Appears in the Proceedings of the 31st International Symposium on Computer Architecture (ISCA-31), June 2004.

Use-Based Register Caching with Decoupled Indexing

J. Adam Butts and Gurindar S. Sohi
Computer Sciences Department, University of Wisconsin–Madison
{butts,sohi}@cs.wisc.edu

Abstract

Wide, deep pipelines need many physical registers to hold the results of in-flight instructions. Simultaneously, high clock frequencies prohibit using large register files and bypass networks without a significant performance penalty. Previously proposed techniques using register caching to reduce this penalty suffer from several problems including poor insertion and replacement decisions and the need for a fully-associative cache for good performance. We present novel mechanisms for managing and indexing register caches that address these problems using knowledge of the number of consumers of each register value.

The insertion policy reduces pollution by not caching a register value when all of its predicted consumers are satisfied by the bypass network. The replacement policy selects register cache entries with the fewest remaining uses (often zero), lowering the miss rate. We also introduce a new, general method of mapping physical registers to register cache sets that improves the performance of set-associative cache organizations by reducing conflicts. Our results indicate that a 64-entry, two-way set associative cache using these techniques outperforms multi-cycle monolithic register files and previously proposed hierarchical register files.

1. Introduction

The physical register file and associated bypass networks represent a substantial barrier to the implementation of future high-performance processor cores. Such processors are likely to be deeply pipelined (to support high clock frequencies) and multiple-issue, resulting in a large number of instructions in flight. To support precise, out-of-order execution, the results of all in-flight instructions must be maintained, demanding a large number of physical registers. The clock frequency, however, will limit the amount of storage that can be addressed in a single cycle, resulting in increased register file latency. The read latency is particularly problematic since it appears in both the branch misprediction and load-hit speculation loops [3]. Furthermore, to allow unrestricted issue of dependent operations, the total number of stages in the bypass network must increase with the register file latency. Bypass networks are dominated by long wires and wide multiplexors, which do not scale well to high frequencies. A limited bypass network [1] causes the performance impact of the register file latency to be even more severe.

One solution to this problem is to use a small *register cache* to supply the majority of values to the execution core at low latency. The problem then becomes maintaining the right values within the cache in order to maximize its hit rate, and, consequently, the overall performance. Previous register cache proposals specify policies—LRU replacement, for example—that result in poor decisions about which values should reside in the register cache. In addition, fully-associative caches have also been specified in order to mitigate the high conflict miss rate resulting from mapping a large number of register values onto a small cache structure. We have found that these schemes require unreasonably large and/or associative caches to achieve performance comparable to the slow, monolithic register file they replace.

We propose *use-based register cache management*, consisting of insertion and replacement policies that determine the usefulness of a register value in terms of the number of remaining readers of that value. The initial number of readers is determined speculatively using a degree of use predictor [5]. We exploit this information to cache only those registers with remaining uses. Our insertion policy filters from the cache register values that have bypassed to all of their expected consumers, reducing cache pollution caused by dead values. A remaining-use count, kept for each cached register value, is updated as subsequent uses are satisfied by the cache. When a replacement is necessary, the cached value with the fewest remaining uses (ideally zero) is selected as a victim, minimizing subsequent misses resulting from the replacement.

Our second major contribution is *decoupled indexing*, a new, general method for indexing set-associative register caches to reduce conflicts. Rather than allowing the cache set to be determined in an ad hoc manner from the physical register identifier, decoupled indexing assigns a set to each value according to a policy designed to minimize conflicts. This assignment occurs as part of the normal rename process; in addition to receiving a physical register, each architectural register is also renamed to a register cache set. We evaluate several index assignment policies, including some that exploit the predicted use information. Decoupled indexing reduces conflict misses by 40% in a two-way set-associative cache, accounting for 62% of the performance benefit of using a four-way cache.

In the next section, we motivate register caching with a discussion of physical register lifetimes. We present previously proposed register file optimizations in this context and select a general register caching framework upon which our policies are built. Section 3 presents use-based insertion and replacement policies. Decoupled indexing is presented in Section 4. We perform an evaluation of our techniques in Section 5 before concluding in Section 6.

2. Register Storage Hierarchies

Due to the central nature of the physical register file in contemporary architectures, much research has been aimed at optimizing this structure, particularly with respect to its access time. A large portion of this research focuses on banking a monolithic register file, dividing its bandwidth (and sometimes capacity) requirements among several smaller structures and/or reducing the number of ports to each bank. This work is generally orthogonal to our own: many of those schemes are equally applicable to a register cache as a register file. Instead, we focus on prior work on register file hierarchies.

Hierarchical (or multi-level) register file schemes all provide for different classes of registers as a means of reducing the average register read latency. Typically, a small number of registers are available at a low latency while a larger number are available at a longer latency. The methods differ on such attributes as the structure of the register hierarchy, whether inclusion is enforced, and, most importantly, how values are managed within the hierarchy. Some of these schemes depend on explicit software assignment of values to levels in the hierarchy [11, 13, 16]. Such schemes not only necessitate compiler support, but also require exposing the register file implementation to the ISA.

Hardware-managed register hierarchies exploit the fact that a register's contents are used for a brief portion of its total lifetime. The lifetime of a physical register may be divided into three phases, shown in Figure 1. The first phase, or *empty time*, exists between the allocation of a physical register to an instruction and the writing of a value into the register. The relatively short *live time* begins when the register is written and ends with the last use of the result by a consumer instruction. The *dead time* comprises the time from the last use until the register is freed; during this time, the processor can not guarantee that no additional uses will occur, so the value must be maintained.

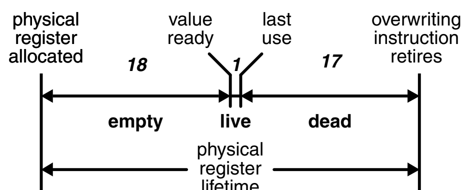

Figure 1. Lifetime of a physical register divided into three phases. The number above each phase represents the average of the per-benchmark median lengths of that phase in clock cycles determined by timing simulation (see Section 5.1).

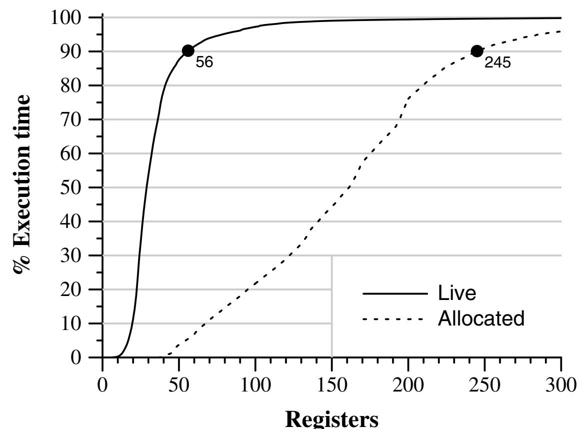

Figure 2. Comparison of allocated and live registers. Cumulative distributions over the sequential execution of the SPECint 2000 benchmarks showing physical registers allocated and those containing live values. The 90th percentile point is labeled.

