A Content Aware Integer Register File Organization

Rubén González, Adrián Cristal, Daniel Ortega[†], Alexander Veidenbaum[‡] and Mateo Valero
Universitat Politècnica de Catalunya – [†]HP Labs Barcelona – [‡] University of California, Irvine
{gonzalez,adrian,mateo}@ac.upc.es – daniel.ortega@hp.com – alexv@ics.eci.edu

Abstract

A register file is a critical component of a modern superscalar processor. It has a large number of entries and read/write ports in order to enable high levels of instruction parallelism. As a result, the register file's area, access time, and energy consumption increase dramatically, significantly affecting the overall superscalar processor's performance and energy consumption. This is especially true in 64-bit processors.

This paper presents a new integer register file organization, which reduces energy consumption, area, and access time of the register file with a minimal effect on overall IPC. This is accomplished by exploiting a new concept, partial value locality, which is defined as occurrence of multiple live value instances identical in a subset of their bits. A possible implementation of the new register file is described and shown to obtain very good results, even when compared to recently proposed optimized register file designs. Overall, an energy reduction of over 50%, a 18% decrease in area, and a 15% reduction in the access time are achieved in the new register file. The energy and area savings are achieved with a 1.7% reduction in IPC for integer applications and a negligible 0.3% in numerical applications, assuming the same clock frequency. A performance increase of up to 13% is possible if the clock frequency can be increased due to a reduction in the register file access time. This approach enables other, very promising optimizations, three of which are outlined in the paper.

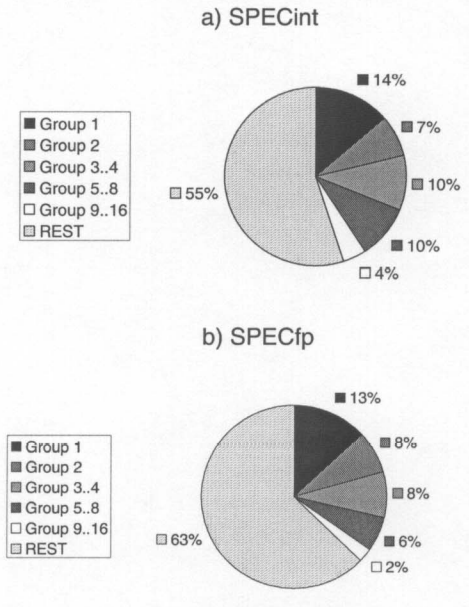
1. Introduction

Integer word size in recent high-performance processors has increased to 64 bits. This increase is primarily due to the need for a larger virtual address, but also helps to represent larger integer values in general. Computer architects have repeatedly shown that within this enormous dynamic range of values the distribution of used values is very non-uniform. A few values account for the majority of values used.

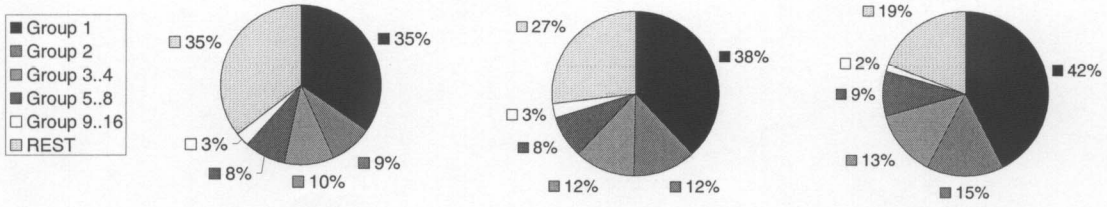
This bias in data values is taken advantage of by several micro-architectural techniques. Value Prediction [14] is one such technique, which takes advantage of identical values that are seen repeatedly in computations or memory accesses. Another example is the cache hierarchy, which works because of address locality – the fact that memory addresses get re-utilized [3]. Moreover, it has been shown [25] that *caching* a small number of frequently used values may help reduce memory utilization by approximately 50% in six SPEC95 programs.

Some of the above techniques exploit repeated occurrence of the same data values as well as of the *nearby* data values. This is the difference between temporal and spatial locality in the memory hierarchy. It is also the difference between two main types of data exploited by value locality.

This research investigates the exploitation of a different type of value locality. This form of value



Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)
1063-6897/04 $ 20.00 © 2004 IEEE



a) 8-similar b) 12-similar c) 16-similar

Figure 2. Distribution of *(64-d)*-similar Live Integer Data Values as a function of d

locality can be described as a *partial value locality*: a frequent occurrence of values that have identical subsets of consecutive bits. Results presented in this paper prove its existence and scope. The paper then shows how Partial Value Locality can be exploited in the design of a integer register file.

The integer register file is a CPU unit with one of the highest energy requirements [7] states that a register file accounts for 15% to 20% of the overall energy consumption. This is partly due to the 64 bit word size, as mentioned above. Another reason is its size of 64 or more physical registers. Finally, due to wide instruction issue, the file needs to have a large number of read and write ports. This research shows how partial value locality can be used in the design of a (physical) register file to significantly lower its energy requirements and, potentially, reduce its access time.

Recent work on low-energy register file design concentrated on reducing the number of ports. In contrast, the research proposed here reduces, on average, the effective register file size and its word size. This results in significant savings, even when compared to architectures with a reduced register file size and a limited number of ports, such as those proposed in [10] [5].

One can exploit value locality in the register file if the number of unique register values is small. Figure 1 shows the distribution of integer data values in the entire SPEC2000 benchmark suite[1]. Different sections of the pie chart show the fraction of distinct, live integer register values. For instance, it shows that a single value accounts for 14% of all live register values across all SPEC2000int programs. This fact has been used in [19] to design a better register file.

