional physical registers. As a result, a four-thread VCA machine with 192 registers can achieve higher performance than a conventional non-windowed SMT machine with twice as many registers. In comparison, Sun's Niagara [12] uses 640 registers per core to support 4 threads and register windows on an in-order machine (i.e., no renaming registers required).

While this paper only examined relatively low levels of multithreading, VCA requires negligible per-thread state (a PC and a register base pointer), so it can in principle support dozens of threads. This opportunity opens the door to efficient support for microkernel operating systems, virtual machines, and very fast interrupt and exception processing.

VCA could also benefit from novel compiler register allocation algorithms that seek to minimize the "footprint" of a thread in the logical register space. For example, compilers could re-use recently dead logical registers in preference to other logical register identifiers. VCA can exploit this locality by leaving unused logical register values in memory.

Furthermore, VCA lends itself to other applications beyond register windows and threads, such as emulation of stack-based architectures or ISAs with large logical register files. The separation of register state from the standard call stack in a VCA implementation of register windows also makes it more secure, as it is immune to stack-smashing attacks.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. CCR-0219640 and CCF-0325898. This work was also supported by gifts from Intel, an Intel Fellowship, and a Sloan Research Fellowship.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *36th Ann. Int'l Symp. on Microarchitecture*, pages 423–434, Dec. 2003.

[2] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *34th Ann. Int'l Symp. on Microarchitecture*, pages 237–248, Dec. 2001.

[3] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In *Proc. Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, Feb. 2003.

[4] S. Cho, P.-C. Yew, and G. Lee. Decoupling local variable accesses in a wide-issue superscalar processor. In *Proc. 26th Ann. Int'l Symp. on Computer Architecture*, pages 100–110, May 1999.

[5] Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual*. Santa Clara, CA, 2000.

[6] J.-L. Cruz, A. González, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *Proc. 27th Ann. Int'l Symp. on Computer Architecture*, pages 316–325, June 2000.

[7] D. R. Ditzel and H. R. McLellan. Register allocation for free: The C machine stack cache. In *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 48–56, Mar. 1982.

[8] Free Software Foundation. GNU Compiler Collection. http://gcc.gnu.org.

[9] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14):11–16, Oct. 28, 1996.

[10] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[11] M. Huguet and T. Lang. Architectural support for reduced register saving/restoring in single-window register files. *ACM Trans. Computer Systems*, 9(1):66–97, Feb. 1991.

[12] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[13] H.-H. S. Lee, M. Smelyanskiy, G. S. Tyson, and C. J. Newburn. Stack value file: Custom microarchitecture for the stack. In *Proc. 7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 5–14, Jan. 2001.

[14] J. L. Lo, S. S. Parekh, S. J. Eggers, H. M. Levy, and D. M. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Trans. Parallel and Distributed Systems*, 10(9):922–933, Sept. 1999.

[15] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), Feb. 2002.

[16] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *30th Ann. Int'l Symp. on Microarchitecture*, pages 125–135, Dec. 1997.

[17] T. Monreal, A. Gonzlez, M. Valero, J. Gonzlez, and V. Vinals. Delaying physical register allocation through virtual-physical registers. In *32nd Ann. Int'l Symp. on Microarchitecture*, pages 186–192, Nov. 1999.

[18] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of MICRO-26*, 1993.

[19] P. R. Nuth and W. J. Dally. The named-state register file: Implementation and performance. In *Proc. 1st Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 4–13, Jan. 1995.

[20] D. A. Patterson and C. H. Sequin. RISC I: A reduced instruction set VLSI computer. In *Proc. 8th Intl. Symp. Computer Architecture*, volume 32, pages 443–457, Nov. 1981.

[21] M. Postiff, D. Greene, S. Raasch, and T. N. Mudge. Integrating superscalar processor components to implement register caching. In *Proc. 2001 Int'l Conf. on Supercomputing*, pages 348–357, 2001.

[22] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on smt processors. In *Proc. 12th Ann. Int'l Conf. on Parallel Architectures and Compilation Techniques*, September 2003.

[23] J. A. Redstone, S. J. Eggers, and H. M. Levy. Mini-threads: Increasing tlp on small-scale smt processors. In *Proc. 9th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 19–30, Feb. 2003.

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proc. Tenth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 45–57, Oct. 2002.

[25] R. L. Sites. How to use 1000 registers. In *Caltech Conference on VLSI*, pages 527–532. Caltech Computer Science Dept., 1979.

[26] R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 3 edition, 1998.

[27] D. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. 23rd Ann. Int'l Symp. on Computer Architecture*, pages 191–202, May 1996.

[28] C. A. Waldspurger and W. E. Weihl. Register relocation: Flexible contexts for multithreading. In *Proc. 20th Ann. Int'l Symp. on Computer Architecture*, pages 120–130, May 1993.

[29] D. L. Weaver and T. Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[30] R. Yung and N. C. Wilhelm. Caching processor general registers. In *Proc. 1995 Int'l Conf. on Computer Design*, pages 307–312, Oct. 1995.