Exploiting Value Locality in Physical Register Files

Saisanthosh Balakrishnan
Gurindar Sohi

Technical Report #1439

April 2002

1439

Exploiting Value Locality in Physical Register Files

Saisanthosh Balakrishnan and Gurindar Sohi
Computer Sciences Dept., University of Wisconsin-Madison
1210 West Dayton St., Madison, WI 53706
{sai, sohi}@cs.wisc.edu

Abstract

The physical register file is an important component of a dynamically-scheduled processor. Increasing the amount of ILP (issue width, instruction window size) places increasing demands on the physical register file, calling for alternative physical register file organizations and management strategies. This paper considers the use of value locality to optimize the operation of a physical register file.

We observe that the value produced by an instruction is often the same as a value produced by another recent instruction, resulting in the same value present in multiple physical registers at the same time. By allocating a single physical register for a value, and by altering the register rename mapping accordingly, we can reduce the physical register requirements. We further observe that the number of writes to the physical register file can be reduced by suppressing the writes of values already present in the register file. Furthermore, the number of reads can be reduced by using other means to obtain the value in special cases.

We present optimizations for the special cases of the values 0 and 1. These are the most frequently-occurring values across the spectrum of programs and also the most suitable ones for optimization. We show how the physical register file size, as well as the number of read and write accesses, can be reduced with trivial microarchitectural modifications by optimizing for 0's and 1's.

Keywords: *value locality, physical register files*

1 Introduction

The physical register file is an important component of the microarchitecture of a modern dynamically-scheduled processor. As processors exploit more instruction level parallelism (ILP), even more demand is placed on the physical register file: larger instruction windows require storing the results of more instructions in the register file, wider instruction issue requires higher bandwidth access to the stored values. A straightforward approach to respond 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 ports to provide additional bandwidth. This approach results in large, monolithic storage arrays: the size grows linearly with the number of registers, and greater than linearly [1,7,19] with the number of ports. This approach is also wire-intensive (wire lengths increasing with the storage array size), and incompatible with upcoming trends in processor design [14,17].

Processor architects have realized the lack of scalability of monolithic storage arrays for implementing a physical register file, and have explored alternative ways. The rationale for these alternative designs is the exploitation of different forms of *locality*: characteristics of the reference patterns to the register storage. Two different forms have received a lot of attention: (i) locality of access, and (ii) locality of communication. Locality of access has been used to explore hierarchical register files and register caches [5, 18, 20, 25]. Localities of communication have been used to design clustered register files [6, 10, 24].

More recently *value locality* has been presented as an interesting opportunity for optimizing physical register file design [9]. This paper considers how *value locality* could be exploited to optimize physical register file design. We make two empirical observations, and present physical register file designs that exploit these observations. The first observation is that the same value is present in multiple physical registers simultaneously. We exploit this observation with a physical register file design where only a single physical register is assigned to a given value. This reduces the physical register requirements. The second observation is that many of the values written into, and read from, the physical register file are statically-determined "special" values, in particular 0 and 1. We exploit this observation to simplify the hardware required to reduce the physical register requirements, and additionally reduce the number of register file writes and reads.

The remainder of the paper is as follows: in Section 2 we review architectural and physical register files to set the stage for observing the phenomenon of value locality and overview our evaluation methodology in Section 3. Section 4 describes the locality of interest and presents quantitative measurements of the locality. Alternatives for reducing register file size, and register file accesses are presented and evaluated in Sections 5 and 6, respectively. We present related work in Section 7 and conclude in Section 8.

2 Architectural and Physical Register Files

To set the stage for exploring alternatives for designing a register file, we start out with a discussion of the purpose of architectural and physical register files, making key observations that could be exploited to devise alternate physical register file designs.

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). With only logical registers, the logical register name space is used to link a producer instruction with consumer instructions. A logical register is used to hold the value created by the producer instruction, and consumer instructions can access the value by accessing the storage associated with the appropriate logical register.

With physical registers, the process of passing values from producer to consumer instructions is slightly different. The logical register name space is used to link producer and consumer instructions, but the values are stored in physical registers. A register rename map provides a level of indirection, redirecting logical register references to the corresponding physical register. When a producer instruction creates a value, it stores it in a given physical register. When an instruction needs a value, it accesses the value from the physical register to which it is directed by the register rename map.

The conventional register rename mapping maps one logical register to one physical register (one-to-one) in the absence of speculative execution, and one logical register to many physical registers (one-to-many) with speculative execution. The mappings reflect a straightforward relationship between the number of in-flight instructions, and the storage needed to hold their results.

For alternative physical register file designs, different ways of dealing with values created by producer instructions, and ways of providing values to consumer instructions, need to be developed. The key to these ways is in the novel use of the rename map. The rename map is a level of indirection that directs requests for values contained in logical registers to specific physical registers. It could easily be altered to service requests for values in logical registers in arbitrary ways. The flexibility of the mapping provides opportunities for optimization: many optimizations could be effected by altering the mapping.

One recently-proposed optimization that utilizes the flexibility of the rename mapping is the concept of virtual-physical registers [8, 15]. Here a virtual name for a physical register is provided at the register rename stage, but the actual physical register not allocated until the instruction has completed execution. At that point, the virtual-physical register name is mapped to the actual physical register allocated, and references to the virtual-physical register are directed to the actual physical register. This decreases the duration for which a physical register is held (it is not held between rename and the completion of an instruction), and consequently decreases the physical register requirements.

The two optimizations that we explore in this paper also use the flexibility provided by the rename mapping. Value locality, which we discuss in Section 4, provides the empirical rationale for the optimizations, and a suitably altered register rename mapping scheme provides the mechanism.

First, if the result of an instruction is already known to be present in a physical register, then a separate register is not needed to hold the result. Accesses to the result of the instruction can simply be directed to the physical register which contains the result, by altering the register rename map [1]. This requires the register mapping to be capable of mapping many logical registers to a single physical register (many-to-one). The consequences of this optimization are twofold: (i) a reduction in the number of

[1]The actions are somewhat more involved than this simple statement. We explore details in Section 5.

physical registers to hold values, since different physical registers are not used to hold the same value, and (ii) a reduction in the number of writes to the physical register file, since a value need not be written if it already exists in the register file.

Second, the mapping can be directed to assign commonly-occurring values to specific physical registers. For example, physical registers P0 and P1 could be assigned to hold the values 0 and 1, respectively. In this case the name of the storage element (*e.g.*, P0) is sufficient to know the value contained in the storage element; no access to the storage element need be performed. The result is a reduction in the number of reads from the physical register file.

3 Methodology

All the results in this work were generated by a complete timing simulator of the Alpha architecture loosely based on SimpleScalar [3] Toolkit. The simulated processor parameters of the base case are shown in Table 1. The only parameter that is changed is the number of physical registers, and the instruction window size is increased to be at least as large as the number of physical registers.