This paper shows that partial value locality is also widespread among register values. Its main contribution is a content aware register file organization, which takes advantage of the *nearness* of live register values. The register file is partitioned into several sub-files, each of them storing a different type of value. Such sub-files can be smaller, more narrow, and have fewer ports than the standard register file. In fact, it will be shown that the clock cycle time may be also be reduced using this approach resulting in performance improvement. The register file area is also improved.

The paper presents one application of *partial value locality* and of the new register file organisation. However, both of these are believed to have significant potential for other types of processor optimization. Section 6 briefly outlines three other applications and discusses their potential.

This paper is organized as follows. Section 2 defines a new concept, *d-similarity*, which is a special case of *partial value locality*, Section III defines a content aware register organization, and Sections 4 and V analyse the impact of the proposed mechanism on performance and energy consumption. Potential benefits of a shorter access time and area savings that the new mechanism produces are also analysed. Finally, future work is discussed in Section 6 and conclusions in Section 8.

2. Data Value Similarity

Let us define *similarity*, which is a special case of *partial value locality*. Two data values will be called *similar* if they are micro-architecturally *near* one another. *Micro-architecturally* reflects the fact that value similarity is expressed in micro-architectural rather than arithmetic terms. For example, two values that differ by 1 can be different in many bit positions.

More precisely, two 64 bit values are called *(64-d)*-similar if they differ in *d* least significant bits and are equal in the remaining *(64-d)* high-order bits. *(64-d)*-similarity captures *partial value locality* in computation that a micro-architecture can exploit. The question is whether the amount of this type of *partial value locality* is high. *(64-d)*-similarity will be evaluated for

[1] Obtained using an *oracle* that each cycle grouped and counted all live values in integer registers (for a processor configuration shown in Table I). The *oracle* then counted the number of live registers with the most frequent value, the second most frequent value, etc.

integer values in registers for both integer and floating-point codes in this paper, but it exists in other parts of the system as well.

(64-d)-similarity was measured as follows. Each cycle all live values in the integer physical register file were analyzed to determine how many had identical (64-d) high-order bits. Registers with (64-d)-similar values were placed in a similarity group and the number of registers in each group was determined. The groups were ordered based on the number of registers in them. Figure 2 a) shows that, on average, 35% of the live registers were in the first similarity group, another 9% in the 2nd group, 10% in the 3rd and 4th groups, etc. The remaining values (another 35%) were placed in the last group (REST). Without partial value locality the distribution would have been uniform.

Figure 2 a), b), and c) differ in the value of *d* used. An increase in *d* changes the distribution. For instance, the *1st-group* captures more partially redundant integer register values than any other group. In general, the group with a very large number of values (REST) decreases with increase in d and all others increase because of that. These values are cumulative in the following sense: tracking the top four *(64-d)*-similar values will capture all values in groups of *1, 2*, and *4* groups for a total of 70% of all values (for d=16).

The results in Figure 2 demonstrate that many values are *(64-d)*-similar. A group of *(64-d)*-similar values shares the high-order bits and each value instance in the group is uniquely represented by its remaining *d* bits. They are therefore called *short* values. Value instances that do not share their higher-order bits with any other value are called *long*. Finally, a subset of short values will be treated in a special way. These values have a simple high-order part, either all zeros or all ones, and are called *simple* values. The existence and frequency of occurrence of the three value types: short, simple, and long, can be exploited by reorganizing the register file as described in the next section. To simplify the notation, the *(64-d)-similar* values will be just called *d-similar* in the remainder of this paper.

3. A new register file organization

The new register file organization shown in Figure 3 in replaces a standard, N-entry physical register file organization. It consists of three separate files which together approximate the behavior and performance of N 64 bit physical registers. These files store values of different type: *short, long*, or *simple*. Each file has a unique size and bit width exploiting the properties of the value type it stores. The effect of these architectural parameters is studied later in this paper.

The Long register file stores 64-bit data and has the same general organization as a traditional register file. The difference is that it can be significantly smaller due to the existence of other value types, which are stored in the other two register files. The reduced size is one reason for its lower energy consumption. Reduced frequency of access is another reason.

The Short register file holds the high-order bits for short values. It is (*64-d-n*) bits wide (n is defined below). The size of the Short register file can be small, as will be shown below. The size reduction leads to lower access energy. The remainder of the 64 bit word, *d+n* bits is stored in the third register file described next.

The third register file is a *Simple* register file and it performs two distinct functions. The first one is to hold all unique *low-order d+n*-bit fields for *simple*, *short* and long values. The second function is to determine the type of register access being performed and to locate the *short* and *long* values. The Simple register file has N entries, which is equal to the size of standard physical register file if it were used. A simple entry is assigned in renaming, but its width is significantly smaller, reducing the access energy.

An entry in the Simple register file is composed of a 2-bit field called a Register Descriptor (RD) and a (*d+n*)-bit field called a Value field. The RD field identifies the type of value stored in the physical register (simple, short and long). The Value field stores a *d+n* bit, signed value for a simple value type or a *d+n*-bit value for a short value type. A pointer to the Short register entry uses the *n* bits. The Value field stores an *m*-bit pointer to the long register file and *d+n-m* bits of a long value. The latter allows the word size of the Long register file to be reduced further. The new register file is shown in Figure 3..