An important observation is that *values are only read during their live time*. Prior to this point, the value does not exist, and afterwards, it is not read. Given the short duration of the physical register live time compared with its full lifetime, the number of values that may be read at any time should be small compared with the number of physical registers. Figure 2 compares cumulative distributions of the number of simultaneously live values and the number of allocated physical registers in an aggressive out-of-order processor (details in Section 5.1). The median number of live values is less than 20% of the number of allocated physical registers. These results also show that 90% of the time, 56 storage locations are sufficient to contain all live values. The various hardware-managed schemes may be differentiated by what phase of the register lifetime they attack to achieve this reduction in the number of registers required.

2.1. Reducing value storage lifetimes

González et al. proposed virtual-physical registers to avoid the allocation of a storage location until the value is available [7]. This delay eliminates the register empty time, allowing the physical register file to be smaller. Unfortunately, there are complications in translating from a virtual-physical register to an actual physical register (required to read the value) and in avoiding deadlock from overcommitting the actual physical registers. Because no additional structure keeps values for recovery purposes, this scheme is not strictly a hierarchical register file; however, it does illustrate the reduction in register file size possible by eliminating the register empty time.

Balasubramonian et al. take a different approach, choosing instead to target the physical register dead time [2]. Registers determined to be dead are moved

from the main (L1) register file to a backup (L2) register file, reducing the capacity demands on the L1 file. A register is eligible for transfer when it has no pending (renamed but unexecuted) consumers and its architectural register has been reassigned. This latter requirement reduces the fraction of the register dead time that can be eliminated. Control mis-speculations and exceptions require some L1 registers to be recovered from the L2. To avoid high recovery penalties, values are moved only when the number of free L1 registers falls below a threshold. This scheme requires substantial extra hardware for tracking the locations of different values and complex recovery procedures in the event of a mis-speculation.

The short average register live time indicates the existence of temporal locality in register accesses, suggesting that a cache would be beneficial. By storing a subset of register values, a register cache [3, 6, 15] can avoid both the empty time and dead time associated with physical registers. As register cache entries are allocated when they are written, they do not have any empty time. Also, the natural replacements that occur as new values are written reduces the dead time. By storing values during their live times only, a small, fast cache can still supply most of the values required for execution. Register caching schemes, then, may be differentiated by how they determine the set of live values. The next section covers register caching in more detail, presenting a framework that encompasses our scheme and two previous register caching proposals.

2.2. Register cache framework

Figure 3 depicts a pipeline diagram illustrating the relationships among dependent instructions from issue through writeback in a machine with a register cache and a two-cycle register file. Operands are communicated through either the bypass network (dotted lines) or storage (solid lines). The bypass network supplies values until they are available from storage (register file or cache). Without a cache, the bypass network would need more stages to cover the longer register file read and write latencies; otherwise, issue restrictions would be necessary to ensure that no instructions required a value not available from either the register file or bypass network.

Adding a register cache reduces the read latency for most instructions (I1-I4a). The register cache takes the place of the register file, providing the access bandwidth required by the execution core. Each instruction implicitly assumes that its inputs reside in the register cache, which is accessed in the cycle after an instruction is issued (e.g., cycle 3 for I2). Register cache writes occur immediately after execution (e.g., cycle 5 for I2). Also, note that register cache entries must be invalidated when their corresponding physical registers are freed to prevent incorrect values from being supplied.

Live values may be evicted from the register cache at any time due to conflicts or capacity limitations. Within this framework, the register file assumes the role of ensuring that no values are lost; therefore, *all* values must be written to the register file (the same is not true of the cache). Thus, we refer to the register file as a *backing file* when coupled with a register cache. The backing file must be able to support the full write bandwidth of executing instructions, although its latency is not critical. Because the register cache filters the vast majority of reads from the backing file, in those rare instances in which a value must be obtained from the backing file, a single read port (which can be shared with one of the write ports) suffices. By virtue of the significantly lower number of ports—as little as one-third of the original number—a backing file will be smaller and faster than a register file of the same capacity operating without a cache.

Some fraction of instructions will not be able to obtain an input operand from the register cache, resulting in a register cache miss. Referring to Figure 3, we see that instructions issuing more than two cycles (i.e.,

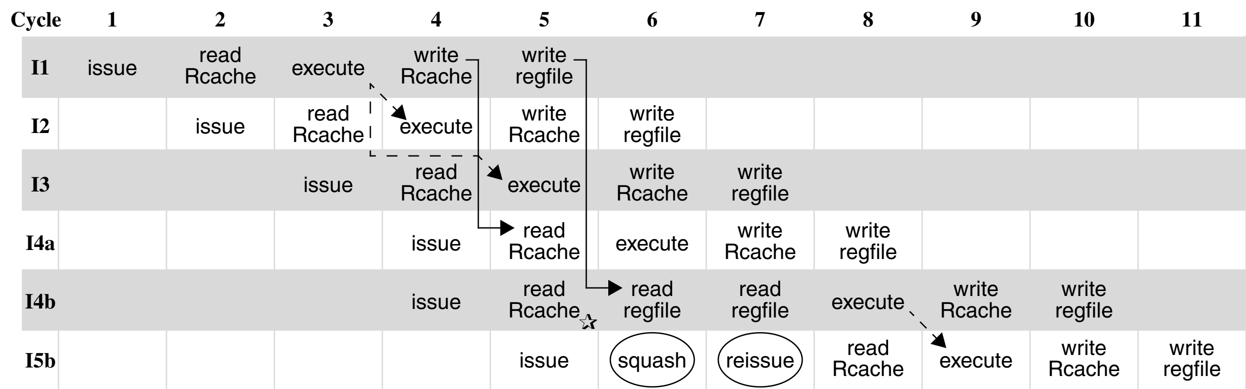

Figure 3. Operation of a single-cycle register cache with a two-cycle backing file. Dotted lines indicate bypassed values; solid lines, values communicated through storage (i.e., register file or cache). I2-I4 are data-dependent on I1 only; I5b is data-dependent on I4b. A two-stage bypass network supplies the inputs for I2 and I3. Instructions issuing with and after I4 can obtain the value through the register cache. I4a shows the normal hit case. The star indicates a register cache miss on I4b (i.e., the value from I1 is not present in the cache). By the time the cache miss is detected, I5b has issued. I4b goes to the file for the result while instructions issued in cycle 5 (I5b) are squashed. Instructions dependent on I4b become eligible for reissue as the register file read finishes, and they obtain their input value from the bypass network at the beginning of cycle 9.

the number of bypass stages) after a parent instruction can experience a register cache miss on the communicated value. Because the backing file stores all values, correct execution is guaranteed by retrieving the needed value from the backing file. Once obtained, the value is placed in the register cache (to avoid subsequent misses) in parallel with its use by the instruction that caused the miss (e.g., I4b's input is written into the register cache during cycle 8).