Execution Core	4-wide machine with 128 entry instruction window, a full complement of simple integer units, 2 load/store ports, single complex integer unit, all fully pipelined. The pipeline depth is 14 stages.
Caches	The first-level data cache is a 2-way set-associative 64KB cache 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-cycles access.
Front End	64 KB Instruction cache, 64 KB gshare branch predictor, 64 entry return address stack. The front end can fetch past taken branches. All nops are removed without consuming fetch bandwidth.

Table 1: Simulated Processor Parameters

Our experiments were performed on SPEC2000 integer benchmark programs. The programs were compiled with Compaq C compiler (version 5.9-005), with optimizations (`-arch ev6 -fast -O4`), and were run on reference inputs. All benchmark programs were run to completion. Data presented is for the entire benchmark run, except when specifically noted. All measurements include only values bound to general purpose registers; they exclude accesses to logical registers holding special values (*e.g.*, R31 holding 0 in Alpha).

4 Value Locality

The concept of value locality has been the subject of extensive study in recent years. The work on value prediction exploits value locality to predict the outcome of an instruction [11,12,21,28]. The work on instruction reuse exploits value locality to reuse the result of a previous execution of an instruction [22, 23]. Both value prediction and instruction reuse exploit the fact that many static instructions produce results that are amenable to prediction/reuse; the locality measured is the locality in the dynamic results of a single static instruction.

The value locality that we are concerned with in this paper is different. We want to optimize the storage requirements for a dynamic window of instructions, by using a single storage element to hold the results of multiple dynamic (and likely multiple static) instructions. Thus, the value locality in which we are interested is the commonality of results of different instructions that are in proximity in the dynamic instruction stream [9].

Further, we want to treat certain values (0's and 1's) specially. Thus, we are interested in the prevalence of 0's and 1's in the computed values. This form of locality is related to the locality — the presence of zero bytes — used to achieve compression in memory and caches [32], or to reduce power consumption in caches [26, 29].

4.1 Model for Exploiting Value Locality

Before presenting measurements of value locality, we briefly consider the model for exploiting value locality. We are interested in reducing the pressure on the physical register file. This includes reducing the demand for physical registers, as well as the read and write accesses to the register file. Reduced demand for physical registers results in a smaller register file, or better performance with the same size register file. Fewer accesses imply the possibility of reducing the number of read and write ports on the register file and its power consumption.

The schemes that we develop to reduce the physical register file size will use a single physical register to hold a value, by altering the register rename mapping. They will reduce the number of register file writes by avoiding writing a value that already exists in the physical register file. They will reduce the number of register file reads by not accessing the physical register file to get a known value (*e.g.*, 0 or 1). The value locality measurements that we present next are geared towards data that measure the potential of the above approaches.

4.2 Commonly Occurring Values

Table 2 presents the 15 most frequent values created by instructions for several of the SPEC 2000 benchmark programs. (We don't present results for all benchmarks to improve readability of the table.) Unlike the remainder of the data presented in this paper, the data in Table 2 is gathered by sampling since the data structures required to track all values generated are impractically large. The common results shown in the table were obtained by sampling at ten evenly-spaced phases during the execution of the program. Each phase collected the occurrence of values for 10 million instructions. The values on the top of Table 2 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 errors[2].

From Table 2 we see that, not surprisingly, the most frequent values across all benchmarks are 0 and 1 (this is true even for the benchmarks not presented). This suggests that special static optimization schemes to deal with these two values may be appropriate, since they are likely to benefit most, if not all, programs [3]. Consequently we will separate out 0's and 1's in most of our value locality measurements and in evaluations of techniques to exploit this value locality. As indicated in Section 3, zero values in our measurement exclude reads and writes to register R31.

bzip2	**crafty**	**gap**	**gcc**	**gzip**	**mcf**	**parser**	**perl**
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
4831843632	7	2	4	4	5368778784	115	32
5368712376	3	81	4831839248	32	5	114	123
62914560	5369648560	5	3	2	4831863760	97	46
65536	8	5369767936	52	3	10	105	101
5368712432	2	8	-1	5368758224	32	111	111
32	5369777344	3	59	16	2	4832016576	48
2	6	4	7	-1	49	99	4
3	5368862128	16	5	8	-1	4832011972	2
4	4	5386545160	2	-32768	24641	101	5368745560
101	128	32	262196	7	37	60	114
5368709136	9	-1	101	16382	64	112	49
71	15	5369767960	6	5	4	108	5
4831407792	5	4831845088	27	32767	108	110	105

Table 2: Common results for 8 SPEC2000 integer benchmarks

Other frequent values differ for the different benchmarks. Values such as 2, 3, and 4 are frequent in some benchmarks (a likely result of values for induction variable in short loops), but not in others. (Later in Table 3 see that 0's and 1's are *much more* frequent than other values.) The large values correspond to addresses of frequently-used variable and these are very likely different for different programs. To exploit value locality for such values, the optimization schemes would have to be dynamic since the values differ by program.

[2]Later in Table 3 we will see a slightly different order for some of the less frequent values when the entire program is considered.

[3]In particular, they benefit all the SPEC2000 integer benchmark programs.

4.3 Commonality of Values in a Dynamic Instruction Window

The results of Table 2 show the frequent values occurring in a dynamic execution of the program, but do not show locality. Figure 1 presents the percentage of instructions which create a value that is the same as a value created by one of the last N committed instructions, for N of 64, 128, 256. 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 by Jourdan *et. al*, for the IA-32 architecture [9]). We have separated out 0's and 1's, and will continue doing so for the remainder of this paper. For gcc, with an instruction window of 64, 54.6% of all instructions produce a value that is the same as a value produced by one of the previous 64 instructions. About 20% of the (matching) values produced are 0's and about 6% are 1's.

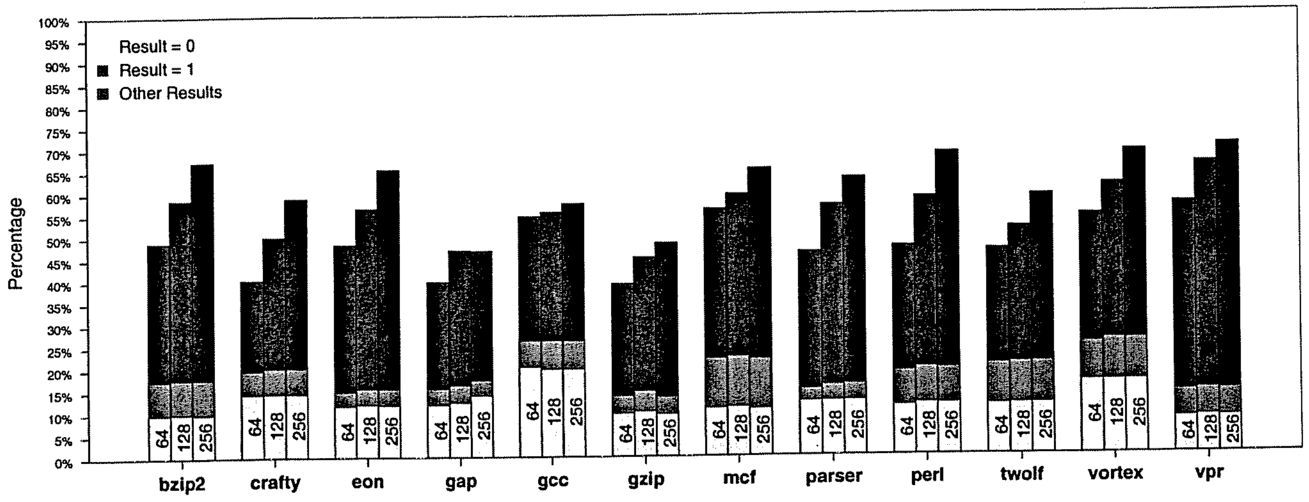

Figure 1: Percentage of results that can be found in a window of values produced by N (64, 128, 256) recently committed instructions

