Exploiting Value Locality in Physical Register Files

Saisanthosh Balakrishnan Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin-Madison
{sai, sohi}@cs.wisc.edu

Abstract

The physical register file is an important component of a dynamically-scheduled processor. Increasing the amount of parallelism places increasing demands on the physical register file, calling for alternative file organization and management strategies. This paper considers the use of value locality to optimize the operation of physical register files.

We present empirical data showing that: (i) the value produced by an instruction is often the same as a value produced by another recently executed instruction, resulting in multiple physical registers containing the same value, and (ii) the values 0 and 1 account for a considerable fraction of the values written to and read from physical registers. The paper then presents three schemes to exploit the above observations.

The first scheme extends a previously-proposed scheme to use only a single physical register for each unique value. The second scheme is a special case for the values 0 and 1. By restricting optimization to these values, the second scheme eliminates many of the drawbacks of the first scheme. The third scheme further improves on the second, resulting in an optimization that reduces physical register requirements with simple micro-architectural extensions. A performance evaluation of the three schemes is also presented.

1 Introduction

The physical register file is an important component of the micro-architecture of a modern dynamically-scheduled processor. As processors exploit more instruction level parallelism, even more demand is placed on the physical register file: larger instruction windows require storing the result of more instructions in the register file, and wider instruction issue requires higher bandwidth access to the stored values. A straightforward approach to these demands is to make the physical register file larger to hold the results of more instructions (typically a physical register for each outstanding instruction that creates a value) and add more read and write ports to provide additional bandwidth. This approach results in large, monolithic storage arrays, and has several disadvantages. The register file size grows linearly with the number of registers, and greater than linearly with increase in read and write ports [7]. Wire lengths increase with the storage array size, increasing access latency. The increased access latency complicates bypass paths, adding multiple levels of bypass to functional units. It also increases branch mis-prediction penalty, capacity pressure on the register file and register lifetime. In short, increased latency generally results in degraded performance, making the brute-force scaling of a physical register file an undesirable design choice.

The lack of scalability of monolithic storage arrays has led processor architects to explore alternative designs for implementing a physical register file. The basis for these alternative designs is the exploitation of *locality*: patterns in references made to the register storage. Two forms of locality have received a lot of attention: (i) locality of access, and (ii) locality of communication. Localities of access have been used to design hierarchical register files [1, 18] and register caches [14, 21]. Localities of communication have been used to design clustered [2, 6, 10] and banked register files [5, 15, 19].

To date most proposals for alternative register files have been concerned with implementations that can provide a requisite amount of storage with desired latency and bandwidth characteristics. An orthogonal approach for physical register design is to make more efficient use of the storage by changing the storage management strategy. Pioneering work by González *et al.* proposed the concept of virtual-physical (VP) registers [8, 12]. VP registers make more efficient use of the physical register storage by delaying the allocation of a physical register until the time the value is ready, as opposed to the time the architectural register is renamed in the conventional management strategy. Following up on the concept of VP registers, Jourdan *et al.,* proposed an alternative scheme for the efficient use of physical registers [9]. This scheme exploits the fact that the same value might be present in multiple physical registers at the same time (*i.e.*, a form of *value locality*), and proposes to eliminate this duplication by using only a single physical register for a given value.

This paper further explores the use of value locality and alternative storage management strategies to optimize the

design of a physical register file. We start out by characterizing the form of value locality that we are interested in and empirically evaluating its magnitude. Our first observation (not new) is that a given value is frequently present in more than one physical register simultaneously. We exploit this observation with *Physical Register Reuse*, a scheme that is similar in spirit to the scheme of Jourdan *et al.* [9] but different in its details. This scheme, though conceptually elegant and powerful, has practical limitations. Our second observation (new, we believe) is a special case of the first one: of the values present in more than one physical register, 0's and 1's are the most common. We exploit this observation to come up with two new schemes for physical register management: *Registerless Storage* and *Extended Registers and Tags*. They are special-case implementations of Physical Register Reuse, and trade off the theoretical power of the concept with the practicality of implementation.

The remainder of the paper is as follows: in section 2 we discuss the background and the general ideas behind our proposed schemes. We describe our evaluation methodology in section 3, and present value locality measurements in section 4. In section 5, we discuss the different schemes for alternate physical register file designs. We present evaluations of these schemes in section 6. In section 7, we describe related work, and we present our conclusions in section 8.

2 Background and General Idea

To set the stage for exploring alternatives for designing a physical register file, we start out with a discussion of value locality. We then discuss the purpose of register files and the register rename map, making key observations that could be exploited to devise alternate physical register file designs. We then provide a brief introduction to our proposed schemes for optimizing the use of physical registers.

2.1 The Phenomenon: Value Locality

Value locality has been the subject of extensive study in recent years, and a variety of different types of value locality have been identified. The most widely studied form concerns the locality of values generated by different dynamic instances of a static instruction. The work on value prediction [11] exploits this locality to predict the outcome of a dynamic instance of a previously observed static instruction. The work on instruction reuse [17] exploits the same form of locality to reuse the result of a previous dynamic instance of a static instruction.

The type of value locality that this paper investigates and exploits is different. We are interested in the locality of the results produced by the instructions in a dynamic window, *i.e.*, the results of one or more dynamic instances of instructions. This kind of value locality has very recently been exploited for value prediction [22]. Value-centric data cache designs [20] also propose exploiting a similar form of locality present in memory references.

We propose to exploit this form of value locality to reduce the storage requirements for the results of the instructions in a dynamic instruction window (*i.e.*, physical register requirements). We are also concerned with the prevalence of special values, namely 0 and 1, for which further optimizations are possible.

2.2 The Enabler: Register Renaming

The purpose of a register file is to pass values from an instruction that creates a value (a producer instruction) to instructions that use the value (consumer instructions). The logical register name space is used to link a producer instruction with its consumer instructions. Named storage is used to hold the value created by the producer instruction, and consumer instructions are directed to the storage associated with the logical register to obtain the necessary value.

With physical registers, the process of passing values from producer to consumer instructions is slightly different. The logical register name space is still used to link producer and consumer instructions, but the actual storage for values is provided by a (typically larger) set of physical registers. A register rename map provides a level of indirection, redirecting accesses to a given logical register to a particular physical register. In other words, the register rename map directs accesses for a given dynamic value to the appropriate storage that contains the value.

