

Register Multimapping: Reducing Register Bank Conflicts through One-to-many Logical-to-Physical Register Mapping

Nam Le Duong and Rakesh Kumar
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Abstract

Banked register files have been proposed as a way to alleviate the latency and wiring overheads of centralized register files. However, banking can result in read/write conflicts that did not exist for centralized register files. In this paper, we propose register multimapping, a technique to reduce register bank conflicts. Register multimapping involves mapping an architectural register to multiple physical registers belonging to different banks. Reads can proceed using any of the physical registers, thereby minimizing read bank conflicts. Register multimapping can increase write conflicts, however. To alleviate write conflicts, we also investigate enhancing register multimapping with delayed allocation of physical registers [16, 13]. Our experiments show that register multimapping can result in performance improvements up to 14.9% (10% on average). It can also allow port reduction at minimal cost. Halving the number of read ports resulted in register area savings of 43% and register power savings of 48% for only 3.2% degradation in performance.

1 Introduction

Residing at the heart of a modern processor, the register file, similar to other memory structures (e.g., main memory and caches), is responsible for storing temporary data during computation. Compared to these memories, the register file is the structure that nonmemory functional units (e.g., arithmetic, logic units) directly communicate with. The register file therefore must provide high bandwidth and low latency.

Unfortunately, a modern processor needs a register file with a large number of registers and read and write ports, making it a complex structure. This increases latency of the register file significantly. The complexity of the register file also results in increased area and power consumption.

Register banking is a popular technique to reduce the latency [5], area [23], power, and wiring overheads of large centralized register files in VLIW [7], DSP [21], Stream [11], SMT [25, 24], and high frequency superscalar [10] processors. The low overheads are due to the small number of ports that each memory cell in a banked register file is connected to as compared to a memory cell in a centralized register file. As power and area become zero-order design constraints for future processors, the pressure to partition register files into banks is only going to increase.

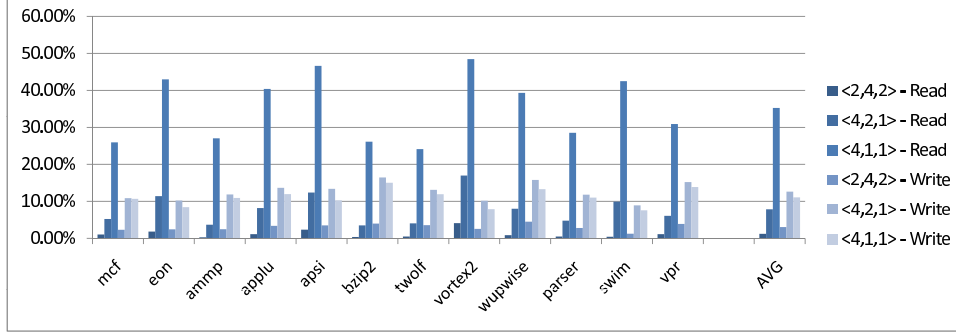


Figure 1. Percentage of register read and register write attempts that fail due to bank read and write conflicts, respectively.

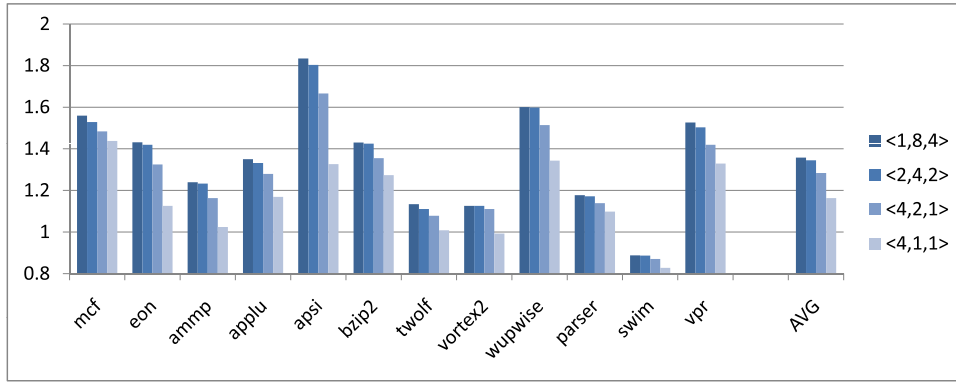


Figure 2. Absolute IPC for processors with different bank organizations of the register file.