Register cache misses cause further complications because, by the time a miss is detected, subsequent dependent operations may have already issued assuming their parent would find its inputs in the register cache. When this assumption fails, these instructions must either stall until their parent completes or replay (i.e., reissue at a later time). This situation is analogous to the result of a data cache miss under load-hit speculation [14]. Stalling the dependent instructions is difficult because the issue pipelines must buffer them while allowing other, independent instructions to pass them. Replay-based solutions are also complicated although several different processors have already implemented them to support load-hit speculation (e.g., the Alpha 21264 [10]). The consequences of a register cache miss and the miss model are explored in more detail in Section 5.2.

The register caching framework just presented makes no assumptions about the organization of the register cache, its policies (e.g., replacement policy), or how it is accessed. Previously proposed caching schemes that fit within this framework [6, 15] suffer from two primary shortcomings. The first is their use of fully-associative register caches. Using such caches for an 8-wide machine requires 24 associative ports (two read and one write per issue port), resulting in access latencies that may not provide enough advantage over the direct-mapped, full-sized register file the caches were intended to replace. The larger issue, however, is their use of an LRU strategy to manage the cache contents, which, as we will show in Section 5.4, fails to capture the behavior of register accesses. In the next two sections, we present our policies for managing and indexing register caches more intelligently.

3. Use-Based Register Cache Management

The critical issue in the design of any caching scheme is what needs to be cached. Our key observation is that only values that have yet to be read by unexecuted consumers need to be cached. Given perfect a priori knowledge of the upcoming uses of values, only live values need be maintained in the cache, reducing the required cache size. In this section, we propose new insertion and replacement policies that attempt to keep only live values in the cache.

Central to these policies is the ability to determine the "liveness" of a given value. A live value is one that has outstanding uses; knowledge of upcoming uses enables us to assess the need to cache each value. Our scheme tracks the remaining uses for values while the values are present in the bypass network or register cache. Each entry within the register cache is augmented with the number of uses of the stored value remaining. As the value is supplied to consumers by the cache, the remaining-use count is reduced. When the count is zero, the value is no longer live and its cache entry may be reclaimed when needed. The integration of the augmented register cache into the issue-execute pipeline is illustrated in Figure 4.

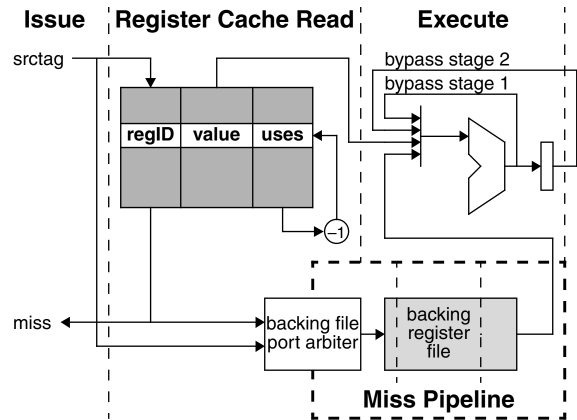

Figure 4. Register cache read pipeline. The register cache (dark gray) is accessed after an instruction issues. On a hit, the value is supplied to the ALU and the use count of the cached value is decremented. On a miss, the tag is sent to the backing register file, and the issue port is stalled; an arbiter is used to resolve simultaneous misses on multiple values by delaying some misses.

The rest of this section details use-based register cache management. The insertion and replacement policies are the topics of Section 3.1 and Section 3.2, respectively. The process of tracking the number of remaining uses for each value is detailed in Section 3.3. Section 3.4 covers the implications of incorrect use information.

3.1. Register cache insertion policy

The register cache insertion policy seeks to avoid caching values that will never be read from the cache. Since the bypass network supplies many input values (57% in our simulations), our policy simply avoids writing those values that have no uses left after bypassing. Referring back to the pipeline diagram of Figure 3(b), we see that prior to an executing instruction (e.g., I1) writing its result into the register cache, any consumers that issued one cycle later (e.g., I2) are reading from the register cache. Simultaneously, the bypass circuitry detects that their input value is on the bypass network rather than in the cache. If these consumers comprise all of the predicted consumers of the parent instruction, the results of the parent instruction need not be written into the register cache.

Only next-cycle consumers can affect the cache write decision of a parent instruction. By the time a consumer instruction issuing two cycles after its parent

(e.g., I3) obtains its input from the bypass network, the parent instruction will have already commenced writing into the register cache. Despite this limitation, we find that many values can avoid writing the register cache using this mechanism due to the high proportion of consumer instructions that issue immediately after their parents and the low average number of uses of each value.

This policy is very similar to a heuristic proposed by Cruz et al. [6] with an important advantage. Their heuristic, labeled *non-bypass*, wrote a value into the register cache if it was not bypassed to any instructions prior to the write occurring. Since most values have a single consumer, this scheme was intended to keep these values from polluting the register cache when that consumer was satisfied from the bypass network. However, values with many consumers that bypass to only some of their consumers prior to the write are also filtered from the cache, resulting in additional misses. In effect, this scheme uses bypassing as a rough proxy for the number of remaining uses; our scheme tracks the remaining uses explicitly.

3.2. Register cache replacement policy

While we would ideally like to be able to cache all values with remaining uses, the capacity and/or organization of the cache may preclude this. Therefore, a replacement policy is also necessary. As we will show in Section 5.4, LRU is a poor choice as a replacement policy for a register cache. However, the availability of future use knowledge (in the form of remaining-use counts) allows for use-based victim selection.

To minimize the number of register cache misses, we select the victim with the smallest number of remaining uses. In the event of a tie, we fall back on LRU or an approximation thereof. Most of the time (84%), the victim selected in this manner has zero remaining uses, and evicting the value does not result in a cache miss. For victims with one or more uses remaining, at least one miss will result from the eviction (provided the use information is accurate). Although a value is brought back into the register cache after a miss, the use count is lost and assumed to be zero (see the discussion in Section 3.3), making it the first choice for subsequent replacements. Thus, the greater the number of remaining uses a value has, the more misses it can cause.

3.3. Counting remaining uses

The use-based policies we have just described depend upon knowledge of a value's future uses, which we can obtain through prediction. Degree of use prediction [5] employs a history-based predictor that associates the number of consumers of an instruction's result with that instruction's address and some branch information. When subsequent instances of the instruction are observed, the predictor recalls the number of uses of the instruction's result that occurred previously, which can be used to initialize the remaining-use count. The degree of use predictor used for this work achieves an average accuracy of 97%.

Predictions are unavailable for some values because the predictor is finite and must be trained prior to being able to supply predictions. For these values, we assign an implicit prediction, which we call the *unknown default*. A similar situation arises after a register cache fill because the backing file does not contain use information. Maintaining use counts in the backing file adds significant complexity (the need to update the backing file on every use) for little benefit (since fills are infrequent). As in the case of an unknown initial degree of use, we set the remaining-use count to an algorithm parameter called the *fill default*. Any difference between the actual number of remaining uses and the default assumed in these cases manifests in the same way as an initial degree of use misprediction (see Section 3.4). The effect of these default values is discussed in Section 5.3.