To summarize, an N by 64-bit physical register file is replaced in the proposed organization by three separate register files of the following sizes:

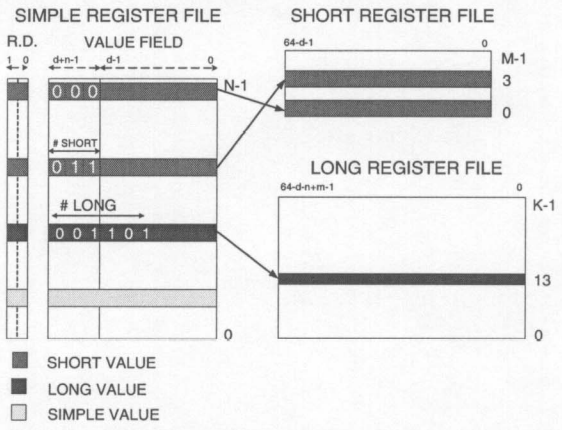


Figure 3. The Content Aware Register File Organization

Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)
1063-6897/04 $ 20.00 © 2004 IEEE

- an N-entry by (d+2+n) bit Simple file,
- an M-entry by (64-d-n) bit Short file,
- a K-entry by (64-d-n+m) bit Long file

where $n=\log_2 M$ and $m=\log_2 K$

Clearly, both the performance and the energy consumption of the register file heavily depend on parameters M, K, and d. Results presented below show the performance impact of these parameters and are used to select their values to optimize the IPC and energy consumption of the proposed architecture.

Finally, the content aware register file organization uses multiple read and write ports. The number of ports and existing techniques to reduce this number are largely orthogonal to the design of the content aware file. This will be further discussed in Section 4.

3.1. Register file operation

The operation of the new register file consists of two main steps. Let us first describe it for a read access. When a physical register *I* is accessed, the first step is to determine the type of value it stores. Based on this information, a type-specific register file (or files) is accessed and the value retrieved. The value type for the physical register *I* is determined by accessing the RD field of the *I*-th entry in the Simple register file. The value field is read out in parallel.

If the RD field specifies a *long* value, then m bits of the *d+n* Value field are used as an index to the Long register file. The *long* register file is accessed next, the desired value is read out, combined with the bits from the Simple file and sent to the functional unit(s). The long value access thus becomes a two-cycle operation.

If the RD field specifies a *short* value, then the *d+n*-bit and the *n*-bit sub-fields of the Value field are used as follows. The *n*-bit sub-field is used as an index to the *Short* register file, which is accessed next to read out the (64-*d*-*n*) high-order bits of the value. The d+n bits from the Value field and the (64-d-n) bits from the Short register file are concatenated to form the 64-bit value. This type of access also requires two cycles.

Finally, if the RD field specifies a simple value then the (*d+n*) bit Value field from the *Simple* register file is sign extended to 64 bits and sent to the functional unit(s). Using *d+n* bits for the *simple* values says that they belong to a *d+n*-similar class of value instances. For the *short* and *long* classification only the *d*-similarity is used. This simply takes advantage of the *n* extra bits to enlarge the range of potential *simple* values.

One final issue for reads is what to use as the n-bit pointer for short values. The low-order n bits of the (64-*d*) high-order bits of a value group are used for this purpose. This mechanism works well in that its use does not result in pointer conflicts. It is also shown below that this register file does not need to be very large and that a small n (*3* or *4* bits) suffices to index it. The Short register file entries are actually (64-*d*-*n*) bits wide since the last n bits are already stored in the Simple register file.

Writes present a different set of issues to resolve. First, a destination register of an instruction is allocated long before the value type of the result is known. Second, a value type determination needs to be performed before the write can be performed. The next section explains how this is done in the context of the instruction pipeline and its stages.

3.2. A Modified Instruction Pipeline

This section describes a possible implementation in the instruction pipeline. The previous section indicated that read access to the content aware register file requires additional steps, and therefore extra time. The extra time requires an additional register read stage in the instruction pipeline. As will be shown later in the paper, the extra stage has a negligible effect on performance. Modification of other pipeline stages is also required, as explained below.

Figure 4 shows the modified instruction pipeline starting with the ISSUE stage. Earlier stages, such as FETCH and DECODE, are not shown since they remain in the same place even if their functionality is modified. The main additions are an extra operand read stage, a value type determination (comparison) stage after execute or memory access, and extra bypass logic. These modifications are described next in pipeline stage order.

Decode/Rename: this stage now assigns a *simple* register entry to every destination logical register. The number of *simple* registers is equal to the total number of physical tags, so this is done in exactly the same way as in the baseline micro-architecture. At this point in the pipeline it is impossible to determine the value type of a destination physical register. This determination (and allocation for Short and Long values) is deferred until a later pipeline stage when the result value is available.

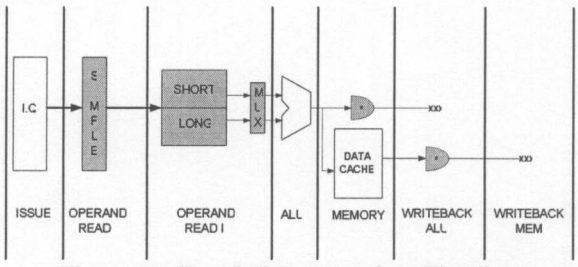


Figure 4. (Partial) Instruction Pipeline

RF Read: the register file access is performed in two separate stages, RF1 and RF2.

The first stage RF1 performs an access to the Simple register file and is equivalent to the baseline register file read stage. The access to the Simple register file produces the value type information (RD field) as well as the content of the Value field.