The flexibility provided by the rename map is the key to optimizing the use of storage that holds the dynamically-created values. Different mapping strategies can be used to provide different ways of storing values and different ways of allowing consumer instructions to get the appropriate value. An early proposal for optimizing the use of physical registers was the concept of virtual-physical registers [8, 12]. Here a virtual name (a virtual-physical register) is assigned to a value during the rename stage, but an actual physical register (*i.e.*, storage) is not allocated to hold the value until it has been produced. Mapping tables are used to redirect the virtual-physical register to the actual physical register. The result is more efficient use of physical registers since physical register storage is not allocated to a value until the value actually exists (which is typically many cycles after the corresponding instruction has been renamed).

Both a conventional physical register strategy as well as a virtual-physical register strategy maintain a one-to-one correspondence between instructions creating values and storage elements used to hold the values. A conventional physical register strategy allocates a storage element for every value-producing instruction in flight, whereas a virtual-physical strategy allocates a storage element for every value-producing instruction that has actually produced its value.

Front-end	64 KB Instruction cache, 64 KB gshare branch predictor, 64 entry return address stack. The front-end is optimistic, and can fetch the full-width of the machine every cycle. All nops are removed without consuming any machine resources.
Execution Core	4-wide machine with the number of instruction window entries decided according to physical register file size. The pipeline depth is 14 stages. Available functional units include 4 simple integer units, 1 complex integer unit, and 2 load/store ports, all fully pipelined. Separate register rename maps and physical register file. Queue-based register free list management.
Memory Hierarchy	First-level instruction and data caches are 2-way set-associative 64KB caches with 64 byte lines and a 3-cycle access latency, including address generation. The L2 cache is a 4-way set-associative 2MB unified cache with 128-byte lines and 6-cycle accesses.

Table 1: Simulated processor parameters

One can conceive of alternate mapping strategies. One such strategy could map the values produced by multiple instructions to a single storage element (assuming, of course, that the values are the same), such as in [9]. Another strategy could choose to not use a physical register to hold the result of an instruction, providing some other means for consumer instructions to get the appropriate values. Such mapping strategies could permit the passing of values from producer to consumer instructions to be carried out in ways that reduce the burden on the physical register storage, and are the crux of the novel physical register usage schemes that we propose in this paper.

2.3 The Outcome: Novel Physical Register Usage

We propose three schemes to optimize the usage of physical registers. The proposed schemes reduce the physical register requirements by eliminating or reducing the duplication of values. An overview of the schemes follows; details are presented in section 5.

In the first scheme, *Physical Register Reuse*, a single physical register is used to hold a given value; multiple instructions producing the same value have their destination logical registers mapped to the same physical register (many-to-one mapping). This scheme is similar in spirit to the scheme proposed by Jourdan *et al.* [9], but the proposed implementation is different. This scheme reduces the number of physical registers required to hold values since it eliminates duplication, but requires a lot of hardware, is very complicated, and is of questionable value. Nonetheless, it serves as a benchmark for comparing our other two schemes which are much more practical, but only reduce the duplication of values, not eliminate it entirely.

The second and third schemes optimize the storage and communication of selected values only. While these schemes could, in general, be used for arbitrary sets of selected values, we focus on the two most common values, namely 0 and 1. These schemes are based on the observation that the full functionality of a physical register file is overkill for storing and passing such values between instructions. If alternative methods could be used for them, the burden on the physical register file could be reduced. In *Registerless Storage*, the register map table is directed to assign 0's and 1's to specific physical registers (*e.g.*, P0 and P1). In this case the name of the storage element is sufficient to know the value; no access to the storage element needs to be performed. The third scheme, *Extended Registers and Tags*, extends each physical register by two bits, and uses a special tag naming scheme that can refer to the physical register and its two bit register extension either as a combined entity or as separate entities. This scheme overcomes a potential drawback of the other two schemes: the need to re-broadcast the tag of a remapped register. The Extended Registers and Tags requires little additional micro-architectural support and is a practical and effective way of partially exploiting the value locality that we observe.

3 Methodology

The results presented in this paper are generated from a timing simulator. The simulator is based on the Alpha ISA definitions and system call emulation of SimpleScalar [3], with a detailed out-of-order timing model. The simulation parameters are listed in table 1, and model pipeline of the core is shown in figure 1(a).

Our experiments were performed on all integer and floating-point programs in the SPEC CPU2000 benchmark suite that were compiled with the Compaq C compiler (with flags `-arch ev6 -fast -O4`). All programs were run to completion on reference inputs except for the results presented in section 4.1. The experimental setup for that result is described later. Value locality results presented in section 4 are based on the values that get written into the physical register file. We exclude accesses to special logical registers (such as logical register `r31` in Alpha that always holds 0), since they are not renamed, and hence are never part of the physical register file.

4 Register Value Locality

In this section we study the value locality that provides the basis for the register file optimizations that we consider in this paper. We start out by determining the common values generated by program instructions, observing that 0 and 1 are the two most frequent values. We then consider the value locality in a window of recently-committed instructions, and in writes to the physical register file. We conclude this section with a measurement of the duplication of values in physical registers.

bzip2	crafty	gap	gcc	gzip	mcf	ammp	art	equake
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
4831843632	7	2	4	4	5368778784	5368710000	2	5368718454
5368712376	3	81	4831839248	32	5	2560	4.59187e+18	1.05557e+14
62914560	5369648560	5	3	2	4831863760	1884672376	4.58549e+18	24577
65536	8	5369767936	52	3	10	3584	3	4831893495
5368712432	2	8	-1	5368758224	32	6656	5370448344	2
32	5369777344	3	59	16	2	5632	3	32
2	6	4	7	-1	49	48	7	1.84467e+19
3	5368862128	16	5	8	-1	14848	10	10

Table 2: Ten most commonly created values for some SPEC CPU2000 integer and floating-point benchmarks

4.1 Common Values

Table 2 presents the ten most common values created by six integer benchmarks (`bzip2`, `crafty`, `gap`, `gcc`, `gzip` and `mcf`) and three floating-point benchmarks (`ammp`, `art` and `equake`) (we don't present results for all benchmarks in this table to improve the readability of the table). Unlike the remainder of the data presented in this paper, the data in the table is gathered by sampling, since the data structures required to track values produced by all instructions are prohibitively large. The results were obtained by sampling at ten evenly-spaced phases during the execution of the program. Each phase collected the occurrence of values for ten million instructions. The values on the top of the table are the most frequently produced values. However, the order does not strictly correlate to the frequency of occurrence of these values in the entire program run, due to sampling inaccuracies.