Banking, however, also results in bank conflicts that do not exist for centralized register files. Bank conflicts arise due to the unavailability of read/write ports in a register bank when multiple instructions are competing for the same bank in a given cycle. Figure 1, for example, shows how the number of read and write conflicts changes as the number of banks is increased for a fixed number of read and write ports. $\langle B, R, W \rangle$ corresponds to a register file organization with B banks, R read ports, and W write ports (methodology described in Section 4). Bank conflicts often delay ready instructions in the instruction queue as sources are not available, although they are ready. Similarly, finished instructions often cannot be committed due to write port conflicts and have to wait until the next cycle to compete for available port(s). Figure 2 shows the effect of these conflicts on the processor IPC. As can be seen, the impact of bank conflicts on processor performance can often be significant, forcing the designers to have more ports at the expense of area and power.

In this paper, we propose *register multimapping*, a technique for reducing register bank conflicts. Register

multimapping involves mapping an architectural register to multiple physical registers (two in this work) belonging to different banks. A value is written into both the physical registers. When a subsequent instruction needs to read the architectural register, the read for a value has two banks to try from, unlike the conventional banked register file approach where there is only one bank to read a value from. This doubles the probability of finding a free port to read a certain source, thereby minimizing bank read conflicts.

We also investigate *enhanced register multimapping* where we combine register multimapping with delayed allocation of physical registers until the writeback stage [13]. Delaying the allocation of physical registers until writeback stage has been shown to minimize bank write conflicts.

We apply our technique for both single-threaded execution and multithreaded execution. SMT processors, while providing higher IPC than superscalar processors, will suffer more conflicts due to the higher number of instructions to be issued in a certain cycle.

We also study factors that affect benefits from register multimapping. The factors include the number of registers, bank port organizations, and IPC. By studying these factors in relation to the register file's area, latency, and power consumption, as well as performance of the whole system, best configurations of the register file with register multimapping technique are determined.

Our experiments show that register multimapping can result in performance improvements up to 14.9% (10.0% on average). Results also indicate that register multimapping allows port reduction at minimal cost. Halving the number of read ports results in register area savings of 46% and register power savings of 48% for only 3.2% degradation in performance. We also see that register multimapping and delayed allocation are synergistic, and combining the two significantly reduces total number of bank conflicts.

The rest of the paper is organized as follows. Background of the work is discussed in Section 2. Section 3 discusses the register multimapping technique. Simulation methodology is presented in Section 4. Section 5 focuses on simulation results. Conclusions are given in Section 6.

2 Background

2.1 Physical Register Files in Modern Processors

2.1.1 The physical register file and design requirements

Residing at the heart of a modern processor, the register file, similar to other memory structures (e.g., main memory and caches), is responsible for storing temporary data during computation. Compared to these memories, the register file is the structure that nonmemory functional units (e.g., arithmetic, logic units) directly communicate with. The register file therefore must provide high bandwidth and low latency. Unfortunately, a modern processor needs a register file with a large number of registers and read and write ports, making it a complex structure. This increases latency of the register file significantly. The complexity of the register file also results in increased area and power consumption.

Parameters that impact the design of a conventional register file include the physical registers, ports (read and write), and organization (connections between ports and memory cells). Complexity increases as the number of ports and/or the number of registers increase. Given a fixed number of registers and ports, the number of connections between the memory cells and ports determines the complexity. For example, a centralized register file has higher complexity than a banked register file given the same number of registers and ports, due to the fact that every memory cell must be connected to all ports.

In modern processors, the minimum number of required registers is equal to the number of architectural registers. Often, a physical register file contains more physical registers. Extra registers are used in register renaming, a technique used to resolve false dependencies (WAR and WAW). In conventional processors, architectural registers are mapped (renamed) to physical registers in rename stage. In subsequent stages of the processors, indices of the physical registers are used instead of those of architectural registers. Physical registers are freed when they are no longer used (e.g., when the value of an allocated physical register is about to be reproduced by another instruction). More details can be found in [8].