The second stage RF2 performs the following functions depending on the RD field value read in RF1:

- Set up the final result multiplexor based on the value type
- If the RD value obtained in RF1 was
  - Simple Value :
    The (d+n) bit value read out in RF1 is sign-extended to 64 bits.
  - Short Value
    - An n-bit pointer from the Value field accessed in RF1 is used to read an entry from the Short register file.
    - The value read from the Short file is concatenated with d+n bits obtained in RF1.
  - Long Value
    - An m-bit pointer from the Value field accessed in RF1 is used to read an entry from the Long Register File.
    - The value read is concatenated with (d+n-m) bits obtained in RF1.

One may wonder if the addition of a multiplexor delay in RF2 increases the cycle time. This is impossible to answer without a detailed design. However, this additional delay is likely masked by a shortened access time to the Short and Long register files. In addition, access to the Simple register file in RF1 is not likely to need a full cycle and time can be borrowed from there. The access time is lower because these files have fewer entries and lower bit width.

Write-Back: the write-back in the new register file organization also requires two stages, WR1 and WR2.

Recall that a physical register is assigned to an instruction by the renaming mechanism before the result value type is known. This is one of the reasons why all register accesses go through the *simple* register file first (although this may be avoided, as will be shown in Section 6). The assignment of a Short or Long register file entry, if required, has to be delayed until the data value is produced and its type determined. It then requires a Short or Long register allocation at this point in the pipeline (see description below).

The first stage WR1 performs the following two functions concurrently:

- Determine if the result is a simple value. This is a (64-d-n) bit compare of the higher bits with 0 and 1.
- Use the appropriate n bits of the result as a pointer to determine if it is a known short value. This is accomplished by reading a Short register pointed at and performing a comparison. The value is Short if the comparison succeeds, otherwise the result is marked as a Long value.

This is the longer of the two functions. Its timing is determined by the Short register file access time. This file is very small and thus has a much lower access time then the other two files. Therefore, this should not be a problem.

It also requires additional read ports on the Short file, one for each write port. Again, this file is very small and extra ports on it are not a serious problem.

At this point the type of an integer value is determined and the actual write can be performed in the next stage.

The second stage WR2 performs the following two functions concurrently

- Write the value type in the Register Descriptor field of the corresponding Simple register file entry.
- Perform one of the following depending of value type determined in WR1:
  - Simple value
    - write the low (d+n) bits in the Value field of the Simple Register File. Includes the sign bit.
  - Short value: perform the following two operations concurrently
    - write the low (d+n) bits in the Value field of the Simple Register File.
    - write the high (64-d-n) bits in entry **n** of the Short Register File.
  - Long value: perform the following three operations concurrently.
    - write the number of the newly allocated register in the Value field of the Simple register file.
    - write the value to the allocated Long register.
    - update the register allocation state (see below)

Result register allocation for Short and Long values is performed during the write-back process. The allocation process needs to be fast. A different strategy is used for each value type.

Short: a short register is allocated using the n-bit pointer contained in the value. If the desired entry in the Short register file is not free the entry is declared Long and the Long register allocation process is used. Bits d, d+1, …, d+n-1 of the result are used as an n-bit pointer.

Long: maintain a pointer(s) to the next free register to use and a free-entry counter. This pointer and the counter are updated each time an allocation is made during the WR2 stage.

If there are no free long registers a *Recovery State* is entered. A *pseudo-deadlock* is possible because an instruction is issued before it is known that it will need a *Long* register. For simplicity, the solution chosen in this work is to stall the pipeline and free long registers at this point. A description of how short and long registers are normally freed can be found below.

The *pseudo-deadlock* situation was observed to happen very infrequently. The deadlock probability is further reduced by initiating the recovery action when the number of free registers is low (but not 0). It was experimentally determined that stalling issue when he number of *long* physical registers is equal to issue width greatly reduces the chance of *pseudo-deadlock* with a negligible impact on performance.

The allocation of *Short* register entries above did not describe when and how to select a value with a lot of partial reuse. It is possible to try to allocate a new short entry on every result and then see how useful it is. This proved to cause a lot of thrashing because the Short file is small and does not capture all *short* values with a lot of reuse. Our analysis showed that good *short* values mainly come from address computations. Therefore the allocation of new *short* values was restricted to this type of instructions. A base register used in a LD or ST instruction is written into a Short file if a location it maps to is free. This is actually performed in parallel with the ALU stage and is the only time the Short File is written. An address computed by a Load or a Store instruction is used to allocate an entry, but only if the indexed Short Register File location is free.

The Long and Simple registers are freed upon instruction commit. Determining when to free a Short register entry has to be done in such a way that frequent value information is kept across different registers. Reference counters can be used but their management is complicated, in particular on branch misprediction. Instead, a mechanism reminiscent of the reference bits in the Virtual Memory system is used. These reference bits are periodically cleared and then usage is recorded again.

When an instruction reaches WR1 stage and its value type is Short, a bit *Tcur* is set to indicate that this Short register still has a frequent value. There is one such bit per Short register. When the entire ROB is consumed (a period called *ROB interval*), *Tcur* is *OR'd* with *Tarch*, another flag, to save the information from all retired instructions. *Tarch* tracks whether a value in a given Short register is used by an architectural register. The OR of *Tcur* and *Tarch* shows that this Short value was used during the last *ROB interval* and is copied to Told. The *Tcur* is now cleared and *Tacrh* is re-calculated for the current content of the Short file. The recalculation is performed by a simple background mechanism. A short register is free when Told, Tcur, and Tarch are all zero.