4.4 Commonality of Values Written to Physical Registers

The results of Figure 1 correspond only to instructions that are committed. Physical registers, however, are used to store values not only for instructions that are committed, but also for instructions that end up being discarded. A more accurate picture of the value locality that we might expect to exploit would consider values that have been placed in the physical registers during program execution. Figure 2 presents the percentage of instructions which create a value that is already present in the active set of physical registers (registers in the free list are not considered). The three bars for each benchmark correspond to physical register sizes of 80, 128, and 160 registers, respectively.

The heights of the bars in Figure 2 are slightly lower than those in Figure 1 (see, for example, the bars for N = 128 in Figure 1 and 128 physical registers in Figure 2). This suggests a slightly lower amount

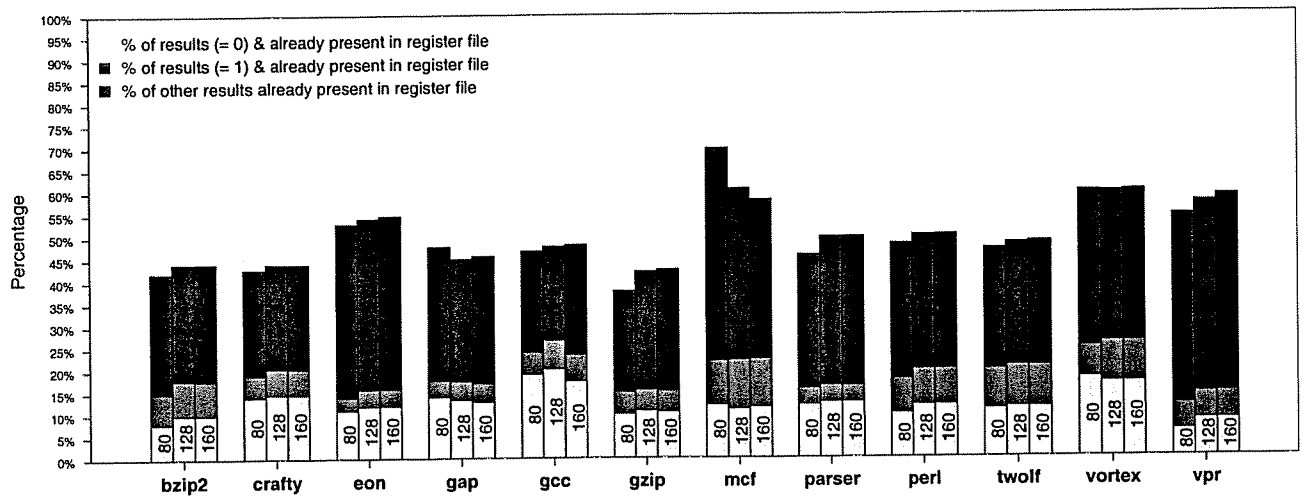

Figure 2: Percentage of results produced already present in register file

of exploitable duplication than the amount suggested in Figure 1.

4.5 Duplication of Values in Physical Registers

The data of Figure 2 suggests the presence of lot of duplication of values in the physical registers, with conventional register file management strategies. For example, in gcc, with 80 physical registers, 46.7% of the values written into the register file already exist in it; each write creates a duplicate value and takes up a physical register. With the alternative register mapping and file management strategies, a new physical register need not be assigned for a duplicate value. Thus, it would seem that the physical register file size could possibly be cut down by 46.7% in this case, without a performance loss. This statement, however, is not correct, since the data in Figure 2 is somewhat optimistic as we explain next.

The register file size requirements do not decrease proportionately to the number of duplicate values created. This is because some of the registers that are counted as holding duplicate values (as in Figure 2) might end up shortly being returned to the free list anyway, as part of a conventional physical register management discipline. For example, consider that P3 contains 2, and P19 is assigned to an later instruction, that also creates the value 2. The results of Figure 2 would consider P19 to hold a duplicate value, and thereby be counted towards the possible reduction in the number of physical registers. But suppose that shortly after 2 has been written to P19, the normal physical register discipline returns P3 to the free list. P19 is no longer a duplicate value. Returning P19 to the free list and holding on to P3 is not much better than 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 number of registers holding duplicate values.

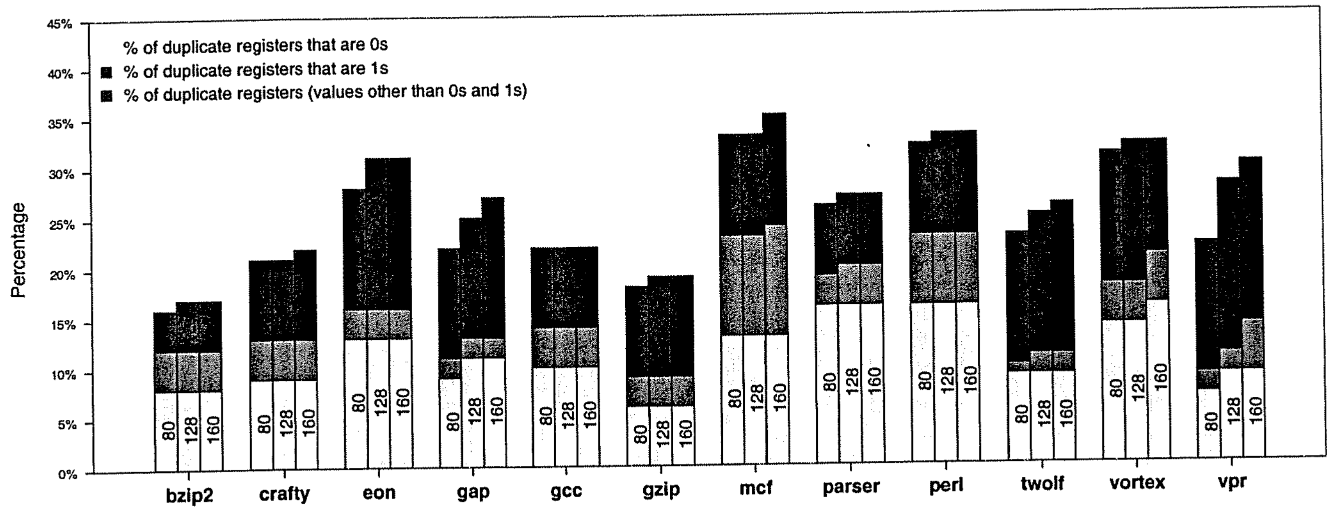

Figure 3: Percentage of duplicate registers in register file

Figure 3 presents the average percentage of registers that contain a duplicate value, i.e., the same value resides in multiple registers. We measure this data by counting the duplicate values in the register file when an instruction retires, and averaging the measurements.