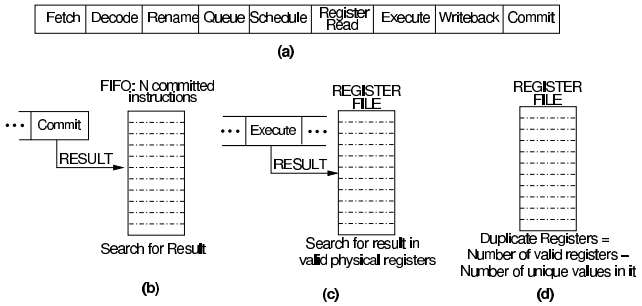

Figure 1: (a) Model processor pipeline. Illustrations on how we measure: (b) value locality in recently committed instructions, (c) value locality in register writes, and (d) value duplication in physical registers

From table 2, we see that the most frequent values across all benchmarks are 0 and 1 (this statement is also true for the benchmarks not presented in the table). This suggests that special *hard-wired* optimization schemes to deal with these two common values may be appropriate, since they are likely to benefit most, if not all, programs. Other frequent values differ among the benchmarks. Values such as 2, 3, and 4 are frequent in some benchmarks (likely generated by induction variables in short loops), but not in others. The large integer values correspond to addresses of frequently-used variables and these are very likely to be different for different programs. Exploiting these would require optimization schemes that track such common values dynamically.

4.2 Value Locality in Committed Instructions

The results in table 2 show the common values occurring in a dynamic execution of program, but do not show the locality of these values. Figure 2 presents the percentage of committed instructions that produce a result that is the same as the value created by one of the past N committed instructions, for N equal to 64, 128 and 256. Due to space constraints, we present only the average percentages for integer and floating-point benchmarks. Figure 1(b) illustrates the measurement.

We see that a large percentage of values are the same as those generated by recent instructions, suggesting opportunities for optimizing the storage to hold these values (similar data has also been presented for the IA-32 architecture in [9]). Each bar separates the contribution due to 0, 1, and other values. For integer benchmarks, about 48% of all instructions produce a value that is the same as a value produce by one of the previous 64 instructions. About 12% of the values are 0's and about 6% are 1's.

4.3 Value Locality in Physical Register Writes

The data of figure 2 are somewhat misleading if we are interested in getting a feel for the potential of physical register usage optimization schemes. This is because the data correspond only to instructions that are committed whereas physical registers are used to store values not only for instructions that are committed, but also for mis-speculated instructions that end up being discarded. Any strategy for optimizing physical register usage would have to consider all the values that would normally get written into physical registers.

Figure 3 presents the integer and floating-point averages of the percentage of executed instructions which create a value that is already present in a live physical register. Figure 1(c) illustrates this measurement. The results are presented for three physical register file sizes of 80, 128 and 160 registers. The contribution due to 0, 1, and other values are separated. The locality observed is lower than that suggested in figure 2 (note the different scales). The height of the bars is a direct measure of the percentage of registers that could be reassigned with Physical Register Reuse (described in section 5.1).

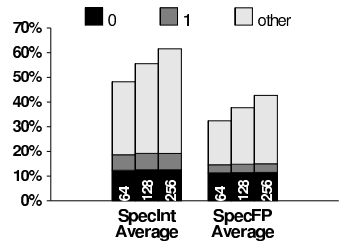

Figure 2: Percentage of results present in a window of results produced by recently committed instructions

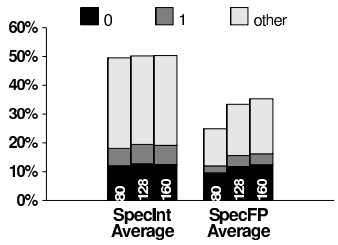

Figure 3: Percentage of results already present in the register file

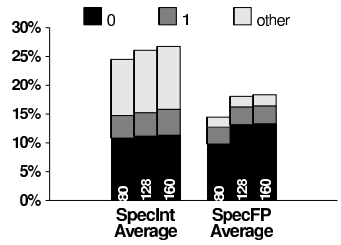

Figure 4: Percentage of values present in more than one physical registers

4.4 Value Duplication in Physical Registers

The data of figure 3 indicates that a significant number of instructions create values that already exist in the physical register file. For example, in integer benchmarks, with 80 physical registers, an average of 49% of the values written into the register file already exist in it; almost half of all writes create a duplicate value, unnecessarily consuming a physical register.

While this data is a direct indicator of the number of physical register mappings that could be reassigned to a different physical register (and yet get the same value), it is not a true indicator of the percentage reduction in the number of physical registers that we could hope to achieve. This is because some of the registers that are counted as holding duplicate values in figure 3 might shortly end up being returned to the free list anyway, as part of a conventional physical register management discipline. Optimizing for such cases is not very effective in reducing the physical register requirements. For example, consider that P3 contains the value 2, and P19 is assigned to a later instruction that also creates the value 2. The results of figure 3 would consider P19 to hold a duplicate value, and thereby be counted towards the possible reduction in physical register requirements. But suppose that, shortly after the value 2 has been written to P19, the normal physical register discipline returns P3 to the free list. P19 no longer holds a duplicate value. Over the long run returning P19 to the free list and holding on to P3 does not noticeably reduce the physical register requirements as compared to holding on to P19 and returning P3 to the free list a few cycles later. A more accurate measure of the exploitable locality would therefore be the percentage of live physical registers that hold duplicate values.

These results are presented in figure 4 (again, we present only the average percentages for integer and floating-point benchmarks), and an illustration of this measurement is shown in figure 1(d). The data is less impressive than the results in figures 2 and 3. However, it is a more realistic measure of the register value locality that we can exploit. With a scheme that assigns only a single physical register to a value we can expect to get by with a smaller fraction of registers corresponding to the data in the figure. For example, starting with a base case of 80 registers, a scheme that assigns a single physical register to only 0's, to 0's and 1's, and to all duplicated values, could reduce the physical register requirements by an average of 11%, 15% and 24%, respectively, for integer benchmarks, and 10%, 13% and 15%, respectively, for floating-point benchmarks. Note that the duplication of values other than 0's and 1's is lower for floating-point benchmarks. A similar trend can also be seen in the previous two measurements (figures 2 and 3).

5 Exploiting Value Locality

We now present three schemes that exploit the value locality presented in previous section to optimize the operation of the physical register file. The schemes — Physical Register Reuse, Registerless Storage, and Extended Registers and Tags — are discussed in sections 5.1, 5.2, and 5.3, respectively.