Once the remaining-use count has been initialized, it must be updated both before and after a value is written into the register cache. The process of updating the remaining-use counts for cached registers is straightforward because the reads must access the cache. However, special care must be taken to maintain the use counts for values that are bypassed prior to being written into the register cache.

Finally, we note that it is desirable to pin high-use values in the register cache. The degree of use predictor uses the maximum representable number of uses to denote that and all higher numbers of uses. If a single value has millions of uses, subtracting from the saturated maximum is not the desired behavior due to the high cost of evicting that value. Therefore, for values with the maximum predictable degree of use, the remaining-use count is not updated. Such values remain pinned in the cache until the corresponding physical register is freed. The effect of varying the maximum degree of use is studied in Section 5.3.

3.4. Incorrect use information

Inaccurate remaining-use counts arise from degree of use mispredictions, the use of unknown and fill defaults, and the counting of wrong-path uses resulting from control-flow mis-speculation. These events result in disagreements between the number of remaining uses recorded in the register cache and the number actually outstanding. The differences manifest in two ways. First, a value might be present in the register cache with predicted remaining uses that will never be observed. We refer to these as *stale values*. The storage of stale values inflates the number of register cache entries required. Alternatively, the cache state could indicate that a value has no remaining uses even though that value is still live. These *falsely-dead values* can lead to register cache misses if the values are evicted.

The impact of stale values is limited by two factors. Most importantly, the invalidation of register cache entries when the corresponding physical registers are freed (necessary to ensure correctness) bounds the lifetime of stale values in the register cache to the dead time of the physical register (see Section 2). Also, most stale values have only a small number of specious remaining uses once their actual uses have been counted, making them susceptible to the normal replacement process.

The potential cost of falsely-dead values is also mitigated in practice for two reasons. First, values remain in the cache—even if their remaining-use count reaches zero—until they are explicitly chosen as a victim by the replacement policy. Thus, unless there is actual contention among live values for entries in the same set as the falsely-dead value, the cache will continue to supply the value. Second, on a wide-issue machine, many consumer instructions obtain their inputs from the bypass network. Therefore, especially for values with few uses, all of those uses may be satisfied without incurring a register cache miss, even if the predicted number of uses was too low.

4. Decoupled Indexing

As we will see in Section 5, conflict misses are a significant portion of all register cache misses, resulting in performance that is highly dependent on cache associativity. Previous schemes have addressed this issue by using fully-associative structures [3, 6, 15], which are expensive in both area and latency. Also, the implementation of our replacement policy requires that the entry with the fewest remaining uses be identified, a costly task when the associativity is high.

These factors motivate our proposal of *decoupled indexing*, a technique that assigns register cache set indices intelligently to improve the performance of set-associative register caches. In this section, we describe why standard indexing methods are inappropriate for register caches. Then, we present a general mechanism that enables an arbitrary register cache set to be assigned to a value at the same time a physical register is assigned (i.e., well before the value exists). Finally, we discuss specific algorithms for making these assignments with the goal of reducing conflict misses. As in the register cache policies of Section 3, use information will play an important role.

4.1. Cache indexing

All caching schemes demand that values have a unique identifier so that the presence of the desired value in the cache may be ascertained. A standard indexing scheme uses a portion of this identifier to generate the set index. For example, the low order bits of an address (the identifier) serve as the index for a data cache; the remaining identifier bits comprise the tag that is stored in the cache itself.

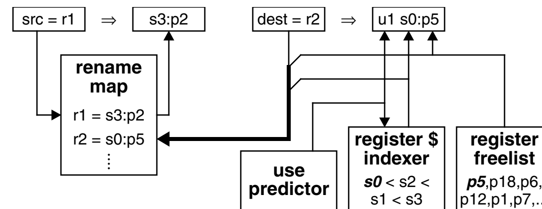

Figure 5. Decoupled indexing. *p* denotes the physical register, *s* the corresponding register cache set, and *u* the predicted number of uses remaining. Each architectural register is renamed to both a physical register and an independent register cache set. Subsequent consumers obtain the set index along with the physical register tag.

For a memory cache, deriving the cache set index from the value identifier (i.e., the address) works well because of spatial locality: consecutive memory locations are likely to be used together, so consecutive addresses should be mapped to sets that will not conflict. However, there is no spatial locality in physical register identifiers—they are simply assigned from a freelist. Thus, it makes no sense that the index into a register cache should be related in any way to the physical register identifier. Fortunately, nothing precludes the use of an index completely independent of the identifier. Using an independent index allows arbitrary assignment of identifiers to sets within the cache in order to avoid conflicts. In addition, the technique also trivially enables the use of non-power-of-two-sized caches. We refer to this scheme as decoupled indexing.

Due to the nature of physical register handling, it is straightforward to apply decoupled indexing to a register cache. When a physical register is allocated, the physical register tag obtained from the freelist is augmented with a register cache index from another source. The index is tracked and provided to consumers of the value using the standard rename process: the map table is widened to include the register cache index for the current mapping of each architectural register in addition to the associated physical register number. From the perspective of the rename map, the combination may be treated as a wider physical register identifier. Like the rest of the map table contents, these indices must be recovered on a mis-speculation. This mechanism does not introduce a level of indirection: the index is immediately available to all potential consumers, and it does not change until the associated physical register is freed and reallocated. Finally, because the cache index no longer has any relationship to the physical register identifier, the full identifier for each value must be stored in the register cache as a tag. This process is illustrated in Figure 5.

4.2. Set-assignment algorithms

The generality of the decoupled indexing mechanism allows for the implementation of many index assignment policies. Note that the register cache set for a value must be assigned far in advance of the writ-

Table 1: Simulator configuration

Pipeline	4-stage fetch (next address+I-cache), 2-stage decode, 3-stage rename, 2-stage dispatch (write into instruction window), 1-stage issue, 1-stage register cache read/write, 1- to 18-stage execution, 3-cycle RF latency, 2-cycle backing RF latency (see text). 15 cycle min. branch mis-speculation loop
Front-end	8-wide fetch (nops skipped) with perfect BTB and up to one taken branch per fetch block, 12KB YAGS conditional branch predictor, 64-entry return address stack, 32 KB cascading indirect branch predictor
Issue	128 entries with 512 entry reorder buffer, 8-wide issue, oldest ready first, 512 physical registers, 8-wide retirement (2 stores per cycle maximum), 128-entry load queue, 128-entry store queue
Execution	6 integer ALUs, 1-cycle latency, 2 branch resolution units, 2-cycle latency, 2 integer multipliers, 4-cycle latency, 4 floating point ALUs, 3-cycle latency, 2 floating point multiplier/dividers, 4/18 cycle latency, 2 load units, 4-cycle load to use latency on L1 hit, 2 store units, 3-cycle execute to earliest retirement; all units fully-pipelined. Two-stage bypass network (ALU feedback + register cache write to read).
Memory	32KB, 2-way L1-I and D caches, 64-byte lines, perfect TLBs; 1 MB, 4-way unified L2 cache, 128-byte lines, 12-cycle latency, critical word first; 64-entry unified prefetch/victim buffer on each of L1 and L2; 16-entry coalescing store buffer; 180-cycle memory latency; opportunistic unit-stride prefetcher
Use predictor	9KB: 4K-entry, 4-way set-assoc., 2-bit confidence, 6-bit future control flow, 6-bit tag, 4-bit prediction

ing of that value into the cache. Thus, in order to be successful, the set assignment algorithm must be able to *anticipate* the state of the register cache. Fortunately, the predicted use information available for renamed instructions provides useful hints to make accurate guesses. Values with larger numbers of uses are more likely to be placed in the register cache and remain there longer. Another source of information is the sequential rename order, which, due to data dependencies is related to the order of their execution. We describe three policies for set assignment: **minimum**, **round-robin**, and **filtered round-robin**.