The data in Figure 3 is somewhat less impressive than the data of Figures 1 and 2. However, it is a more realistic measure of the value locality that we can expect to 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 physical registers for `gcc`, a scheme that assigns a single physical register to only 0's, to 0's and 1's, and to all values, could reduce the physical register requirements by 10%, 14%, and 22%, respectively.

4.6 Statically-Determined Values Created and Read

We conclude our value locality measurements by considering one other important piece of information: the prevalence of the most common values (0 and 1), and some other values that we might consider for static optimization (2, 3, 4, 5, 16, 32), in the total number of values created and read as source operands.

Table 3 presents the occurrence of the statically-determined values as a percentage of all values created during program execution. Thus for `gcc`, 19.8%, 6.5% and 0.7% of all values created by value-creating instructions are 0, 1, and 2, respectively.

Table 4 presents the occurrence of the statically-determined values in the source operands of instructions. Thus for `parser`, 8.0%, 2.1%, and 0.5% of all source operands read are 0, 1, and 2, respectively. Reads of the register holding 0 (R31) are not counted in these measurements.

We see that 0's and 1's account for a significant fraction (18.9%) of all the values created. The per-

Values	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr	Avg.
0	9.77	14.28	11.67	11.14	19.84	9.28	11.17	12.51	11.62	11.54	16.23	8.10	12.26
1	8.06	5.09	3.69	4.19	6.59	3.54	11.54	3.68	8.11	9.41	9.90	6.12	6.66
2	1.47	1.29	0.56	2.39	0.77	0.72	0.29	0.59	1.87	2.03	0.24	0.83	1.09
3	0.94	1.07	0.44	0.61	0.38	0.62	0.25	0.35	0.79	1.10	0.12	0.43	0.59
4	2.41	1.17	1.26	0.30	0.74	0.75	0.22	0.54	1.52	0.96	1.79	0.67	1.03
5	0.46	1.02	0.27	0.59	0.12	0.50	0.19	0.50	0.27	0.81	0.99	0.36	0.51
16	0.07	0.54	0.15	0.80	0.70	0.52	0.14	0.51	0.37	3.23	0.10	0.22	0.61
32	1.24	0.39	0.16	0.46	0.26	2.88	0.15	1.26	0.48	0.25	0.05	0.18	0.65

Table 3: Percentage of common values created

Values	bzip2	crafty	eon	gap	gcc	gzip	mcf	parser	perl	twolf	vortex	vpr	Avg.
0	7.10	10.00	6.42	7.18	9.11	5.69	5.35	8.04	6.82	7.09	8.28	5.31	7.20
1	5.14	3.53	1.95	2.09	3.55	2.12	6.41	2.10	5.67	6.03	8.25	3.63	4.21
2	1.18	1.14	0.30	2.23	0.55	0.74	0.24	0.53	1.44	1.63	0.20	0.90	0.92
3	0.70	0.90	0.18	0.36	0.18	0.71	0.19	0.22	0.47	1.06	0.10	0.48	0.46
4	1.23	1.09	1.18	0.30	0.53	0.85	0.25	0.50	0.98	0.87	1.25	0.76	0.82
5	0.37	0.92	0.16	0.62	0.08	0.66	0.15	0.30	0.19	0.69	0.71	0.39	0.44
16	0.06	0.48	0.02	0.92	0.34	0.40	0.12	0.44	0.34	2.39	0.08	0.25	0.49
32	1.13	0.38	0.18	0.49	0.26	1.97	0.12	0.81	0.33	0.24	0.04	0.15	0.51

Table 4: Percentage of common values as source operands

centage of 0's and 1's in the values used as source operands (11.4%) is less than the percentage of 0's and 1's in the values created, but high enough to consider special-case optimizations. Other statically-determined values don't contribute significantly to either the values created, or the values used as source operands, suggesting that statically optimizing for these values is unlikely to be beneficial.

5 Reducing Physical Register File Size

We now consider the reduction in physical register file size by allocating a single physical register for a given value. The physical register allocation and register mapping process 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 is available, a check is performed to see if the value already resides in the register file. If it does, the destination logical register is re-mapped to the physical register that contains the value[4]. Actions are taken to ensure that dependent instructions get the correct value, and the previously-allocated physical register for the result is made a candidate for returning to the free pool. If no match is found, no special actions are taken. Several options are possible as to how this can be accomplished; we discuss some of them below.

[4]An update of the rename map is needed only if the logical register has not already been re-mapped by a later instruction in the program.

5.1 Microarchitectural Support for Physical Register Reuse

To accomplish the tasks of: (i) locating the presence of a value in the physical registers, and (ii) allowing multiple logical register to be mapped to a single physical register, we need additional microarchitectural support. We discuss this next.

5.1.1 Value Cache

The Value Cache is a CAM structure that contains mappings of values to physical registers, *i.e.*, identifies the values that reside in the physical registers. A match of a value returns the physical register which contains the value. If no match is found, an entry comprising a <value, physical register> pair is created in the Value Cache. The physical register returned by the Value Cache is used to update the register rename map, if needed. When a physical register is returned to the free pool, the corresponding entry in the Value Cache is invalidated.

5.1.2 Reference Counts for Physical Registers

To account for the many-to-one mapping of logical to physical registers, each physical register has a reference count associated with it. The reference count is set to one when the physical register is first assigned. It is incremented when another logical register is mapped to it, and decremented when an instruction whose destination register is mapped to it is committed. (More accurately, it is decremented when the conditions, under which the original physical register assigned to hold the result would be freed, are met.) A register can be returned to the free list only when its reference count is zero. Figure 6 illustrates an example physical register file before and after physical register reuse.