5.1 Physical Register Reuse

The idea of Physical Register Reuse is to avoid using multiple physical registers to hold the same value. The scheme works as follows. The physical register allocation and register renaming start out conventionally: a physical register is allocated for the result of an instruction, and the destination logical register is mapped to this physical register. When the result of the instruction becomes available, a check is performed to see if the value already resides in the register file. If it does, the destination logical register is *remapped* to the physical register that contains the value. Actions are taken to ensure that dependent instructions read from the remapped register, and the previously-allocated physical register for the result is returned back to the free pool. If the value is not present in the register file, no special actions are taken. Several options are possible to accomplish this remapping of dependent instructions. We discuss one implementation here, an illustration of which is shown in figure 5.

To determine the presence of a value in the physical registers we maintain a *Value Cache*. The cache is looked up immediately after the instruction is executed (shown in figure 5). The cache is a CAM structure that contains mapping of values to physical register tags; a match returns the tag of

the physical register that contains the value. If no match is found, an entry is added to the cache. The size of the cache is bounded by the number of physical registers. Appropriate invalidation actions are taken when a physical register is returned back to the free list or when a register can no longer be reused (described next).

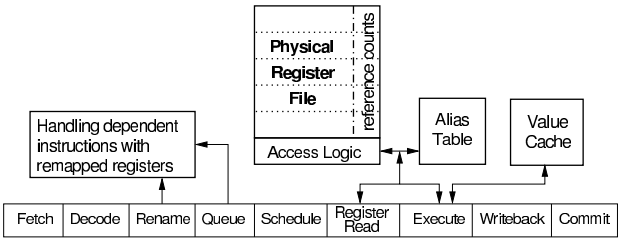

Figure 5: Processor pipeline with Physical Register Reuse

To account for the many-to-one mapping of logical to physical registers, each physical register has a *reference count* associated with it. The counter is incremented when the physical register is mapped and decremented when it is freed. The register is returned back to the free list when the counter is zero. If the counter saturates, the physical register should no longer be reused (the corresponding entry in the Value Cache is invalidated).

Actions need to be taken to handle instructions dependent on a register that has been remapped to a different physical register. Dependent instructions that have not yet entered the instruction window are taken care of by updating the rename map when remapping a physical register; these instructions get the identity of the new physical register directly from the updated rename map. There are two options to handle dependent instructions that have been renamed before the execution of their parent instruction. Both options are initially best thought of as modifications to the traditional *wakeup* operation; in addition to waking up the waiting instructions the operation will also update some additional state information. Later, we will see that we might need a separate wakeup-like operation for these state updates, thus leaving the normal wakeup operation as is.

The first option is to broadcast both the old and the new physical register tags during wakeup. Instructions that match the old physical register tag update their source physical register tag with the new tag, and then use this tag to get their value from the physical register file when they are scheduled for execution. Modern micro-architectures, however, use the notion of *speculative scheduling*—dependent instructions are woken up not when an instruction completes but when an instruction is scheduled for execution—so that dependent instructions can execute in back-to-back cycles. This poses a problem as the result of an instruction, and therefore a potentially new physical register tag, is not available at the time of the broadcast for instruction wakeup. We can solve this problem with two broadcasts: a conventional broadcast to wakeup dependent instructions for speculative scheduling, and a *re-broadcast* with the new physical register tag so that dependent instructions that have yet to be scheduled can update their source tags. Dependent instructions that have already been scheduled get their values from bypass paths in either case, and typically do not access the register file anyway. Since the broadcast for instruction wakeup is not modified, no overhead is added to the critical path of instruction scheduling. However, the broadcast paths of the processor need to be enhanced to broadcast both the old and the new physical register tags during re-broadcast.

The second option is to use an *Alias Table*, a structure that contains <instruction id, old physical register, new physical register> tuples. This table is used to redirect references for <old physical register> to <new physical register>. The size of the table is bounded by the product of the number of times a physical register can be reused and the number of physical registers in the register file. An entry is added to the table when a physical register is remapped[1]. The broadcast for instruction wakeup happens as before. Likewise, a re-broadcast is used to inform dependent instructions that their source physical register has been remapped. Unlike the first option, however, this re-broadcast does not include the new physical register tag. Rather this re-broadcast is used to inform the (matching) source operand identifiers that they need to consult the Alias Table to get the identity of the new physical register. This option trades off the degree of modification to the broadcast paths with the need to access an Alias Table for these dependent instructions (with a possible increase in the latency to access operands due to the level of indirection).

We next describe the steps that need to be taken to recover from exceptions. Reference counts of the physical registers are kept consistent by saving the counts of the mapped physical registers during the commit stage. This handling mechanism is similar to that for the architectural rename map, which is also maintained at the commit stage for recovery from exceptions. The Value Cache and Alias Table structures are kept consistent by validating only those entries that have physical register tags mapped to a logical registers during exception recovery.

To summarize, the micro-architectural changes proposed for this scheme are non-trivial: reference counts and their management, a Value Cache, an Alias Table, and significant modifications to other logic. This complexity and additional overhead is perhaps the reason why the concept of mapping multiple logical register to a single physical register has not received much attention since the initial work of

[1] A corner case exists where a physical register that is remapped and freed could be assigned to a subsequent instruction and again be remapped. This could introduce multiple entries in the Alias Table with the same <old physical register>. We handle this case with <instruction id>, but do not discuss this in the paper.

Jourdan *et al.* [9]. However, the framework of this scheme provides us with an upper limit: assuming that the additional structures do not degrade the latency of normal operation and there are no limits on the number of times a physical register can be reused, this scheme completely eliminates the duplication of values in the physical register file and serves as an ideal implementation of Physical Register Reuse. The next two schemes trade off the ability to eliminate the duplication of arbitrary values with the complexity of the hardware required.

5.2 Registerless Storage

The objective of *Registerless Storage*, is to avoid duplication of selected values that we have decided *a priori*. For our purpose these values are 0 and 1 since, as we observed in section 4, these values are the most common and account for a significant portion of the duplicate values in the physical register file. We describe this scheme using the framework of Physical Register Reuse that we developed in the previous section. We will see that by concentrating on values that have been selected *a priori*, most of the complexities of Physical Register Reuse are eliminated. The drawback, of course, is the restriction that only the duplication of these pre-selected values (*i.e.*, 0's and 1's) can be eliminated.