In practice, the number of physical registers is determined by balancing the trade-off between the fraction of instruction stalls due to the lack of free physical registers at the rename stage (rename stalls) and complexity of the register file. In order to guarantee zero rename stalls, the number of physical registers that are needed is:

$$P = A + R$$

where P is the number of physical registers, A is the number of architectural registers, and R is the size of the reorder buffer (ROB). This number is calculated by taking into account the worst case, when the ROB is full and every in-flight instruction produces a register result. However, because about 25%-40% of in-flight instructions are store and branch instructions, which do not produce register results, the number of required registers can be smaller.

Another factor that is considered during the design of physical register files is the number of read and write ports. Ideally, no instructions are stalled due to the lack of read and write ports in the register file. This means that every instruction being issued is able to read sources from the register file, and every finished instruction is able to write results back. In an n -way processor, in the peak case, the maximum number of instructions to be issued and the maximum number of instructions to be written back are both n . As an instruction has at most two sources and one destination, the read port requirement of a centralized register file, for example, with no conflicts, is $2 \times n$, and the write port requirement with no conflicts is n .

Beside number of registers and number of ports, organization of registers is also considered carefully when designing the register file. Organization pertains to the way connections are made between memory cells and ports. A banked organization, for example, is where a memory cell is connected to only some certain ports, as opposed to a centralized register file which involves a memory cell being connected to all ports.

In this work, register multimapping is studied for a banked register file with different numbers of physical registers and different numbers of ports.

2.1.2 Issues with a centralized register file

As superscalar processors offer high IPC, the number of physical registers must be large to support a large number in-flight instructions. Register file sizes are even larger for SMT processors. An n -thread workload being executed concurrently has as many as $n \times A$ architectural registers and requires many more registers to resolve false dependencies. Since the register files lie in the critical path when determining the processor frequency, large register files can slow down a processor. In [25], for example, the number of physical registers is as high as 384, making register file access latency to be 2 clock cycles. This not only impacts performance of the whole system, but also needs extra logic to deal with a 2-cycle register file access.

As Table 1 shows, increasing register file complexity also results in large area and power consumption.

Table 1. Area, latency, and power consumption of various centralized register file organizations with 64-bit registers for 65-nm technology (INT denotes integer registers, FP denotes floating-point registers).

Organization	INT	FP	Area (mm^2)	Latency (ns)	Dynamic power (W)	Leakage power (mW)
$\langle 1, 8, 4 \rangle$	96	96	6.92	1.28	3.43	220.20
$\langle 4, 2, 1 \rangle$	96	96	2.67	1.06	3.56	220.20
$\langle 4, 1, 1 \rangle$	96	96	1.43	1.02	1.84	120.15
$\langle 1, 8, 4 \rangle$	80	80	6.22	1.22	2.55	184.09
$\langle 4, 2, 1 \rangle$	80	80	2.40	1.00	3.14	184.09
$\langle 4, 1, 1 \rangle$	80	80	1.22	0.97	1.63	100.42

The table shows the impact of number of registers and number of ports on a register file’s latency, area, and power consumption. We used CACTI [22] to produce this table, with the assumption that 65-nm technology is used and the register sizes are 64 bits. The table shows that the change in port organization has greater impact on the latency, area, and power characteristics of a register file than a change in number of registers. Latency, area, and power consumption drop significantly when the register file is organized into banks ($\langle 4, 2, 1 \rangle$ and $\langle 4, 1, 1 \rangle$) as compared to the centralized register file ($\langle 1, 8, 4 \rangle$). Meanwhile, when the number of registers of each type (integer and floating point) falls from 96 to 80, the change is not significant.

2.1.3 Banked register file

Register banking is a popular technique to reduce the complexity of the register file. This technique changes the register file organization by splitting it into multiple banks, each bank containing a certain number of physical registers. Memory cells in a bank can only be connected to read and write ports belonging to that bank. A centralized register file can be considered a special case, where there is only one bank.

The advantage of register banking is that it reduces the complexity by cutting down the connections between memory cells and ports, while the number of ports and the number of registers are kept the same. Table 1 shows the benefits in terms of latency, area, and power consumption of a banked register file over a centralized register file. Banking, however, also introduces an issue which does not appear in a centralized register file: *bank conflicts*. Conflicts happen when, due to the lack of free ports, not all simultaneous attempts to read/write from/to a bank can be satisfied. We will discuss this phenomenon later in this section, after comparing the banked register file with the centralized register file.