Tag	Value
p0	0
p1	1
p2	1
p3	0
p4	-1
p5	0
p6	32768
p7	0
p8	1
p9	2
p10	3
p11	2

(a) Before Register Reuse

Tag	Value	Ctr
p0	0	4
p1	1	3
p4	-1	1
p6	32768	1
p9	2	2
p10	3	1

(b) After Register Reuse

Figure 6: Physical Register File

5.1.3 Handling Instructions Dependent Upon Re-mapped Registers

We now discuss how to handle instructions that are dependent upon the instruction whose destination register has been re-mapped. There are two cases: (i) instructions that have not yet entered the instruction window and therefore their source registers have not yet been renamed, (ii) instructions that are waiting in the reservation stations and whose source registers have been renamed. The former category of instructions need no special treatment since they can simply access the desired name for the re-mapped physical register. Instructions in the latter category need special treatment. The schemes that we develop for this purpose would need to free the old physical register before it would be freed in the normal way, else benefits of a smaller register file would not be achieved.

Updating Source Physical Register Numbers. One option is to update the source physical register tags of the instructions waiting in reservation stations to point to the new physical register when a logical register is re-mapped. The disadvantages of this approach include additional latency (the latency of the Value Cache) in waking up dependent instructions and additional hardware (and wires) to broadcast the new physical register number. The advantage of this approach is that once source physical registers have been updated, no additional modifications are needed. A similar approach is used by Gonzalez *et. al* [8] to update the instruction queue entries with the allocated physical register.

Alias Table. The Alias Table is a structure which contains <old physical register, new physical register, valid> tuples, and is used to redirect references for <old physical register> to <new physical register>. When a logical register is re-mapped to point to the new physical register, the old physical register is freed. Instructions waiting in reservation stations, that are dependent upon the old physical register are informed to consult the Alias Table before accessing physical registers [5]. With this approach the execution of instructions that need to consult the Alias Table is delayed by the latency of the Alias Table. Entries in this table are created when a logical register is re-mapped to another physical register and freed when a matching <new physical register> is returned to the free list.

Partitioned Free List. The idea in this approach is to allow instructions to access the old physical register as long as it has not been allocated to hold another instruction's result. By doing so, the penalty associated with accessing the Alias Table is incurred only when the old physical has actually been reassigned.

This can be accomplished by partitioning the free list. One partition holds the free registers, freed the normal way. The second partition holds registers that were freed because of register re-mapping. When a physical register is needed, it is obtained from the first partition if it is not empty. The second

[5]Note the fact that the Alias Table needs to be consulted is known to the instruction scheduler. Consequently, its operation is not unnecessarily constrained, as it might be if this information was not available.

partition is accessed only when the first partition is empty. Doing so delays the reassignment of the physical register, allowing instructions using the previous mapping to continue to use it until then, thereby avoiding the penalty to access the Alias Table. Onder and Gupta [16] proposed a similar partitioned free list strategy to reuse loads and stores.

5.2 Optimizations for Statically-Determined Values

As we observed in Section 4, some values are quite common (*e.g.*, 0 and 1). These statically-determined values represent a disproportionate percentage of the values created, and of the values used as source operands. If we are interested in only optimizing for a small number of such values, the microarchitectural changes described in the previous Section are greatly simplified.

We start out by reserving physical registers for given values (*e.g.*, P0 for 0 and P1 for 1). With a given static assignment, the value produced as a result uniquely identifies the physical register where it would reside, and a dynamic tracking mechanism like the Value Cache is not needed.

The notion of reserving physical registers for specific values bears some resemblance to the notion of reserving logical registers for specific values, but is quite different. Many architectures reserve a logical register for the value 0; reads to that register return 0, and writes are treated as NOPs. Reserving a logical register reduces the number of general-purpose registers. The value 0 could, and does, occur frequently in other logical registers at different points in time during the execution of a program. Reserving a physical register for a value has no implications for logical registers, but allows accesses to multiple logical registers to be directed to a single physical register.

Since these special physical registers are not going to be used to hold any other values, they are not released to the free list, and therefore reference counts are not needed. Reserving registers for specific values has a potential drawback in that they cannot be used for other values. We make three points to address this concern. First, is is likely that "registers" will be reserved for only a very small number of values, for example, for two values: 0 and 1. Second, these values are frequent enough that they are almost always present in the physical registers. Third, the physical register names are simply a convenient way of naming these values: they need take no space in the physical register file, and can be provided directly as inputs to an operation's execution with suitable modifications to the datapath and control.

We now consider providing these statically-determined values to dependent instructions. Traditionally, when an instruction producing a value completes, dependent instructions waiting for the value are informed about the availability of the value (*i.e.*, dependent operations are woken up). The dependent instructions track the availability of the value (in the register file) via state bits. By maintaining a

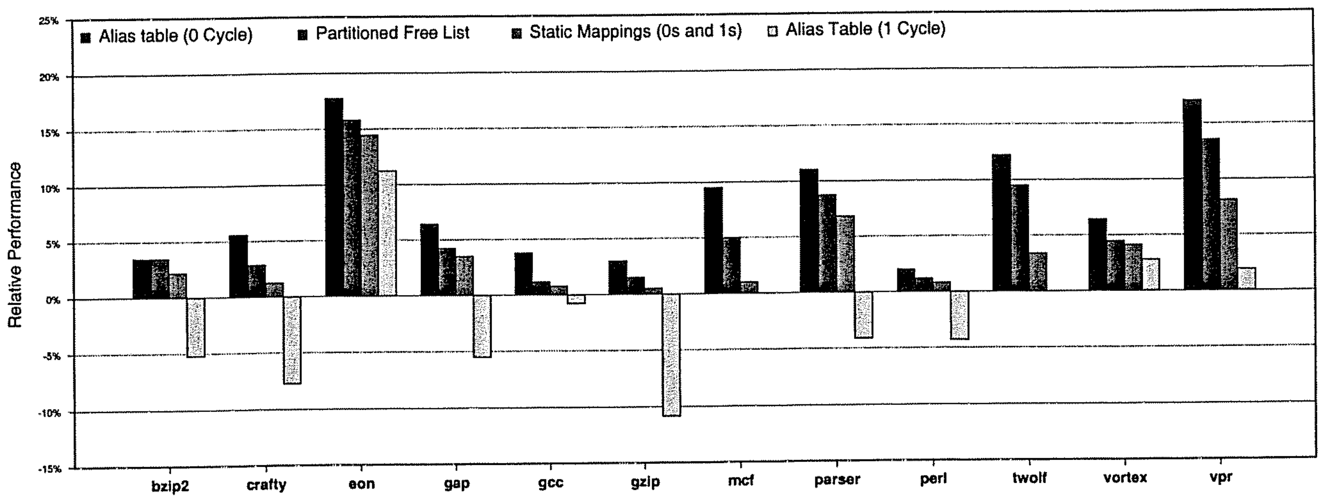

Figure 7: Performance relative to base case (80 physical registers with no register reuse) for different schemes

few extra state bits, the wakeup logic and the dependent instruction could track a small number of statically-determined values. For example, 2 bits could be used to represent a value of 0, 1, 2, or unknown (*i.e.*, read the value from the register file). Dependent instructions can therefore obtain these statically-determined values simply by looking at their state bits, without having to access the physical register file. If the physical register file does not have to be accessed, the Alias Table is not needed.