We start out by reserving physical register tags for the special values (*e.g.*, P0 for 0 and P1 for 1). If the result of an instruction is 0 or 1, its destination logical register is remapped to P0 or P1, respectively, and the previously-mapped physical register is freed. Since we are only interested in tracking the duplication of 0's and 1's, a dynamic tracking mechanism that can check for the duplication of arbitrary values (*i.e.*, a Value Cache) is not needed. Moreover, since only physical registers containing 0 and 1 are going to be remapped, reference counts are not needed for physical registers that hold other values. Finally, since P0 and P1 are never going to contain other values, they are not going to be returned to the free list, and therefore they do not need reference counts either. We refer to P0 and P1 as physical registers solely for purposes of explanation. They are simply a way of naming the values 0 and 1, and if there are other more convenient ways of naming these values (*e.g.*, state bits), those could be used instead.

We now consider providing values to dependent instructions. We add two state bits to the source operands of waiting instructions; these bits track whether the operand is <0>, <1>, or <some other value>. These bits are initially set to <some other value>. As in Physical Register Reuse, the broadcast for instruction wakeup happens as before but the two options for re-broadcast effectively collapse into one. The re-broadcast operation broadcasts the old physical register tag and the value of these state bits, which are <0>, <1> or <some other value> if the result is 0, 1, or some other value, respectively. Source operands that match the tag update their state bits accordingly; the old physical register tag is left unchanged. When the instruction is scheduled, the state bits are used to determine if the operand value is 0, 1, or should be read from the physical register file. Since there is no redirection of physical registers, an Alias Table is not needed.

From the above description we see that if we are only interested in exploiting value locality of the most common values (0's and 1's) the micro-architectural changes required are greatly simplified. Over a traditional physical register file mechanism, we need only: (i) the ability to update the register rename map when the result of an instruction is one of these selected values, (ii) additional state bits to the instruction window entries to keep track of the selected values, and (iii) suitable modifications of the control logic. However, in addition to the normal broadcast for instruction wakeup, a re-broadcast is still necessary when a physical register is remapped.

5.3 Extended Registers and Tags

The goal of our third scheme, *Extended Registers and Tags*, is to eliminate the re-broadcast operation of Registerless Storage by using a novel physical register organization and naming scheme. Associated with each physical register (64 bits in our case) is a two-bit wide *register extension*. One bit of the register extension is a *value bit*, used to hold a 0 or 1, and the second bit is a *valid bit*, to indicate if the register extension is being used to hold a value. The idea in this scheme is to hold the result of an instruction in a register extension if it is 0 or 1, freeing up the associated physical register to hold the result of another instruction. The novel storage naming scheme directs dependent instructions to either the physical register or the register extension, without having to alter the source register tags of those instructions. As the tags do not need to be altered, a re-broadcast operation is not needed.

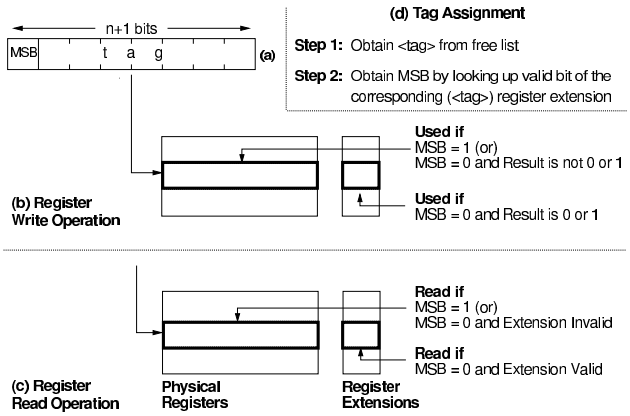

Figure 6: Illustration of: (a) tag, (b) register write operation, (c) register read operation, and (d) tag assignment in Extended Registers and Tags scheme

We describe the operation of the scheme in several steps. First, we present the changes to the register tags. Then, we see how they are used to direct write and read operations to the appropriate storage location (the physical register or the register extension). Finally, we consider free list and tag management.

The size of the register tags is increased by one bit, from n to n+1 bits (figure 6(a)); n bits are used to select a <physical register, register extension> tuple, and the MSB is used to decide whether to refer to: (i) the physical register (MSB=1) only, or (ii) to either the physical register or its corresponding register extension (MSB=0). The MSB is used to direct write and read accesses to the storage tuple as we see next.

Write operations proceed as illustrated in figure 6(b). If the result of an instruction is 0 or 1, we would like to use the register extension to hold the value if possible, and free the associated physical register. Thus, the value is written into the selected register extension if it is available (*i.e.*, its valid bit is currently not set). The valid bit is then set to indicate that the register extension is being used. The n-bit tag for the tuple is returned to the free list, thereby permitting the physical register associated with the tuple to be used to store the results of other instructions. If the result is a value other than 0 or 1, or if the result is 0 or 1 but the register extension is not available, the value is written into the selected (64-bit) physical register as normal; the register extension is left as is.

Read operations proceed as illustrated in figure 6(c). If the source tag of an instruction's operand refers to a physical register (MSB=1) within the selected tuple, the operation proceeds normally. If it refers to a physical register or its register extension (MSB=0), further information is needed to direct the read to the appropriate storage in the selected tuple. This information is the valid bit of the corresponding register extension. If the register extension is valid, the read is directed to the register extension; otherwise it is directed to the physical register.

We now consider free list and tag management. The free list contains n-bit tags that refer to a <physical register, register extension> tuple. When a tag is assigned to the destination register of an instruction during the rename process, one of the corresponding storage elements (physical register or the register extension) is where the result of the instruction will eventually reside. The MSB is prepended based upon where in the extended register the result may reside. If the valid bit of the corresponding register's extension is set, the register extension is already being used to hold 0 or 1, the result of some prior instruction. In this case, the full physical register must hold the result, and an MSB of 1 is prepended. Otherwise, an MSB of 0 is prepended indicating that either the physical register or the register extension can be used to hold the result; which one of these is actually used will be determined when the result is available (*i.e.*, when the corresponding write operation happens, as above). This set of operations is outlined in figure 6(d).

When a physical register is freed, the corresponding n-bit tag is returned to the free list as usual. Note that a physical register can be freed while its register extension is still in use. The register assignment scheme we have described ensures that subsequent users of the physical register will not be eligible to use the extension. Consumers of the extension will have a different source tag (MSB=0) than those of the full register (MSB=1). When a register extension is freed, its valid bit is reset; there is no special free list for the register extensions.

This scheme places 0's and 1's in the register extensions whenever possible, thus freeing the main physical registers to be used for other values. A value of 0 or 1 is placed into a physical register only if the corresponding register extension is already in use. Also, the register extension can be used to hold a 0 or a 1 only if the corresponding physical register is not being used at the time the instruction is renamed. Thus, this scheme is able to eliminate most, but not all, of the duplication of 0's and 1's in the physical registers.