The **minimum** policy tracks, for each set, the sum of the predicted number of uses of all values assigned to that set. An instruction generating a result is assigned the set with the minimum sum, increasing that sum when the instruction is renamed and decreasing it when it retires. This algorithm reduces the likelihood of long-lived (high-use) values being evicted from the cache by subsequent assignments to the same set.

While conceptually simple and appealing, the actual implementation of the minimum policy would be quite difficult. The **round-robin** policy forgoes all use information for the sake of simplicity and merely assigns sets sequentially to instructions as they are renamed. This scheme relies upon the correlation between instruction rename and execution sequences to avoid assigning the same set to the results of instructions that will execute together.

Our final policy, **filtered round-robin**, attempts to add some notion of use information to the round-robin scheme. A count of high-use values assigned to each set is maintained, and sets for which the count exceeds a certain threshold are skipped in the normal round-robin order, reducing conflicts in sets with high-use values. Counts need only be updated on the rename or retirement of high-use values, a relatively infrequent occurrence. We have found empirically that defining high-use values to be those with greater than five predicted uses and using a threshold equal to half the associativity (number of entries per set) works well.

5. Evaluation

We now perform a detailed evaluation of use-based register caching. We begin with a description of our simulation infrastructure and benchmarks in Section 5.1. In Section 5.2, we discuss some of the important details surrounding the accurate modeling of register caching. Section 5.3 investigates the impact of the various cache parameters on performance to select a design point for further study. Section 5.4 performs this detailed study, presenting many non-performance metrics (e.g., hit rate and occupancy) and comparing them with results for previously proposed caches. We conclude the section with a performance evaluation in Section 5.5.

5.1. Simulation methodology

We generated our results using a detailed execution-driven simulator of the Alpha ISA. Only user-level code is simulated. The system call and functional execution portions of the simulator come from the SimpleScalar v3.0 tool suite [4], while the timing simulator has been written to accurately model real hardware limitations. For our benchmarks, we use the SPEC 2000 integer suite. The benchmarks were compiled using the Compaq Alpha compiler at optimization level –O3. Simulation results were generated by executing the first two billion instructions (excluding nops) of each benchmark on the training inputs.

We have chosen a processor configuration designed to reflect an aggressive out-of-order processor for which the read latency of a monolithic register file would be a performance issue. The processor has a deep pipeline, and the front end, execution resources, and cache hierarchy are aggressive. The processor attributes are summarized in Table 1.

We distinguish between a register file (no register cache) and a backing file (used to back up a register cache). Register file latency does not affect our tuning or characterization, but sets the baseline performance

against which register caching is evaluated. Based on the pipeline depth and other latencies assumed in our simulation model, we assume a register file with two bypass stages and a latency of three cycles (each for read and write). Performance with other register file latencies are superimposed on many of the figures in this section. Except where varied in Section 5.5, we have used two cycles as the latency for the backing register file. As discussed in Section 2.2, the two-thirds reduction in the number of ports when used behind a cache allows a backing file to be faster than the equivalent-capacity register file.

5.2. Register cache simulation

There are two important issues to consider in the accurate evaluation of a register cache. The first of these is the overall structure of the pipeline and the second is the handling of register cache misses. These details contribute to a larger miss penalty in our evaluation than has been previously suggested for register caching; we believe this may account for the significant difference in the performance advantage of register caching presented here versus in prior work.

As discussed in Section 1, the implementations for which a register cache are likely to be beneficial are wide machines with deep pipelines. Therefore, it is important to evaluate register caching in a deeply-pipelined, wide-issue machine; the simulator configuration outlined in Table 1 reflects this consideration.

Regarding register cache misses, we assume that a miss necessarily results in the replay of all instructions issuing in the cycle after the missing instruction issues (equivalent to the model implemented by the Alpha 21264 [10]). Instructions independent of the missing instruction may then reissue in the following cycle, while the dependent instructions are delayed. The delay experienced by an instruction that misses in the register cache also depends on both the register file read and write latencies. Besides additional delay caused by contention for a backing file read port, the instruction may have to wait to ensure that the desired result has finished writing into the register file. Our simulator accounts for both of these effects.

5.3. Tuning

In this section, we explore the parameter space of the register cache to find a good design point for further study. For each parameter except the cache size and associativity, we choose the value that maximizes the mean performance over our benchmarks. Increasing cache size and/or associativity always increases performance, so we choose the best-performing organization that is likely to have a single-cycle latency given the full register file latency and its size relative to the register cache.

We begin with the selection of cache organization. For this experiment, we presciently select the final values used for the maximum use count (7), the unknown default (1), and the fill default (0). Figure 6 presents the results. The most striking feature of the data is the impact of the associativity on register cache performance. Two-way set-associativity is probably the minimum for reasonable performance. Direct-mapped register caches, even beyond 80 entries, fail to break even with the performance of the baseline machine with a three-cycle register file. Associativity continues to make a noticeable impact all the way up to a fully-associative design, especially for caches with fewer than 64 entries. The flattening of the fully-associative curve at around 56 entries is hardly surprising given that this was also the 90th percentile number of live values that we determined earlier (see Figure 2). Based on these data, we propose the use of a 64-entry, two-way set-associative design.

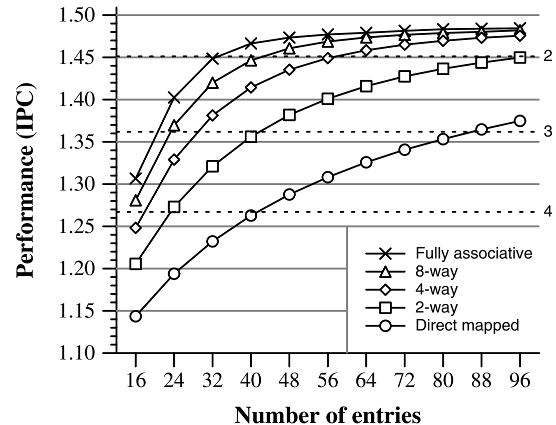

Figure 6. Register cache size and organization. Each datapoint represents the average performance of that configuration on all benchmarks. All caches use standard indexing. Dotted lines show the performance of a machine without a register cache for varying register file latencies.