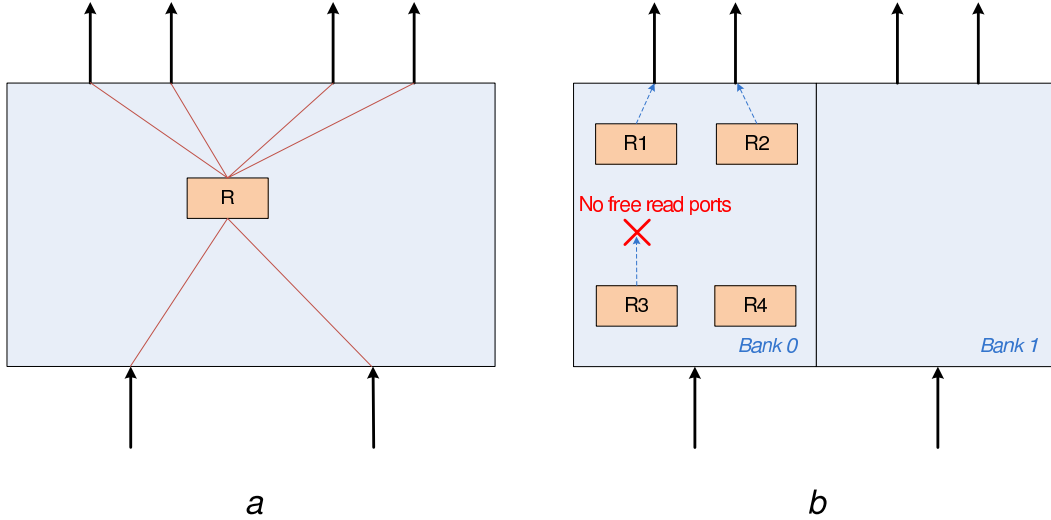


Figure 3. Organization of a banked register file. (a) Centralized register file, where a register is connected to all read and write ports. (b) 2-bank register file with the same number of read and write ports.

Figure 3 illustrates this with an example. Figure 3(a) shows a centralized register file with 4 read ports and 2 write ports. This organization guarantees full issue and writeback of up to 2 instructions per cycle without any stalls. In Figure 3(b), the register file is organized into 2 banks, each with 2 read ports and 1 write port. Half the registers are located in the first bank, and the rest are in the other bank. This is a homogeneous banked register file, where all banks are identical. This is opposed to a heterogeneous banked register file [5], where banks may have different numbers of registers and ports. In our work, we only focus on homogeneous banked register files. For a fair comparison, the two register files have the same number of registers and ports.

To illustrate the bank conflict issue, we consider an example where we have the following two ready instructions in the issue queue (at cycle t_x):

```

ADD   R5 <- R1, R2
MUL   R6 <- R3, #3

```

R1, R2, and R3 are physical registers corresponding to different architectural registers. Unfortunately, all of these register reside in the same bank (bank 0). During cycle t_x , these two instructions want to read data from the three registers, but bank 0 has only two read ports. This means that the two instructions have to compete for read ports. In Figure 3(b), both read ports in bank 0 are used to read sources (from R1 and R2) of instruction ADD. The MUL instruction, meanwhile, cannot read data from R3 because

there are no free write ports during that cycle. Hence, it is delayed in the issue queue before competing for its source again in the next cycle $t_x + 1$.

Port conflicts delay ready instructions by at least one cycle, and this degrades the performance of the whole system. The degradation is high when the fraction of delayed instructions is high, or delay probability of an instruction is high. Probability of conflicts is high when:

1. The number of instructions to be issued or written back is high. This corresponds to high IPC. Out-of-order superscalar processors will suffer more from bank conflicts than simple processors, and it is worse for SMT processors.
2. Average number of source operands or destination operands is high. This is related directly to the applications.
3. Too many attempts to access a certain bank.

While the first and the second reasons are characteristics of applications, the third reason can be reduced or even eliminated by using architectural support. Our technique, register multimapping, exploits free resources to improve the access distribution of read and write ports. We do this by exploiting unused resources during the execution.