Once a tag has been chosen for the source operands of an instruction, it is not remapped, eliminating the need for a re-broadcast to update the tag. Rather, the tag is used to steer the access to either a physical register or its register extension based upon the value of the valid bit of the register extension and the MSB of the tag.

6 Evaluation

Table 3 summarizes the actions taken by the three different schemes. A common benefit of these schemes is better register utilization because of reduced register requirements. Physical Register Reuse achieves this by avoiding duplication of all values in the physical register file. Registerless Storage and Extended Registers and Tags achieve this by avoiding or reducing, respectively, the duplication of 0's and 1's. We discuss and quantify the performance impact of this reduction in section 6.1.

An additional benefit is the possible reduction in the number of accesses to the register file. With Physical Register Reuse, a reduction in register write traffic is achieved because of register reuse: when a physical register is reused, the result of the instruction need not be written to a register. The non-trivial micro-architectural changes along with the added sizes and latencies of the Alias Table and the Value Cache lookups make this benefit questionable and therefore we do not quantify it. Registerless Storage reduces register read and write operations by not storing 0's and 1's in physical registers. We quantify this in section 6.2. The Extended Registers and Tags scheme does not reduce the overall register traffic. Rather, it distributes the traffic amongst the physical registers and the register extensions, which can

	Physical Register Reuse	Registerless Storage	Extended Registers & Tags
Detecting locality	Value Cache	Identify if result is 0 or 1	Identify if result is 0 or 1
Exploiting locality	Free assigned physical register. Take actions needed to reuse the physical register that already holds the value.	Free assigned physical register.	If assigned a physical register with its extension, write value (0 or 1) to the extension. Free the physical register.
Handling dependent instructions	Update rename map. Broadcast new physical register tag or use Alias Table to handle dependent instructions waiting in instruction window.	Update rename map. Broadcast result (0 or 1) to dependent instructions in instruction window.	No changes needed.
Handling exceptions	Recover physical register reference counts, Value Cache and Alias Table.	No changes needed.	No changes needed.
Outcome of technique	Register file with unique values.	Register file without the most-common values (0's and 1's).	Most common values (0's and 1's) in register extensions. Physical registers used for other values, and for 0's and 1's if corresponding register extension unavailable.

Table 3: Actions taken in the implementations of Physical Register Reuse, Registerless Storage and Extended Registers & Tags

be physically separate. Most of the reads and writes of 0's and 1's are to the register extensions; other references go to the physical registers.

6.1 Performance

There are several ways in which we can assess the performance impact of our proposed schemes. To keep the discussion manageable, we assess the performance impact using two sets of experiments. In the first set of experiments we increase the number of in-flight instructions by increasing the instruction window size while keeping the size of the physical register file constant. In our second set of experiments we keep the instruction window size fixed while reducing the number of physical registers. In both cases, we have the size of the instruction window larger than the number of physical registers. This is done to increase capacity pressure on the physical registers, allowing us to distinguish between the different physical register file designs. Schemes that use the physical register storage more efficiently will allow more instructions to be in-flight, increasing the available parallelism and thus performance. One outcome of efficient physical register usage schemes, we believe, will be a micro-architecture where the size of the instruction window is somewhat larger than the number of physical registers.

Figure 7 presents the relative performance for four different cases for our first set of experiments. Case 1 is an ideal implementation of Physical Register Reuse that assumes zero-cycle access latencies for the Alias Table and Value Cache lookup, with no limits on the number of times a physical register is reused. Case 2 and case 3 are the Registerless Storage and the Extended Registers and Tags schemes, respectively. Case 4 is a perhaps somewhat practical implementation of Physical Register Reuse that has a 1-cycle Alias Table access latency with the access and register read operation pipelined, no overhead to access the Value Cache (0-cycle access latency), and with three bits for reference counting (our experiments indicate that more than three bits are not needed for reusing most values in the register file). We present our results in the form of stacked bars. Each portion of the bar represents the additional improvement that the scheme contributes and the height of the bar represents the performance improvement with respect to the base case.

The bars in the figure present the performance relative to a base case machine with 128 physical registers and with no optimizations. All cases have a 256 entry instruction window for in-flight instructions. We see that practical Physical Register Reuse (case 4) is the worst scheme among all cases, actually decreasing performance for half of the benchmarks. This is because register operations are critical to performance, and a one cycle penalty for register read operations on some instructions has a significant negative impact. The ideal implementation of Physical Register Reuse is the best option, significantly improving performance for many integer benchmarks. However, for floating-point benchmarks the benefit of this scheme over schemes that are specialized for 0's and 1's is negligible. This is because of the insignificant percentage of duplicate values other than 0's and 1's (figure 4). The interesting part of the figure is the impressive results for optimizations that target 0's and 1's. Registerless Storage does quite well with an average performance increase of of 6.6% for integer benchmarks and 4.5% for floating-point benchmarks. Extended Registers and Tags also achieves good performance improvement (average of 6% and 4% for integer and floating-point benchmarks respectively), while being much easier to implement.

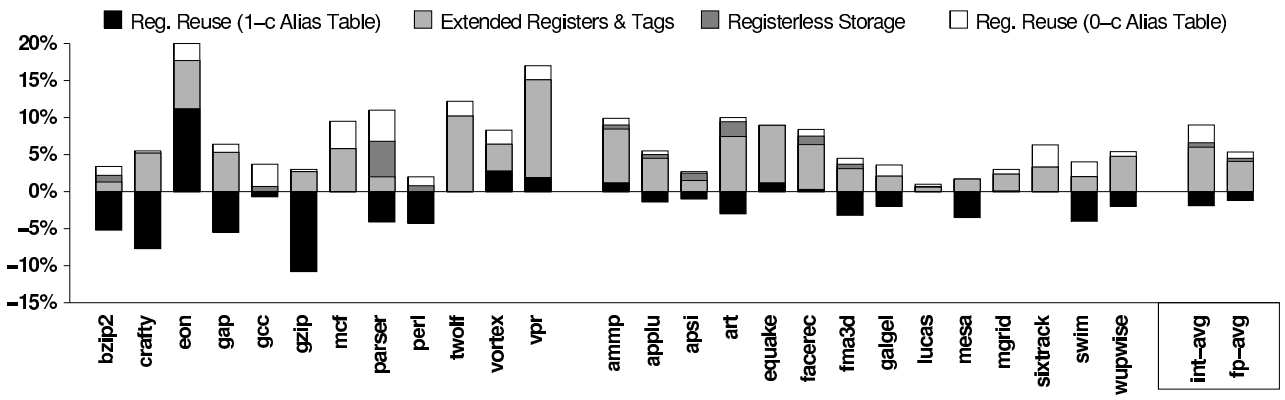

Figure 7: Relative performance of different schemes with increased instruction window size