The strong dependence of performance on associativity indicates that conflict misses are problematic. Figure 7 explores the various decoupled indexing algorithms proposed in Section 4.2. As expected, the two use-based set assignment schemes (filtered round-robin and minimum) perform the best. Filtered round-robin increases performance by 1.9% on a two-way set-associative register cache. Minimum performs nearly as well, but is much more complex. Even round-robin set assignment improves performance measurably over the baseline scheme. As expected, advantages are more pronounced for less associative register caches.

The performance behavior versus the maximum number of uses tracked is straightforward: a lower maximum leads to lower performance. Note that it is not the cache's ability to distinguish between a value with (say) six and eight predicted uses that makes a higher limit perform better; rather the higher limit reduces the number of values that are pinned in the cache (recall that use counts are not updated for entries with predicted counts equal to the limit) until their

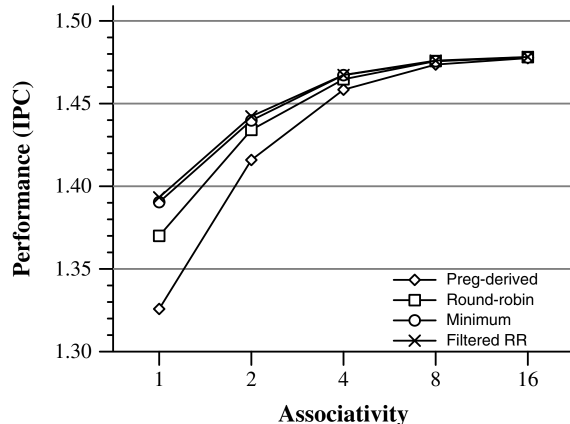

Figure 7. Decoupled indexing algorithms. preg = low-order bits of physical register tag; round-robin = sequential assignment of set indices; minimum = set with fewest total predicted uses assigned; filtered = round-robin with sets containing high-use values skipped.

physical registers are freed, reducing capacity pressure. Experimental data (not shown) indicate higher limits are beneficial up to about 12; performance falls off rapidly for limits below six. Given that three bits can represent up to seven uses and this number is approximately at the knee of the performance curve, we employ a maximum use limit of seven uses for the remainder of our evaluation.

We also gathered data on the effect of the unknown and fill defaults (not shown). If either default is too low, it results in premature evictions; too high, and the resulting stale values increase register cache occupancy, leading to evictions of other useful values. An unknown default of one use gives the best performance for most configurations; this is intuitive given that the majority of values are used once. We find that a fill default of zero maximizes performance over the range of cache sizes studied. While this may seem non-intuitive, it actually is reasonable since any given use of a value (i.e., the one that caused the fill) is most likely to be its last. Also, since values with zero remaining uses can remain in the cache for long periods, the value may still satisfy additional consumers beyond the one that caused the miss.

5.4. Characterization

Having selected a design point for our register cache, we are now in a position to evaluate it in more detail and compare it with other hardware-managed register hierarchies. Our reference designs are also register caches, differing from our scheme only in their policies for insertion, replacement, and indexing. **LRU** refers to an implementation of the basic idea put forth by Yung and Wilhelm [15] in which all values are written into the register cache and the LRU entry is selected for replacement. The **non-bypass** design [6] avoids writing values into the register cache if they bypass to any consumers prior to the cache write (as for our cache, only first-stage bypasses affect the write decision). Again, the LRU entry is selected for replacement. Unlike the original design, we allow an instruction that experiences a register cache miss to receive the value directly from the register file; thus, the fill takes place in parallel with the instruction's first execution stage. In this section, we focus on non-performance metrics specific to caching schemes—relative performance figures are presented in the next section.

Figure 8 depicts the register cache miss rate, which has a first-order impact on the overall performance. For both standard and filtered round-robin indexing, the figure breaks down the misses into those caused by (1) avoiding the initial write of the value, (2) evicting the value due to capacity limitations, or (3) conflicts. The LRU cache writes all values into the register cache as they are generated; the other designs attempt to reduce capacity and conflict misses by avoiding the writes of certain values. The data show that write filtering is effective at reducing these misses at the expense of causing misses on filtered values. In the case of the non-bypass scheme, these new types of misses cause the aggregate miss rate to exceed that of the simple LRU scheme for this cache size. Use-based filtering, however, results in a substantially lower total miss rate. In all schemes, cached values are only evicted due to capacity limits or conflicts. Therefore, the use-based replacement policy, which evicts fewer useful values, further reduces misses in these categories beyond that obtained from filtering only.

Figure 8 also illustrates the 30-40% reduction in conflict misses attainable from decoupled indexing. Because this benefit is independent of the policies that differentiate the three caches, we use decoupled indexing on all three caches in the remainder of our evaluation. The two reference designs will use round-robin decoupled indexing, which does not require use information, while our design will use filtered-round-robin indexing.

The average read and write bandwidths to both the register cache and register file are depicted in Figure 9. The non-bypass and use-based schemes demonstrate reduced cache write bandwidth versus the LRU-managed cache because of write-filtering. The register file read bandwidth is approximately proportional to the miss rate since the register file is only read on a cache fill (i.e., a miss). Both the cache read and register file write bandwidths essentially track the performance. This behavior is expected since (to first order) the same number of values are read and written by all configurations. Because the register file sees all writes and the register cache sees all reads, shorter execution time increases these bandwidths.

The effects of write filtering are also evident in Figure 10. The non-bypass and use-based schemes attempt to filter only those values that are thought to be dead at the time of the cache write, significantly reducing the number of values written to the cache that are

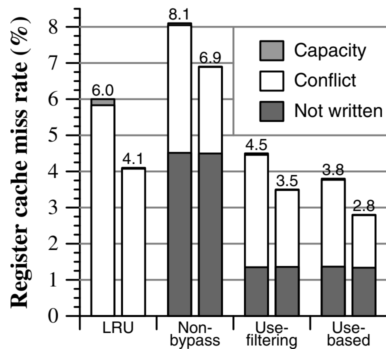

Figure 8. Register cache misses. Contribution of different types of misses to the overall miss rate under standard indexing (left bar) and filtered round-robin indexing (right bar). Miss rates are per operand, not instruction.

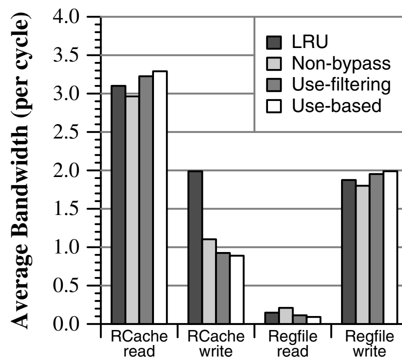

Figure 9. Average access bandwidth. Accesses per cycle by type and structure. Cache write bandwidth includes initial writes and fills. In each case, the fill bandwidth portion equals the register file read bandwidth.