Table 1. Architectural parameters

Baseline Out-of-Order Microprocessor	
Simulation strategy	Execution-driven
Issue/Fetch/Commit width	8 instructions/cycle
Branch predictor	Gshare ,14 bit history
I-L1 size	32 KB, 4-way, 1 cycle
D-L1 size	32 KB, 4-way, 2 rd/wr ports, 1 cycle latency
L2 size	1MB, 4-way 10 cycle latency
Memory latency	100 cycles
Memory bus width	32 bytes
Physical registers	112 Integer/128 FP
Reorder Buffer	128
Load/Store Queue	64
Integer Queue	32
FP Queue	32
Integer Funcional Units (lat.)	8 (latency 1)
FP Funcional Units (lat.)	8 (latency 2)

Result Bypassing. The extra stage WB2 requires an addition of an extra level of bypassing to avoid stalls. It is estimated that the extra delay and energy consumption of the bypass logic is compensated by the savings in the register file. However, this additional bypass does not have to be implemented if too expensive. It is not used very frequently and provides little performance improvement.

4. Performance

This section presents the performance and shows the potential of the content aware register file with respect to a baseline architecture defined in Table 1. SPEC2000 benchmark suite was used to evaluate the performance of this approach. The results presented are averages over both integer and floating point applications. All benchmarks have been simulated for 300 million *representative* instructions, where representative is defined following [23].

Micro-architectures using a physical register file with unlimited resources do so to maximize performance and to eliminate the need to stall the processor due to lack of physical registers. The baseline architecture performance is compared to such an architecture. It uses 2x8 read ports and 8 write ports for this design. The register file size is equal to the

ROB size plus 32 (the number of architectural registers) or 160 registers for this design.

The baseline architecture is defined to have a register file with a reduced number of registers and ports. It has been shown [15] that such a reduction results in a negligible performance degradation and our evaluation confirms this. The baseline register file contains *112* physical registers (instead of *160 registers)*. This size was observed to lead to a slowdown of only 1% compared to the unlimited case. It is a much more meaningful base for comparing energy reduction in the register file than the register file with unlimited resources.

Various configurations of the baseline register file with a limited number of read and write ports were considered. The configuration with the best energy-performance had 8 read and 6 write ports. The use of 8 read ports results in only a 0.17% slowdown compared to 16 ports. The use of 6 write ports results in a 0.21% IPC loss. The baseline integer register file uses this configuration.

The configuration of the content aware register file also needs to be defined. The important parameters here are the value of *d+n* and the size of various register sub-files. The IPC of a number of configurations was measured to help select the best overall configuration that reduced energy consumption and access time with minimal performance loss.

Figure 5 shows the effect of the *d+n* parameter on the relative IPC for various organizations. The 100% performance corresponds to the register file with unlimited resources. The baseline IPC is also shown. It is clear that for all benchmarks the baseline configuration performance is virtually identical to that of a register file with unlimited resources.

The number of *simple* registers is always equal to the number of physical registers in the processor, i.e. 112 in this case. The results in Figure 5 were obtained using 48 *long* and 8 *short* registers. The choice of these sizes is explained next.The effect of the size of the *short* register file on performance is as follows. Even a very small number of short registers delivers close to the maximum attainable IPC of 98+% for integer and 99+% for floating point benchmarks. The results are not significantly better for 32 *short* registers. *8 short* registers are chosen as the best solution providing a small IPC increase over the configuration with just 2 registers.

The effect of *long* register file size was studied in a similar way using *112* physical registers and *d=20*. The IPC for 48 long registers is practically the same as for 112 registers, IPC loss for 40 as compared to *48* registers is 0.6%. Floating point applications need *56 long* registers to reach near-maximum baseline

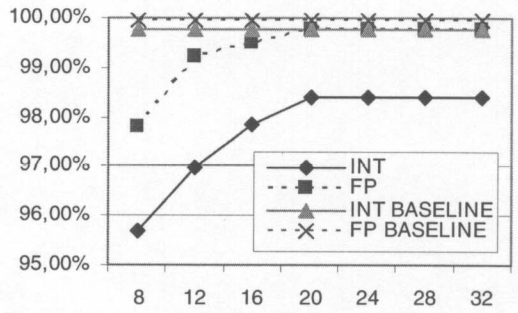


Figure 5. Average Relative IPC as a function of *d+n*

performance of 99.75%. Numerical applications have a very low overall IPC loss reaching 99.6% of baseline with *48* long registers. The resulting choice is 48 long registers.

d+n equal to *20* bits is selected based on the results from Figure 5.Integer applications reach a near-optimum point at *20*, while floating point are extremely close to maximum IPC. Higher values are not worth it since any increase in *d+n* will lead to higher energy consumption, area and cycle time.

One other alternative design for the *short* register file is to use a fully associative search for values instead of direct indexing, which was described in Section 3.1. This was found lead to a very small increase in IPC, but the energy consumption increase is high compared to the *n*-bit index access and compare. The reason is that a fully associative search requires the use of a CAM, which is slow and energy inefficient.

Finally, the number of register ports on each of the simple, *short,* and *long* sub-files was left unchanged from the baseline configuration. In fact, additional ports were provided in the Short file for comparison access in stage WB1 (this is not a problem given how small this file is). The number of accesses to each sub-file is significantly reduced (as shown below) which should allow a reduction in the number of both read and write ports. However, this will lead to additional control complexity since the *long* register allocation is done after scheduling, while the potential energy savings are estimated to be relatively low.

One would expect that the IPC, given the pipeline with two stages for register file access, is the same regardless of the *simple* register file width (*d+n*). Instead, the IPC loss decreases with increase in *d+n* for both integer and fp applications. The reason is that a smaller *d+n* results in a lower value reuse in the *short* registers and, therefore, more values are classified as being *long*. This increases the probability of stalls due to lack of *long* registers (and of the *pseudo-deadlock* described above).