To summarize, if we are only interested in exploiting value locality for frequently-occurring, statically-determined values such as 0 and 1, the microarchitectural 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 the statically-determined values (if need be), (ii) add additional state bits to the reservation stations to track whether the operand is one of the statically-determined values, and (iii) suitably modify the control logic.

5.3 Performance Evaluation

We now evaluate the performance of the different techniques discussed above. It is possible to get the magnitude of the reduction in physical register file size. We do not do this for two reasons. First, it is somewhat tedious to determine since it involves multiple simulations, with different physical register sizes for different benchmarks. Second, a practical machine is unlikely to be designed with different register file sizes for different programs. Rather, it will have a fixed register file size, and the benefits of physical register reuse will be observed as additional performance. Thus, the data we present shows the performance benefits possible with the physical register reuse schemes.

Figure 7 presents the relative performance for four different cases: an Alias Table with no access penalty, a partitioned free list, statically-determined optimizations for only 0's and 1's, and an Alias Table with a 1-cycle access penalty. Performance is measured as the number of cycles to execute the program, and the base case is a machine with 80 physical registers. We see that an Alias Table with a 1-cycle penalty is the worst scheme in all cases, actually decreasing performance in some cases. An Alias Table with no access penalty (likely an unrealistic scenario) is the best option, significantly improving performance in many cases. The partioned free list is able to achieve performance close to that of the 0-cycle Alias Table in most cases. Optimizing only for 0's and 1's does quite well too, but not as well as the partioned free list case. However, it requires very little additional hardware.

Figure 8 changes the number of physical registers. Performance is presented for three cases, each with 60 physical registers, relative to a base case of 80 physical registers with a conventional management discipline. As the figure suggests, having only 60 physical registers with a conventional management discipline degrades performance significantly in many cases (*e.g.*, 25.4% in `crafty`). However, employing physical register reuse, with a partitioned free list, recovers most of the performance degradation, achieving a relative performance benefit in some cases. Thus, a physical register file with 60 registers, employing physical register reuse (with a partitioned free list) is performing roughly equivalent to a physical register file with 80 registers with conventional management.

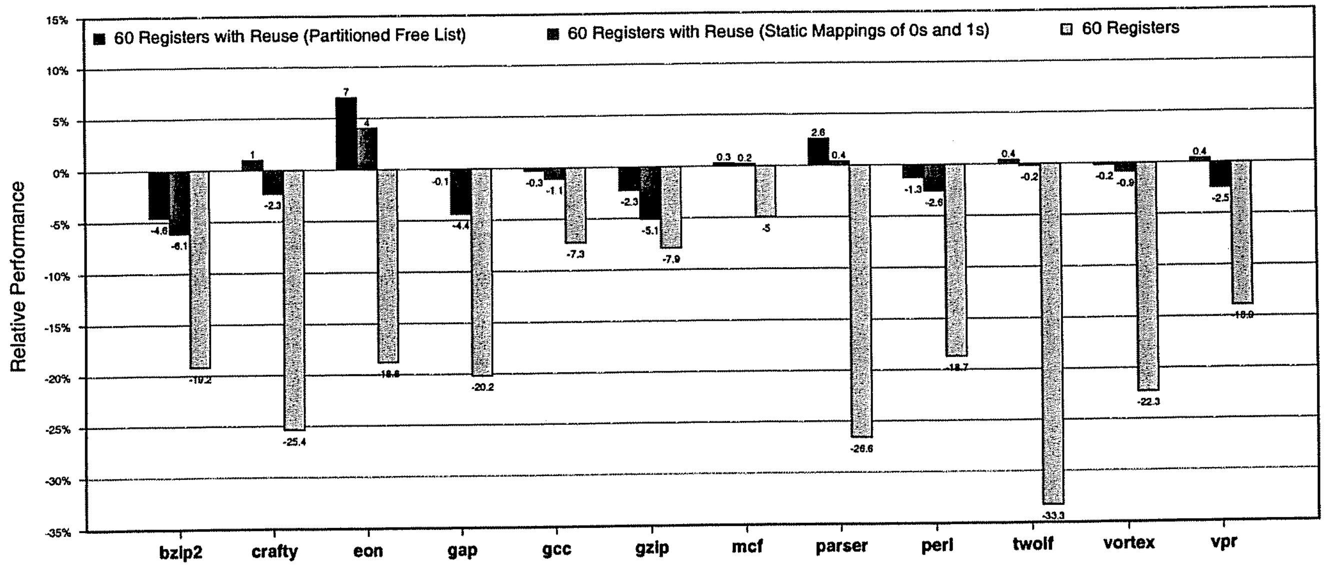

Figure 8: Performance with respect to IPC of base case (80 physical registers with no register reuse)

Only optimizing for 0's and 1's allows a physical register file with 60 registers to perform only slightly worse than an 80-entry conventional register file. This approximately 25% reduction in physical register file, for equivalent performance, is what would have been expected from the data of Figure 3, for the benchmarks that were limited by the number of physical registers (*e.g.*, `crafty`, `parser`, twolf). For benchmarks that are not limited by this amount of physical registers (*e.g.*, gcc, mcf), a proportionate reduction is not achieved.

6 Reducing Register File Accesses

We now consider how value locality could be used to reduce the number of register file accesses. We first consider writes, and then reads. While the reduction in register file access may or may not allow us to reduce the number of ports to the register file, it will nonetheless have a power benefit. We consider schemes to reduce access and quantify the reduction in access, rather than quantify the likely power or performance benefits of this reduction.

6.1 Reducing Register File Writes

As before, we maintain a Value Cache, and use it to determine if a value is already present in the physical registers. The register mapping in the rename map is updated, but the value need not be written into the assigned (new) physical register. Thus, the write of the register file is replaced with a write of the rename map (As noted earlier, the rename map is updated only if the logical register has not already been renamed). However, if we choose not to write into the assigned (new) physical register, instructions that have already been renamed to the new physical register will have to be redirected to another physical register (*e.g.*, via an Alias Table). As we saw in Section 5, the additional latency of this indirection can negate the benefits of reduced number of physical registers. Any savings in writes are likely to be of questionable benefit in this case. The partitioned free list approach does not save register write traffic since the assigned physical register needs to be written.

The data in Section 4 also suggested that by limiting our attention to statically-determined values (0's and 1's), we could achieve much of the benefits of register file size reduction. By concentrating on statically-determined values, specifically 0's and 1's, we can trivially reduce the number of register file writes as described in Section 5. Writes of statically-determined values update the rename map, if need be, and the additional state bits in the reservation stations, but not the actual physical register file.

6.2 Reducing Register File Reads

Similar to the situation for writes, no benefit is apparent for dynamically-determined duplicate values. These values have to be read from the register file, or some other storage similar to the register file.

For statically-determined values, however, the optimizations we described in Section 5 reduce the read access to the physical register file. Recall that we added state bits to the reservation stations that were updated when the availability of a result was announced. When the value can be determined from these state bits, a register file read is not required.