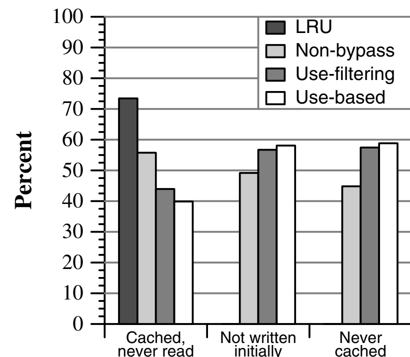

Figure 10. Filtering effects. The bar groups from left to right depict the percentage of all cached values not read before invalidation or replacement, of initial writes filtered from the cache, and of retired values never cached.

not subsequently read (cached dead values) versus LRU. A live value may *also* be classified as cached and never read if it is evicted prior to being used: this will necessarily result in a later conflict or capacity miss. Therefore, use-based replacement, which reduces these types of misses beyond use-based write filtering alone, shows the lowest fraction of cached and never read values.

The difference in values not written initially and those never cached is also enlightening. The use-based caching scheme filters a higher proportion of initial writes than does the non-bypass scheme. Along with the bandwidth data in Figure 9, this demonstrates that the lower miss rate on filtered values (Figure 8) is not a result of less aggressive filtering. The overall better filtering decisions result in a larger fraction of values that never occupy the cache at all.

Table 2 presents additional data comparing the caching algorithms. More effective schemes keep values that have more uses in the cache, resulting in a greater number of reads per cached value. Ideally, the number of times each value is cached (the cache count) should be as low as possible for a given miss rate. The LRU scheme is guaranteed to have an average cache count of at least one (every value is cached)—the extent to which this metric exceeds one depends on the number of fills for evicted, useful values. Write filtering reduces the average cache count below one for the other two schemes. The cache occupancy and entry lifetimes are also affected: the fewer values that are written to the cache, the lower the average occupancy and the longer the lifetime of each entry. Note the low average occupancy relative to cache size (64 entries).

Table 2. Comparison of register cache metrics

Average	LRU	Non-bypass	Use-based
Reads per cached value	0.67	1.18	1.67
Times each value is cached	1.09	0.61	0.44
Cache occupancy (entries)	36.66	28.84	26.60
Cache entry lifetime (cycles)	25.18	36.34	43.58

This data explains both the negligible contribution of capacity misses (Figure 8) and the small performance change with cache size for fully-associative caches larger than 32 entries (Figure 6). Thus, the use of a 64 entry cache over a smaller design is primarily beneficial because it provides more sets (reducing conflicts).

5.5. Discussion

Now we wish to ascertain when it is beneficial to use a register cache and the relative advantage of use-based register caches over other register hierarchies. In this section, we introduce an additional reference design, labeled **two-level**. This design is not a register cache, but is an optimistic version of the two-level register file proposed by Balasubramonian et al. [3] (see Section 2). We made the following changes: (1 the L1-L2 bandwidth is increased from one to four values per cycle, (2) L2-L1 register transfers after exceptions are explicitly modeled,† (3) the L2 register file is assumed to be infinite, and (4) the floating-point and integer register files are unified. All of these changes except the last improve the performance over the original design, the first two significantly. The final change was necessary to incorporate this scheme into our simulator, and represents an L1 capacity penalty equal to 31 architected floating-point registers for our mostly-integer benchmarks. Therefore, we compare each register cache of a given number of entries against the two-level register file with an L1 containing that number plus an additional 32 entries. Because the L1 register file is direct-mapped rather than set-associative, it can contain a larger number of entries and potentially remain competitive with a cache-based scheme.

We begin with a performance comparison of the different algorithms versus register cache/L1 size in

† L2-L1 transfers occur at four registers per cycle in parallel with the pipeline refill; the pipeline is stalled if the register transfers are incomplete by the time the new instructions reach the rename stage.

Figure 11. Due to its lower miss rate, use-based register caching outperforms the other caching schemes across a large range of capacities. The miss rate results from the prior section account for the performance advantage observed versus the two other caching designs. The lower performance of the two-level scheme results primarily from rename stalls, although delays in copying registers after mis-speculations also contribute, in spite of the generous L1-L2 bandwidth provided. Due to the nature of the two-level scheme, its L1 register file must contain at least one more register than the number of architected registers; in practice, an even larger number is required to prevent stalls for lack of rename registers, which cause performance to fall off rapidly as the L1 size decreases. The two-level scheme requires the difficult determination of when values should be moved to the L2—too soon, and the recovery cost dominates; too late, and the pipeline stalls frequently for lack of registers. Both the recovery time and the ability to maintain free L1 registers are highly dependent on the L1-L2 bandwidth: with a more realistic bandwidth of two register moves per cycle, the performance of the two-level register file drops over 2%, putting it below that of even an LRU cache over the entire cache/L1 size range.

The performance advantage of use-based caching over the other caches increases as the caches get smaller—for small caches, it is crucial to minimize unnecessary writes to the cache and to select the proper victims. The filtering performed by the non-bypass and use-based caching algorithms is nearly independent of cache organization and results in a miss rate approximately constant versus capacity. Therefore, the advantage of these schemes over LRU diminishes as the cache gets larger. For large enough register caches, misses due to filtering will always dominate capacity and conflict misses, and the LRU scheme will perform the best. However, caches of this size are unlikely to have a single-cycle latency.

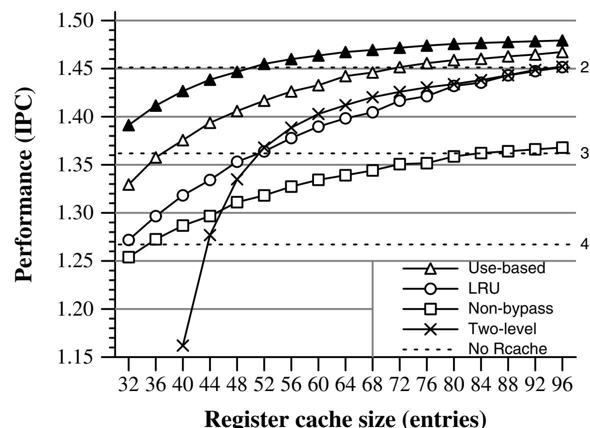

Figure 11. Performance versus cache/L1 size. All caching schemes use a two-cycle backing file and round-robin (LRU, non-bypass) or filtered round-robin indexing (use-based). Caches are two-way set-associative, except that a four-way use-based cache is also shown (solid triangles). The L1 file of the two-level scheme uses the indicated number of entries plus 32. Dotted lines show the performance without a register cache for varying register file latencies and two bypass stages.