The impact of using the content aware register file

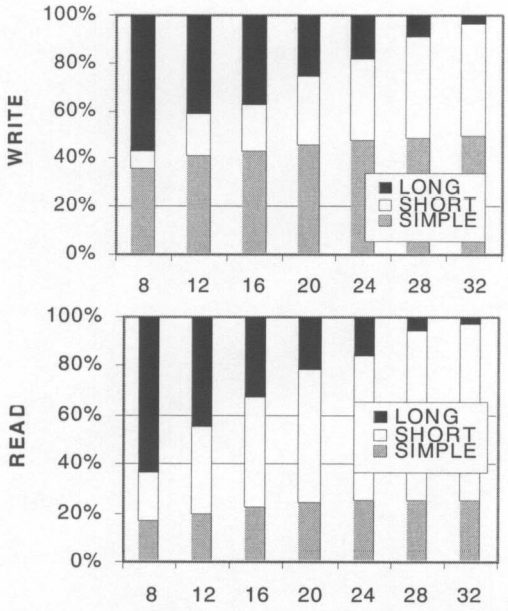


Figure 6. Register file READ and WRITE access distribution by value type (d+n)

with 8 *short* and 48 *long* registers can be seen in the distribution of the read and write accesses for each value/register type. The results presented in Figure 6 also show the dependence of the distribution on the value of $d+n$ (where n is fixed at 3), the width of the *simple* file. The processor architecture contains bypass logic, which eliminates many read accesses to the register file in the content aware architecture (see Table 2 and [11][5]). This architecture has an additional level of bypassing which further reduces the number of register file accesses, as can be seen in Table 2 (and actually reduces energy savings in the register file).

The Figure 6 shows the relative number of accesses to the *long* register file to decrease with increase in $d+n$, as can be expected. Larger d implies that more values are considered *short* and *simple* rather than *long*. The *long* register file consumes more energy per access than the other two, so this is a good trend for energy savings. The results show that for $d+n=24$ over 50% of all the accesses are short values. The number of *long* accesses is below 20%.

Table 2 shows the percentage of all source operands that came from bypass logic and did not require a register access. This is something well known and techniques such as the operand bypassing in the functional units take advantage of this fact [11][5].

Table 2 Percentage of bypassed operands

	Baseline	Content Aware
SPEC INT	*38,1%*	*47.9%*
SPEC FP	*21,1%*	*28.4%*

Table 3. A simgle access energy for each register file normalized to the unlimited resource file

d+n	Simple	Short	Long	Baseline
8	8.2%	3.6%	20.8%	48.8%
12	11.3%	3.4%	19.5%	
16	14.3%	3.2%	18.2%	
20	17.3%	2.9%	16.9%	
24	20.4%	2.7%	15.6%	
28	23.4%	2.4%	14.3%	
32	26.4%	2.2%	13.0%	

The percentage is lower for baseline architecture because it uses only one level of bypass.

5. Energy, area and cycle time considerations

A model proposed by Rixner et al [20] is used in this work to estimate area, access time, and access energy of the register file. It is done for both the baseline and the content aware organizations. The results are shown in Figure 7 and Table 3.

Table 3 shows access energy for each register file type and the effect of $d+n$. All results are relative to the unlimited organization. These results are obtained by multiplying single access values from Table 3 by the number of accesses for each type. The results show that the baseline configuration was chosen well, it uses just under 50% of the unlimited resource register file energy with no appreciable decrease in IPC. The content aware register file organization further decreases the energy consumed in the register file by another factor of 2. Using this figure in conjunction with Figure 5 can determine the value of d+n which delivers the highest energy-delay product.

Figure 8 presents the area of the three register files relative to the unlimited resource register file. The content aware register file is only 82.1% of the size of the baseline register file.

Another result with implications for performance is

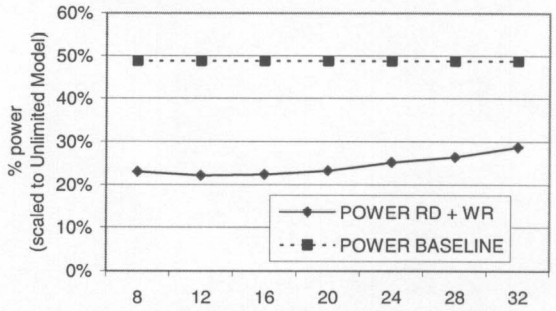


Figure 7. Relative register file energy consumption with respect to the baseline

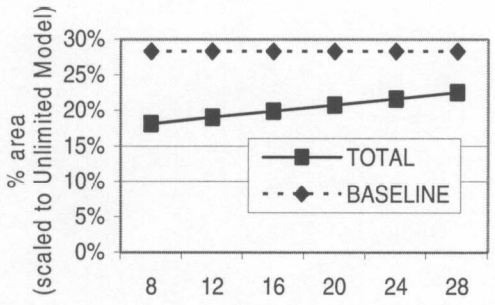


Figure 8. Relative area of the register files

shown in Figure 9. The figure shows the relative access time of the register files. All components of the content aware register file are faster than the baseline register access.

A shorter access time potentially allows a higher clock frequency and this figure indicates a possible 15% increase in clock frequency. Several studies [16][5] show that the register file is on the critical path, and thus any decrease in its access time would be beneficial. However, the exact improvement can only be determined after a careful design. Even if the register access time was reduced by 15%, another stage can become the critical path.