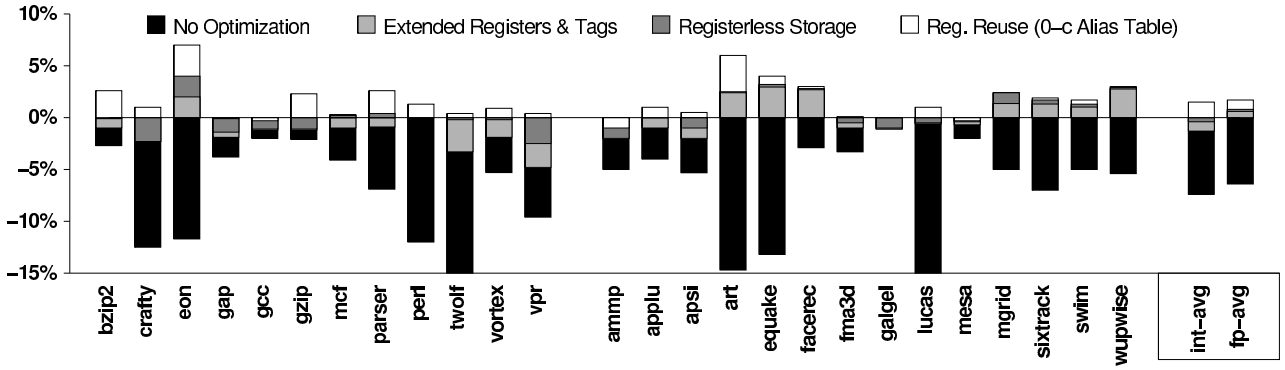

Figure 8: Relative performance of different schemes with reduced physical register file

Figure 8 presents the data for our second set of experiments. Performance is presented as stacked bars for four cases with respect to the base case (128 physical registers with no optimization). All four cases have a smaller physical register file with 100 registers. Case 1 is ideal Physical Register Reuse. Case 2 and 3 are Registerless Storage and Extended Registers and Tags, respectively. Case 4 is an unoptimized physical register file. As the figure suggests, having 100 physical registers with no optimization degrades performance significantly in many cases. Using any of the proposed optimizations results in better use of the available physical registers. A physical register file with ideal Register Reuse performs roughly equivalent to a larger register file with no optimization.

Again, the results for the Registerless Storage and the Extended Registers and Tags schemes are impressive. Both schemes allow a physical register file with 100 registers to perform only slightly worse than an 128-entry conventional register file. This represents approximately a 20% reduction in physical register file size for equivalent performance.

A limitation of the Extended Registers and Tags scheme mentioned earlier is that it does not entirely eliminate the duplication of 0's and 1's since these values can not always be stored in the register extensions. Despite this limitation, this scheme performs reasonably well when compared to Registerless Storage on most benchmarks (note the insignificant additional improvement of Registerless Storage over Extended Registers and Tags in figures 7 and 8). Results not presented in this paper indicate an average of 2.4% (2% for integer and 2.8% for floating-point benchmarks) of the register writes of values 0 and 1 are *not* written to a register extension because of its unavailability. This is an insignificant percentage, and the vast majority of the total register writes that are 0's and 1's are directed to the extensions. Therefore, this limitation does not have a detrimental impact on the performance.

6.2 Reducing Register Accesses

We now consider the reduction in register file accesses that are possible in the Registerless Storage scheme. Figure 9 presents the percentage of register file write operations that can be eliminated. Each bar shows the number of writes of 0's and 1's to the register file with 160 physical registers. We see that an average of 17.6% (19.5% for integer, and 16% for floating-point benchmarks) of the writes to the register file can be eliminated. Results not presented in this paper indicates an average increase of 4.7% in the number of writes to the rename map needed due to register reassignment (see section 5.2).

Figure 10 presents the percentage of read operations that

could be eliminated. The data in the figure considers only register reads; source operands that are obtained from the bypass path are not included. Each bar separates the contribution of 0's and 1's for a physical register file of 160 registers. An average of 7.5% (7.7% for integer, and 7.4% for floating-point benchmarks) of the reads can be eliminated. The data is less impressive, but nevertheless, a technique that might be worth pursuing because of potential power advantages.

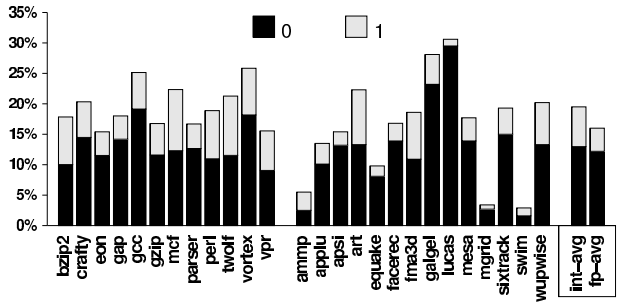

Figure 9: Percentage of register writes that are 0's and 1's

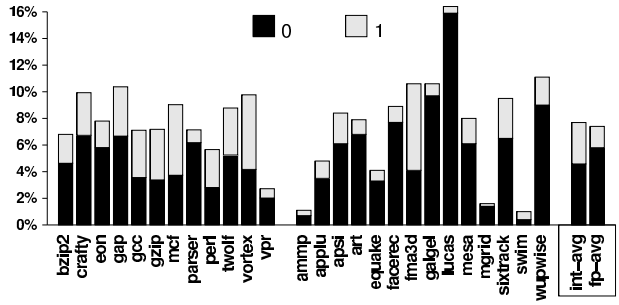

Figure 10: Percentage of register reads that are 0's and 1's

7 Related Work

Register files have been the subject of study for a long time, and several approaches to optimize register files have been investigated. We categorize previous work into three broad categories. The first category of work exploits locality of access to build hierarchies of register files. The idea is to have a small, fast first level, backed up by a larger, slower second level. This hierarchy can be determined either statically or dynamically. The Cray architectures used two levels of architectural registers [16]. Static register file hierarchies were also explored by Swensen and Patt [18]. Dynamic register file hierarchies have been investigated in several recent papers [1, 5, 14].