Quantifying the benefit on read access reduction is complicated by the fact that not all source operands are obtained from the physical register file. Many source operands are read from bypass paths; the register file need not be read in this case anyway.

6.3 Evaluation

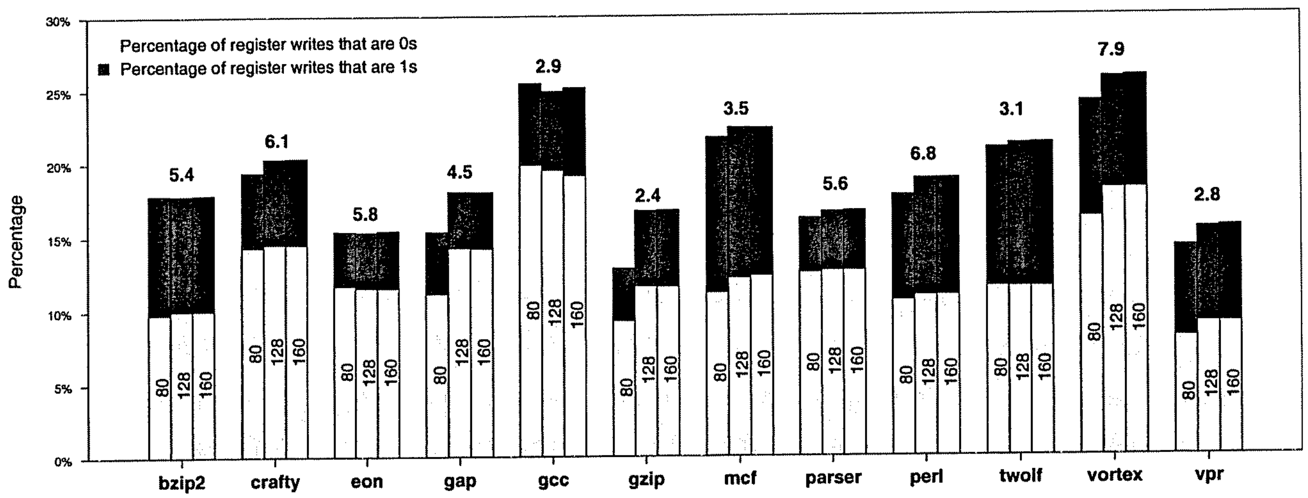

Figure 9: Percentage of register writes (of 0's and 1's) that can be cut down with static mappings

Figure 9 presents the percentage of register file write operations that can be eliminated. For each benchmark, there are 3 bars, for register file sizes of 80, 128, and 160 registers, respectively. Each bar shows the number of writes of 0's and 1's. The number on top of the bars is the percentage increase in writes to the rename map needed due to reassignment of a logical register, for the case of 128 physical registers. We see that with 128 physical registers, an average of 18.9% of the writes to the register file can be eliminated, with an average increase of 4.7% in the number of writes to the rename map. For `gcc`, about 26% of the writes to the register file could be suppressed with an increase of about 2.9% in the writes to rename map[6].

Figure 10 presents the percentage of read operations that could be eliminated. The data in the figure consider all source operand values, including values obtained from the bypass paths, as having been read from the register file. Again there are 3 bars for each benchmark, and each bar separates the contribution of 0's and 1's. The reduction is not as impressive as the reduction in the number of writes, but nonetheless a technique that might be worth pursuing because of potential power advantages.

Figure 11 presents data similar to that of Figure 10, but excludes values that are obtained from the

[6]Note to reviewers: finally a microarchitectural technique that does better for `gcc` than other benchmarks :-)

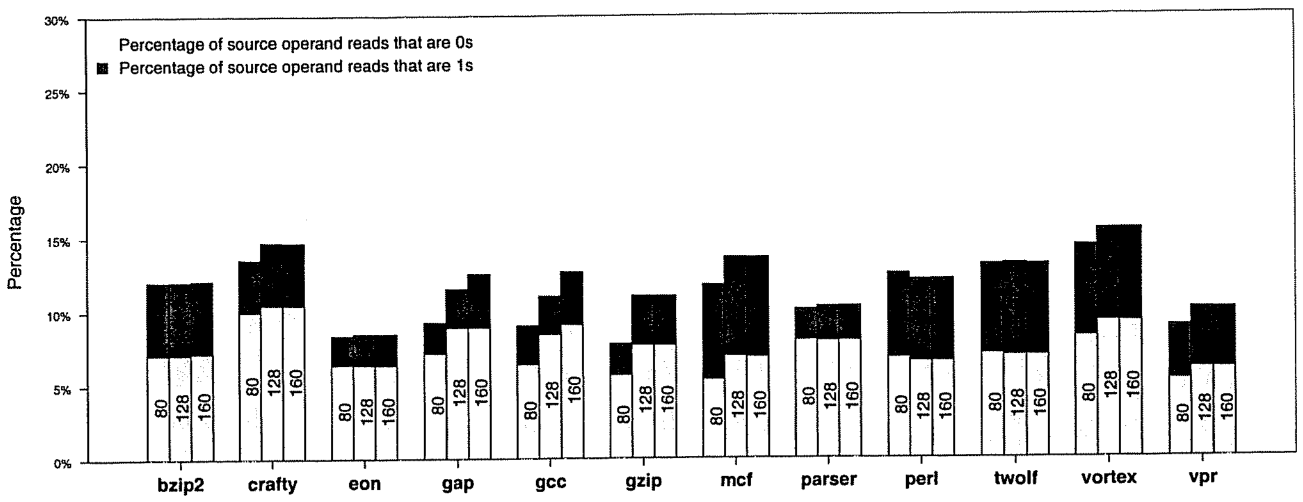

Figure 10: Percentage of source operand reads (from register file and bypass path) that are 0's and 1's