With these disclaimers, let us estimate the increase in performance based on potential frequency increase achieved by our mechanism. If the frequency could be increased by a 5%, the 1.5% average slowdown would turn into a 3% average speed-up. A 10% to 15%, increase could lead to a speed-up between 8% and 13%.

6. Other Potential Optimizations

The content aware register file proposed in this paper has the potential to change the way a superscalar out-of-order processor manages data and registers that goes beyond the energy savings or the clock cycle reduction discussed above. This section presents three additional directions for utilizing the content aware organization to further improve performance. These ongoing projects show very promising preliminary results.

Consider the source operand distribution in integer instructions based on value type. Most of the time both operands are of the same type (for over 86% of all instructions), as can be seen in Table 4. It is also true that the result operand is typically of the same value type as the source operands. Therefore, using the content aware register file allows a different type of clustered micro-architecture to be defined. Existing clustering mechanisms are driven by register usage, the

Table 4. Operation distribution (for d+n=20 bits)

Percentage of operations based on source operands	
Only simple operands	47.4%
Only short operands	21.7%
Only long operands	17.5%
Combination of simple and short	6.3%
Combination of simple and long	6.2%
Combination of short and long	1.0%

dataflow structure of an application, or the availability of functional units. Instead, clusters can be based on the value type in the content aware register file organization. This can lead to performance improvement as well as energy savings.

For instance, a narrow but very fast cluster for simple values can be created. Another cluster can deal with *short* values, which are also highly correlated with memory addresses. The results above indicate that there will be little inter-cluster communication in such architectures. Preliminary results using this type of clustering are very promising.

Another promising direction is to use the content aware organization with simultaneous multithreading. Results indicate that the fraction of *long* register accesses is, on average, very small. In addition, the number of live *long* registers is very small (on average, only 12.7 registers). The need for *48 long* registers is driven by peak usage rather than by the average number of live *long* registers. This indicates that a smaller number of *long* registers can feed more than one thread, especially if only one of them has high peak register usage. This opens many interesting lines of inquiry, such as what are the best thread priority policies for this kind of simultaneous multithreading architecture. This is the direction currently pursued by us. The demand for registers and ports is much higher in SMT architectures [6][4], therefore a larger

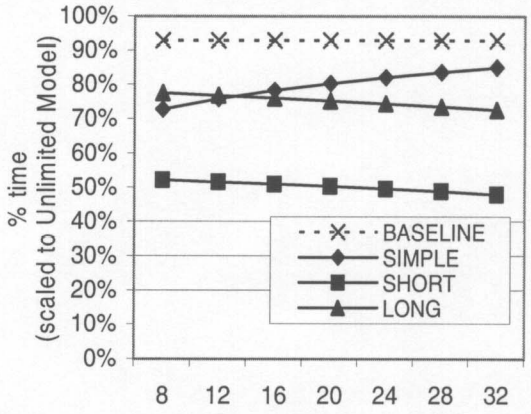


Figure 9. Relative access time of the register files (d+n)

improvement from content aware organization can be expected. Preliminary results are also very promising for this approach.

Memory hierarchy is another very promising area to consider. Values similar to both *long* and *simple* have been explored in previous work. However, both addresses and data have considerable partial value locality and its use can reduce both the energy and the time of cache or memory access. In addition, it can be exploited in the Load/Store unit and in combination with clustering.

7. Related Work

The body of related work on register file energy optimization is large. Some of it has already been mentioned in the paper. Other work can be divided into two broad categories. The first exploits data compression in memory (from main memory to the register file) and is orthogonal to the ideas presented in this paper. The second category changes the register file organization.

IBM developed a memory compression technique based on the X-match compressor called *Memory Expansion Technology (MXT)* [21]. This algorithm works with large blocks of memory and is therefore slow. It is most appropriate for main memory compression.

For lower levels of cache hierarchy there exist proposals such as X-RL [9], which compresses data in the L2 cache. Random access to this data is not allowed and therefore this technique can not be implemented in the L1 cache due to potential performance impact.

L1 data compression based on value reuse uses a small table to keep track of the most frequent values using a type of profile-based dictionary [25]. This work introduced the concept of *frequent* values, which has been extended into frequent partial values in our work. Another contribution uses dynamic zero compression is described in [12]. It only compresses zero-valued bytes.

There are a number of papers trying to exploit value locality in the register file. [19] used the concept of virtual-physical to join several physical registers when they share the same value. [18] proposed a model to optimize the use of specific values, 0 and 1. They are all superseded by a mechanism based on *d*-similarity.

A concept of byte-to-byte compression across the processor datapath was introduced in [17]. When a particular zero or minus one value is found, the processor works with the compressed version of the data. The authors of this paper focused on energy savings, since the mechanism relies on separate parallel banks to store the compressed information.

Modified register file organizations, such as [24], concentrate on reducing the number of ports in a register file to reduce energy. The authors assert that a reduced number of ports may be more efficient both in energy and access time, which can improve performance.

Other studies proposed mechanisms to reduce the number of the ports by means of modifying the register file architecture, such as [11][2][5]. The mechanism proposed in this paper is completely orthogonal to this concept and can also benefit from a reduced number of ports.

Other work attempts to improve the register allocation algorithm to decrease the number of necessary registers. These can be divided into three subgroups. *Early recycling* frees registers before the commit phase [13][16][8][22]. *Virtual registers* try to delay the allocation of a physical register until the writing of the register [1]. Hierarchical register files, such as those presented in [11][16], effectively trade size, speed, and power consumption.