The second category of work uses localities of communication to create clustered register files. The idea is to use multiple register files, each with few read/write ports, to collectively create a register file with larger bandwidth [15, 19]. Different register files in a cluster can service different requests in parallel as long as inter-operation value communication is within a cluster; a penalty occurs when a value is passed from one cluster to another. Again, clustering could be done statically or dynamically. VLIW machines have used statically-clustered register files, with each cluster being a separate register name space [4]. The distributed register file in a Multiscalar processor is dynamically clustered; the different register files in the cluster are a collection of multiple future files [2]. An alternate form of dynamic clustering is used in complexity-effective architectures [13] and in clustered architectures such as the Alpha 21264 [10].

The third category of work uses novel physical register rename mapping strategies to optimize physical register file design. The pioneering work in this area was the work by González *et al.* on virtual-physical registers [8, 12]. Delaying the allocation of a physical register reduces the lifetime of a physical register and thereby the physical register requirements. Jourdan *et al.*'s work [9] enhances the concept of virtual-physical registers to exploit value locality. It introduces the ideas of physical register reuse and many-to-one mappings in the register rename map to reduce the physical register file size.

Previous work, with the exception of Jourdan *et al.* [9], is orthogonal to the work presented in this paper. Most of these ideas can be utilized in concert with the ideas presented in this paper. Our work is a follow on to the work presented in [9]; it differs from that in [9] in several ways. We present different, and more realistic, measurements of exploitable value locality, and our proposed schemes are different. Our schemes that propose special treatment for the most common values (0 and 1) allow us to trade off theoretical potential with ease of implementation.

8 Summary and Conclusions

This paper considered the use of value locality to optimize the design of physical register files. We observed that a value produced by an instruction has a high probability of being the same as a value produced by another recent instruction. This results in the same value being present in multiple physical registers with a traditional physical register management discipline. We also observed that the values 0 and 1 are the most frequently generated values across all the benchmarks we studied. These values also account for a significant amount of the duplication in the physical register file.

We proposed three schemes to exploit the value locality and thereby avoid the duplication of values. Our first scheme, Physical Register Reuse, eliminates all duplication of values in the physical register file, but requires significant micro-architectural support and is quite complicated. Nevertheless, it serves as a benchmark to evaluate the potential of eliminating value duplication. Our other two schemes,

Registerless Storage and Extended Registers and Tags, concentrate only on eliminating or reducing the duplication of 0's and 1's. This restriction significantly simplifies the additional micro-architectural support that is required.

We presented a performance evaluation of the three schemes. Our results indicate that Physical Register Reuse is able to reduce the physical register requirements significantly, but is only able to translate these reduced requirements into performance benefits under idealistic assumptions. With realistic assumptions this scheme actually results in a performance degradation. The Registerless Storage and Extended Registers and Tags schemes are able to reduce physical register requirements to a lesser extent than Physical Register Reuse, but can achieve performance benefits because they have little additional complexity. The Registerless Storage scheme is also able to reduce the number of register file reads and write by an average of 7.5% and 17.6%, respectively.

Acknowledgments

We would like to thank Adam Butts, Koushik Chakraborty, Jichuan Chang, Paramjit Oberoi, Philip Wells and Craig Zilles for their comments on earlier versions of this work. We also thank Adam Butts for discussions on ideas in this paper, Antonio González and Mateo Valero for making us aware of the problem of re-broadcasting the tags, and Craig Zilles for providing his simulation infrastructure.

This work was supported in part by National Science Foundation grants CCR-9900584 and EIA-0071924, donations from Intel, and the University of Wisconsin Graduate School.

References

[1] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proc. of the 34th Annual Intl. Symp. on Microarchitecture*, pages 237–248, 2001.

[2] S. Breach, T. N. Vijaykumar, and G. S. Sohi. The anatomy of the register file in a multiscalar processor. In *Proc. of the 27th Annual Intl. Symp. on Microarchitecture*, pages 181–190, 1994.

[3] D. C. Burger and T. M. Austin. The Simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, 1997.

[4] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proc. of the 2nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, 1987.

[5] J.-L. Cruz, A. González, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *Proc. of the 27th Annual Intl. Symp. on Computer Architecture*, pages 316–325, 2000.

[6] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing cycle time through partitioning. In *Proc. of the 30th Annual Intl. Symp. on Microarchitecture*, pages 149–159, 1997.

[7] K. I. Farkas, N. P. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. In *Proc. of the 2nd Intl. Symp. on High-Performance Computer Architecture*, pages 40–51, 1996.

[8] A. González, J. González, and M. Valero. Virtual-Physical Registers. In *Proc. of the 4th Intl. Symp. on High-Performance Computer Architecture*, pages 175–184, 1998.

[9] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *Proc. of the 31st Annual Intl. Symp. on Microarchitecture*, pages 216–225, 1998.

[10] R. E. Kessler. The Alpha-21264 Microprocessor. *IEEE Micro*, 19(2):24–36, Mar./Apr. 1999.

[11] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proc. of the 29th Annual Intl. Symp. on Microarchitecture*, pages 226–237, 1996.

[12] T. Monreal, A. González, M. Valero, J. González, and V. Vinals. Delaying physical register allocation through Virtual-Physical Registers. In *Proc. of the 32nd Annual Intl. Symp. on Microarchitecture*, pages 186–198, 1999.

[13] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. of the 24th Annual Intl. Symp. on Computer Architecture*, pages 206–218, 1997.

[14] M. Postiff, D. Greene, S. Raasch, and T. Mudge. Integrating superscalar processor components to implement register caching. In *Proc. of the 15th Intl. Conf. on Supercomputing*, pages 348–357, 2001.

[15] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens. Register organization for media processing. In *Proc. of the 6th Intl. Symp. on High-Performance Computer Architecture*, pages 375–386, 1999.

[16] R. M. Russell. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, Jan. 1978.

[17] A. Sodani and G. S. Sohi. An empirical analysis of instruction repetition. In *Proc. of the 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 35–45, 1998.

[18] J. A. Swensen and Y. N. Patt. Hierarchical registers for scientific computers. In *Proc. of the 2nd Intl. Conf. on Supercomputing*, pages 346–354, 1988.

[19] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques*, pages 179–185, 1996.

[20] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *Proc. of the 33rd Annual Intl. Symp. on Microarchitecture*, pages 258–268, 2000.

[21] R. Yung and N. C. Wilhelm. Caching processor general registers. In *Proc. of the Intl. Conf. on Computer Design*, pages 307–312, 1995.

[22] H. Zhou, J. Flanagan, and T. M. Conte. Detecting global stride locality in value streams. In *Proc. of the 30th Annual Intl. Symp. on Computer Architecture*, pages 324–335, 2003.