Figure 11 also shows the performance of a four-way set-associative use-based register cache for comparison. This organization achieves equivalent performance to the 64-entry two-way baseline with only 48 entries, and it exceeds the performance of the three-cycle monolithic register file with fewer than 32 entries. Data we have not shown indicate that use-based caching also benefits more from increased associativity than the other caching schemes due to the intelligent replacement policy. The advantage of four-way associativity comes at the expense of complex victim selection and potentially greater cycle time.

At this point, we take a brief detour to discuss the relative performance of the non-bypass and LRU schemes. We were surprised by the lower performance of the non-bypass scheme, especially since the heuristic is based on sound principles (the low average degree of use and the likelihood of bypassing to all of the consumers). Many of the non-performance metrics indicate that the non-bypass heuristic has the desired effects versus LRU—lower write bandwidth, occupancy, and cache count, and greater entry lifetime and reads per value. However, these benefits are overcome by an increase in overall misses since the number of new misses due to write filtering exceeds the reduction in capacity and conflict misses. As the cache size decreases, non-bypass performs relatively better than LRU because capacity and conflict misses increase while the number of misses due to write filtering remains approximately constant. Other data (not shown) indicate that the performance of LRU and non-bypass break even around 20 entries.

Relative performance versus the latency of the backing file (L2 file for the two-level scheme) appears in Figure 12. Use-based caching exhibits much lower performance degradation with increasing backing file latency than the other two caching schemes. The two-level register file is less sensitive still to its L2 register latency, which is only observed on some mis-speculations, but the performance is lower than that of use-based caching through backing/L2 file latencies of 4 cycles, in spite of the extra storage and L1-L2 copy bandwidth. Until the backing file latency makes the register cache miss penalty exceedingly large, the cost of misses, which delay only a small number of instructions, is less than the cost of the pipeline stalls introduced by the two-level scheme, which delay all of the instructions in the front end.

A use-based register cache provides a performance advantage over a three-cycle monolithic register file even for backing file latencies up to five cycles. With a two cycle backing file, a use-based cache performs 6% better than a three cycle register file, recovering over half of the performance lost from a single-cycle register file. The advantage over a multi-cycle register file increases as the latency of the register file increases.

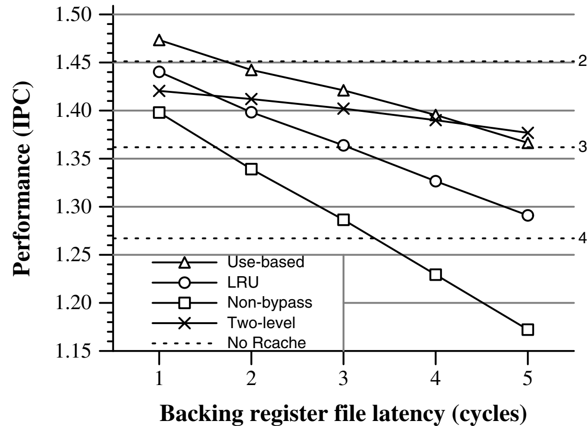

Figure 12. Performance versus backing file/L2 latency. Hierarchical register files with different backing/L2 file latencies. All caching schemes use round-robin indexing (LRU, non-bypass) or filtered round-robin (use-based). The caches have 64 entries while the L1 file of the two-level scheme has 96. As in Figure 11, dotted lines depict non-cache performance.

Provided the backing file can be made just one cycle faster than the original register file (by virtue of having one-third lower number of ports), use-based register caching outperforms all multi-cycle register files.

6. Conclusions

Register caches are small structures that store a subset of the contents of the full register file, enabling them to supply values to the execution core at lower latency. Register caching schemes proposed to date have used ineffective heuristics, resulting in the need for fully-associative caches to avoid high miss rates.

This work introduced two new techniques to improve the performance of register caches. Whether a value has outstanding uses is the single most relevant piece of information in deciding whether it should be in the cache. *Use-based cache management* consists of insertion and replacement policies that track the expected number of uses remaining to manage the cache contents. *Decoupled indexing* eliminates the connection between indexing a register cache and the physical register tags of the contents, allowing for more intelligent assignment of values to cache sets to reduce conflict misses in set-associative caches.

Acknowledgements

The authors would like to thank Paramjit Oberoi and Brandon Schwartz for numerous helpful discussions. Saisanthosh Balakrishnan, Allison Holloway, Amir Roth, and Philip Wells also assisted in preparing this manuscript.

This work was supported in part by National Science Foundation grants EIA-0071924 and CCR-0311572 and the University of Wisconsin Graduate School. Adam Butts was supported by fellowships from the Fannie and John Hertz Foundation and Intel Corporation.

References

[1] P. Ahuja, D. Clark, and A. Rogers. The performance impact of incomplete bypassing in processor pipelines. In *Proc. of the 28th Intl. Symp. on Microarchitecture*, December 1995. pp. 36-45.

[2] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proc. of the 34th Intl. Symp. on Microarchitecture*, December, 2001. pp. 237-48.

[3] E. Borch, E. Tune, S. Manne, J. Emer. Loose loops sink chips. In *Proc. of the 8th Intl. Symp. on High-Performance Comp. Arch.*, February 2002. pp. 270-81.

[4] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, June 1997.

[5] J. A. Butts and G. Sohi. Characterizing and predicting value degree of use. In *Proc. of the 35th Intl. Symp. on Microarchitecture*, November 2002. pp. 15-26.

[6] J.-L. Cruz, A. González, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *Proc. of the 27th Intl. Symp. on Comp. Arch.*, June 2000. pp. 316-25.

[7] A. González, J. González, and M. Valero. Virtual-physical registers. In *Proc. of the 4th Intl. Symp. on High-Performance Comp. Arch.*, February 1998. pp. 175-84.

[8] A. Hartstein and T. Puzak. The optimum pipeline depth for a microprocessor. In *Proc. of the 29th Intl. Symp. on Comp. Arch.*, May 2002. pp. 7-13.

[9] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proc. of the 29th Intl. Symp. on Comp. Arch.*, May 2002. pp. 14-24.

[10] R. Kessler. The Alpha 21264 microprocessor. In *IEEE Micro*, vol. 19, March-April 1999. pp. 24-36.

[11] L. Lozano C. and G. Gao. Exploiting short-lived variables in superscalar processors. In *Proc. of the 28th Intl. Symp. on Microarchitecture*, Dec. 1995. pp. 292-302.

[12] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proc. of the 29th Intl. Symp. on Comp. Arch.*, May 2002. pp. 25-34.

[13] J. Swensen and Y. Patt. Hierarchical registers for scientific computers. In *Proc. of the 1988 Intl. Conf. on Supercomputing*, July 1988. pp. 346-53.

[14] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proc. of the 26th Intl. Symp. on Comp. Arch.*, May 1999. pp. 42-53.

[15] R. Yung and N. Wilhelm. Caching processor general registers. In *Proc. of the Intl. Conf. on Comp. Design*, October 1995. pp. 307-12.

[16] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Two-level hierarchical register file organization for VLIW processors. In *Proc. of the 33rd Intl. Symp. on Microarchitecture*, December 2000. pp. 137-46.