8. Conclusions

This paper introduced the concept of partial value locality and shown its existence in register file values using the notion of *d*-similarity. Data values were divided into three types: long, short, and simple, and a new integer register file organization was proposed based on this. It consists of three separate register files: simple, short, and long. Each of these files has a reduced size and lower frequency of access because the use of value type information allowed register accesses to be directed to the type-specific register file. The paper also described pipeline modifications for utilizing the new register file.

The change in size and frequency of access allowed significant energy savings, while the impact on IPC was almost negligible. The register file area and access time were also noticeably reduced. Overall, the register file implementation described saves, on average, 50% of the energy compared to the baseline register file with an average IPC loss of 1.7% (for SPECint2000). Even further, this loss can be turned into a gain of up to 13% if the access time reduction (up to 15%) would allow a clock frequency increase. This is especially significant if one takes into account that the baseline architecture already used a reduced number of registers and read/write ports.

It is very difficult to directly compare the energy savings and performance loss of the proposed approach and other recent proposals, such as [5][15]. They used different simulators, architectures, and energy models. Nevertheless, we are going to try to approximate. The

reason is that all three papers report relative improvement over an unlimited resource architecture. The unlimited resource register file in [15] has 512 entries and the one in [5] has 180 entries, both with 16 Read and 8 Write ports. The relative energy savings reported in [5] are 67% and those reported in [15] are 60%. The design proposed in this paper produces energy savings by 77% (but only 50% compared to baseline). And this is for a smaller register file of 160 registers with 16 read and 8 write ports.

Several ideas for future work utilizing the approach presented in the paper were discussed. They are currently under development and promise to open additional, new research directions.

Acknowledgement

The authors are also grateful to the reviewers for their extensive comments that have helped to produce a better paper. This work has been supported by the Ministry of Science and Technology of Spain, under contract TIC-2001-0995-C02-01, by the CEPBA and HiPEAC European Network of Excellence.

References

[1] A. González, J. González, and M. Valero. Virtual-physical registers. In Intl. Symposium on High-Performance Computer Architecture, February 1998.

[2] A. Seznec, E. Toullec, and O. Rochecouste. Register write specialization register read specialization: a path to complexity-effective wide-issue superscalar processors. In Intl. symposium on Microarchitecture, 2002.

[3] A J. Smith. Cache memories. In ACM Computing Surveys (CSUR), pages 473-530. ACM Press, 1982.

[4] Emer. Ev8: The post-ultimate alpha. In Keynote speech at the Intl. Conference on Parallel Architectures and Compilation Techniques, September 2001.

[5] I. Park, M. Powell, and T. Vijaykumar. Reducing register ports for higher speed and lower energy. In 35th intl. symposium on Microarchitecture, , 2002.

[6] J. Lo, S. Parekh, S. Eggers, H. Levy, and D. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. Technical Report TR-97-12-01, University of Washington, Department of Computer Science and Engineering, 1997.

[7] J. Tseng and K. Asanović. Energy-efficient register access. In 13th Symposium on Integrated Circuits and Systems Design (SBCCI'00), September 2000.

[8] J. F. Martínez, J. Renau, M.C. Huang, M. Prvulovic, and J. Torrellas. Cherry: checkpointed early resource recycling in out-of-order microprocessors. In 35th intl. symposium on Microarchitecture, 2002.

[9] J. S. Lee, W.-K. Hong, and S.-D. Kim. Design and evaluation of a selective compressed memory system. In Intl. Conference on 26 Computer Design (ICCD '99), October 1999. IEEE.

[10] J. H. Tseng, Krste Asanović. Banked multiported register files for high-frequency superscalar microprocessors. In 30th intl. symposium on Computer architecture, June 2003.

[11] J. L. Cruz, A. González, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In 27th Intl. Symposium on Computer Architecture, 2000.

[12] L. Villa, M. Zhang, and K. Asanović. Dynamic zero compression for cache energy reduction. In 33rd annual intl. symposium on Microarchitecture, 2000.

[13] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In 26th annual intl. symposium on Microarchitecture, 1993.

[14] M. H. Lipasti and J.P. Shen. Exceeding the dataow limit via value prediction. In 29th annual Intl. symposium on Microarchitecture, 1996.

[15] N. Sung Kim, T. Mudge. Reducing Register Ports Using Delayed Write-Back Queues And Operand Pre-Fetch. In of 17th Intl. Conference on Supercomputing. June 2003.

[16] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically allocating processor resources between nearby and distant ilp. In 28th Intl. symposium on Computer Architecture, 2001.

[17] R. Canal, A. González, and J.E. Smith. Very low energy pipelines using significance compression. In 33rd Intl. symposium on Microarchitecture, 2000.

[18] S. Balakrishnan and G. Sohi. Exploiting value locality in physical register files. Research report, University of Wisconsin, 2002.

[19] 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 31st Intl. symposium on Microarchitecture, 1998.

[20] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens. Register organization for media processing. In Sixth Intl. Symposium on High-Performance Computer Architecture, January 1999.

[21] T. B. Smith, B. Abali, D. E. Po_, and R. B. Tremaine. Memory expansion technology (mxt): Competitive impact. March 2001.

[22] T. Monreal, V. Viñals, A. González, and M. Valero. Hardware schemes for early register release. In Intl. Conference on Parallel Processing, 2002.

[23] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In Intl. Conference on Parallel Architectures and Compilation Techniques, 2001.

[24] V. Zyuban and P. Kogge. The energy complexity of register files. In Intl. symposium on Low energy electronics and design, 1998.

[25] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value centric data cache design. In ninth Intl. conference on Architectural support for programming languages and operating systems, 2000.