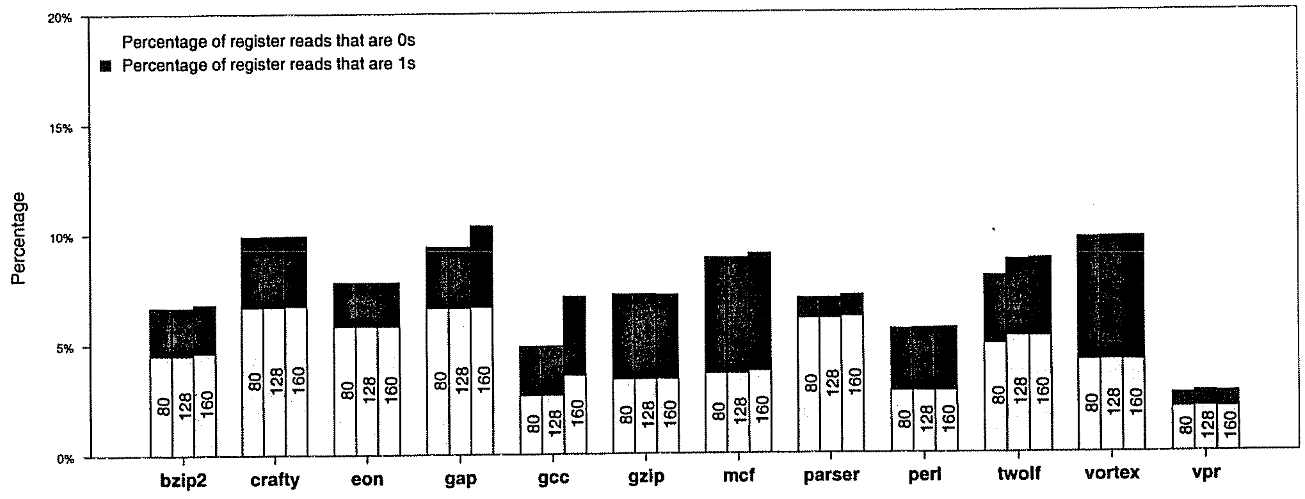

Figure 11: Percentage of register reads (of 0's and 1's) that can be cut down with static mappings

bypass paths [7]. Comparing the data in Figures 10 and 11 suggests that many of the values being bypassed are 0's and 1's, and thus a smaller percentage of the remaining reads (an average of 7.7% for 128 physical registers) can be eliminated.

7 Related Work

Register files have been the subject of study for a long time and several approaches to optimize register file 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

[7]On average, about 37.25% of the values are obtained from bypass paths, and 62.75% from the register file in the benchmarks. `parser` obtains the most values from bypass paths (51%) and `eon` the fewest (21%)

have a small, fast, first level, backed up by a larger, slower second level. This hierarchy can either be determined statically or dynamically.

The Cray architectures used 2 levels of architectural registers [20]. Static register file hierarchies were also explored by Swensen and Patt [25]. Dynamic register file hierarchies have been investigated by several recent papers [2, 5, 18, 30, 31].

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 [19, 27]. Different register files in a cluster could service different request 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 [4] have used statically-clustered register files, with each cluster being a separate register name space [13]. 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 [24]. An alternate form of dynamic clustering is used in complexity-effective architectures [17] and in clustered architectures such as the Alpha 21264 [10]. For example, Alpha 21264 has two clustered register files, with one cycle delay to bypass instructions between clusters.

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 on virtual-physical registers by Gonzalez *et. al* [8, 15]. Delaying the allocation of a physical register reduces the lifetime of a physical register and thereby the physical register requirements. Jourdan *et. al*'s [9] work enhances the concept of virtual-physical registers to exploit value locality. It introduces the concepts 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 Jourdan *et. al*. Our work differs from that of Jourdan *et. al*, in several ways. We present different, and more realistic, measurements of exploitable value locality, the proposed schemes (*e.g.*, the partitioned free list) are different, we emphasize optimizations for the statically-determined case of special values (0 and 1), and we consider optimizing register file access in addition to register file size.

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 frequent values generated across all the benchmark programs we studied, and these account for an average of 11.4% of values read, and 18.9% of values written by instructions.

We investigated using value locality to reduce the size of the physical register file, by using only a single physical register to hold a value. We studied microarchitectural support for compacting the physical register file, and evaluated several options. The proposed technique allows equivalent performance with a physical register file that is typically 25% smaller.

We presented further optimizations for 0's and 1's. In this case the microarchitectural support for reducing the effective size of the register file is simplified greatly. The resulting effective reduction in register file size is still significant. The cost-effectiveness of this technique makes it an attractive candidate for dynamically-scheduled processors employing a physical register file.

Optimizing for the special cases of 0 and 1 also allowed us to reduce the number of write and read accesses to the physical register file. An average of 18.9% of writes to the register file could be eliminated, with an average increase of 4.7% in the number of writes of the register rename map. Furthermore, an average of 11.4% of all source operands are either 0 or 1, and this translates into a 7.7% reduction in the (non-bypassed) values read from the physical register file.

References

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, 2000.

[2] Rajeev Balasubramonian, Sandhya Dwarkadas, and David Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-34)*, 2001.

[3] Douglas C. Burger and Todd M. Austin. The Simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 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 *IEEE Transactions on Computers*, volume C-37, pages 967–979. 1988.

[5] Jose-Lorenzo Cruz, Antonio Gonzalez, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[6] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 149–159, 1997.

[7] K. I. Farkas, N. P. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture (HPCA '96)*, 1996.

[8] Antonio González, José González, and Mateo Valero. Virtual-Physical Registers. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, 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 *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, 1998.

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

[11] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996.

[12] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 138–147, 1996.

[13] J. Llosa, M. Valero, J. A. B. Fortes, and E. Ayguade. Using sacks to organize registers in VLIW machines. *Lecture Notes in Computer Science*, 854:628, 1994.

[14] Doug Matzke. Will physical scalability sabotage performance gains? *Computer*, 30(9):37–39, September 1997.

[15] Teresa Monreal, Antonio González, Mateo Valero, José González, and Victor Vinals. Delaying physical register allocation through Virtual-Physical Registers. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, 1999.

[16] S. Onder and R. Gupta. Load and store reuse using register file contents. In *Proceedings of the 15th ACM International Conference on Supercomputing (ICS-01)*, 2001.

[17] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In 24th *Annual International Symposium on Computer Architecture*, 1997.

[18] M. Postiff, D. Greene, S. Raasch, and T. Mudge. Integrating superscalar processor components to implement register caching. In *Proceedings of the 15th ACM International Conference on Supercomputing (ICS-01)*, 2001.

[19] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens. Register organization for media processing. In *Sixth International Symposium on High-Performance Computer Architecture (HPCA '00)*, 1999.

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

[21] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 248–258, 1997.

[22] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In 24^{th} *Annual International Symposium on Computer Architecture*, 1997.

[23] Avinash Sodani and Gurindar S. Sohi. An empirical analysis of instruction repetition. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 35–45, 1998.

[24] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd Annual International Symposium on Computer Architecture (22nd ISCA'95)*, 1995.

[25] John A. Swensen and Yale N. Patt. Hierarchical registers for scientific computers. In *International Conference on Supercomputing (ICS)*, 1988.

[26] Luis Villa, Michael Zhang, and Krste Asanović. Dynamic zero compression for cache energy reduction. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, 2000.

[27] Steven Wallace and Nader Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, 1996.

[28] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 281–291, 1997.

[29] Emmett Witchel, Sam Larsen, C Scott Ananian, and Krste Asanovic. Direct addressed caches for reduced power consumption. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-34)*, 2001.

[30] R. Yung and N. C. Wilhelm. Caching processor general registers. In *International Conference on Computer Design*, 1995.

[31] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. Two-level hierarchical register file organization for VLIW processors. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, 2000.

[32] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent value locality and value-centric data cache design. *ACM SIGPLAN Notices*, 35(11):150–159, 2000.