2.1.4 Summary

In modern processors, parameters greatly impacting the architectural design of a physical register file include number of ports, number of registers, and organization. Computer architects must take into account these parameters to reduce complexity of the register file. Reducing complexity helps reduce delay, area, and power consumption of the register file. Register banking is a popular technique, which reduces the complexity of the register file. However, register banking introduces port conflicts, a phenomenon that can degrade processor performance.

2.2 Resource Redundancy in Modern Processors

Register file resources, as well as other resources, are designed for peak execution. In a 4-way superscalar processor, while a register file may have up to 8 read ports and 4 write ports, the fraction of time when all ports are busy is small. The reasons why the utilization is low include (this is an extension of [3]):

1. An n -wide processor is able to issue up to n instructions per cycle and write back up to n instructions; however, due to dependencies, not all issue slots or writeback slots are busy all the time.
2. In case of exceptions (e.g., branch misprediction), there would be no issued and written back instructions.
3. Many instructions have single source operands.
4. Many instructions produce results that are not written into the register file, such as stores, branches, or address computation.
5. Some source operands can be obtained from the bypass logic instead of from the register file.
6. There may be multiple operands which get data from unique registers in a cycle, and in this case there is only one read.

As an example, in our experiments, write port utilization was about 28.5% of all time, and 44.5% when at least one port is used.

Similarly, while the number of registers is often kept high to guarantee the worst case execution, as discussed in the previous section, our experiments showed that average usage of registers is only about roughly 50%. The reasons for low utilization of registers are:

1. The fraction of time the ROB is full is low.
2. Not all instructions produce register results.

While the utilization of ports and registers is low, it also introduces a chance to exploit them for improving performance. In this work, we utilize these unused resources to reduce port conflicts introduced by the use of banked register files.

2.3 Related Work

Considerable work has been done on demonstrating the benefits of register banking. The work in [3] studied the effect of using banked register files on the processor's performance. Their study showed a significant reduction in access time and power consumption. Tseng and Asanovic [23] show that the

complexity of a banked register file is significantly smaller than that of a centralized one. Cruz et al. [5] come to a similar conclusion about register banking for both single-level and multilevel register files.

There has also been previous work quantifying the impact of bank conflicts on processor performance and studying ways to reduce them [23, 1, 27].

In [23], the authors propose using a speculative control scheme to predict bank conflicts. In this work, a new stage is added into the main pipeline before the execution stage. The task of this stage, which is called the arbitration stage, is to detect when a bank conflict occurs, and to choose the winning instructions to send to the execution stage. The read ports of the banked register file in this work are organized into two sides: left ports are connected to left inputs of functional units and right ports are connected to right inputs of functional units. The speculative technique is proposed to reduce conflicts that happen when the access to one side is too much compared to the other side. When the arbitration stage detects that too many reads occur in the same side, or too many writes occur in the same bank, the instruction window is repaired, and conflicting instructions must be reissued again. The disadvantage of this technique is that it can degrade performance due to a new added pipeline stage, and reissuing conflicting instructions, because write conflicts will take extra cycles before instructions can reach the current state again. Also, this technique does not resolve the case when multiple registers are read or written, but the number of ports is insufficient to provide all the needed values. Our work addresses this issue.

In [1], the lifetime of instructions in the issue queue and reorder buffer is minimized, hence reducing the number of active registers, and reducing the pressure on the banked register file. In order to save power, the authors decrease the instruction window size and turn off banks of register files when they are not busy. In order to make this technique effective, the registers in a bank must be small enough, or the probability of finding a bank with no used registers will be small.

In [27], the authors build a value-aware banked register file, where each bank is used to store data of different widths. This helps reduce power consumed when a long register issued to hold a short value. Three data widths are supported: 64 bits, 34 bits, and 16 bits. A predictor is employed to predict the width of values.

Other efforts have tried to reduce the number of ports in a physical register file. Park et al. [16] propose two techniques to reduce the number of register ports. For reads, bypass hints predict when results can be read from the bypass logic instead of reading from the register file. Similar to [23], this work requires an

extra stage to implement the bypass hint technique. This added stage is used for bypass determination. For writes, the assignment of physical registers is delayed until just before writeback. This approach is similar to the virtual register technique [13] which is used to enhance the usage of physical registers. This technique was also previously proposed in [26]. We use this technique to enhance register multimapping. In fact, we show that register multimapping enhances the effectiveness of this technique.

Other work on port reduction includes [12], in which the reduction of read ports is based on the observation that most results are consumed within a few cycles of being produced. The authors propose using a delayed writeback queue to buffer results which will be consumed within a few cycles. Data can be read from either the register file or the delayed writeback queue. Hence, the number of read ports can be reduced. Reducing the number of write ports is based on the observation that full usage of write ports is rare. The delayed writeback queue is employed to buffer results that cannot be written back due to the lack of write ports. The actual write is done in the next few cycles. The work in [12] also proposes operand prefetch to alleviate port pressure, which is complementary to our work. Because about 80% to 90% of the source operands are ready when instructions are about to be issued, they can be read ahead. An operand prefetch buffer is used for the prefetched operands.

The work in [19] treats instructions with two source operands differently from other instructions. It limits the number of two-operand instructions to be issued per cycle. There is also work studying the usage of replicated register files to reduce the number of read ports [10, 6, 28]. Duplication of physical register file, however, creates redundancy affecting area as well as power consumption, while it increases the complexity of the write logic.

Finally, value locality in the physical register files has led computer architects to investigate multilevel register files [5, 3, 29, 2] as well as bypass logic to reduce the pressure on the register file [12, 17].

3 Register Multimapping

Register multimapping (RM) is proposed to reduce both read and write conflicts. The technique to reduce read conflicts is *basic register multimapping* (BRM), which maps an architectural register to multiple physical registers in different banks. In this work, we focus on the mapping of one architectural register to two physical registers. The two physical registers are called *main register* and *auxiliary register*, respectively. We distinguish between the two registers because while one physical register is mapped compulsorily

during renaming (by definition), the other register is mapped only optionally. Data can be read from either register depending on available read ports. The disadvantage of BRM is that it increases write bandwidth as both registers have to be written. This might harm the performance of the whole system.

To overcome the disadvantage of BRM, we also investigated *enhanced register multimapping* (ERM). Besides reducing read conflicts, ERM also reduces write conflicts, and even eliminates penalties caused by the write into auxiliary registers in BRM. It does so by combining BRM with *late binding* (LB). LB is a technique which delays the allocation of physical register until writeback [13, 16]. It uses *virtual registers* for resolving false dependencies. The combination exploits the advantage of both techniques, as both main registers and auxiliary registers used by BRM are delayed when using LB.

This section is organized as follows. In Section 3.1, BRM is discussed. The discussion of ERM is in Section 3.2. For each kind of multimapping, architectural changes are proposed and analyzed. Finally, parameters that impact RM are presented in Section 3.3.

3.1 Basic Register Multimapping

3.1.1 Motivation

In the previous section, we showed an example of read port conflicts for a banked register file (see Section 2.1.3). In that example, when two instructions, ADD and MUL, are ready to be executed, only the ADD instruction can start executing, while the MUL instruction is delayed because it cannot find a free read port. In this section, the example is employed again to illustrate the idea of RM, including BRM and ERM (see Figure 4).

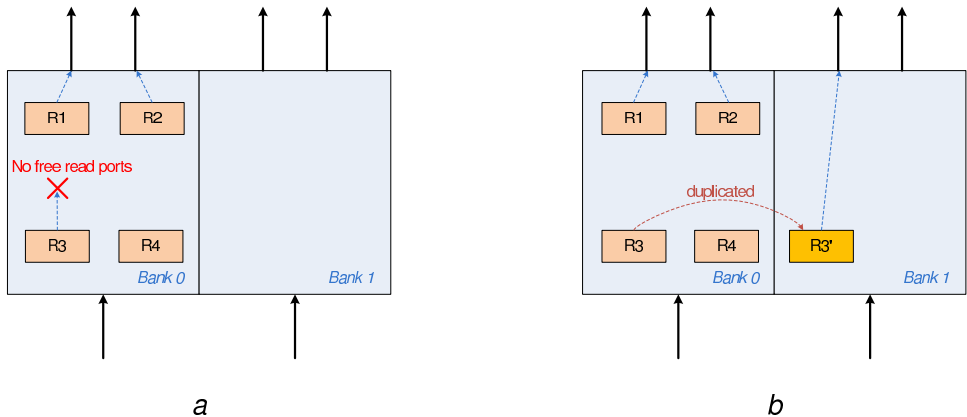


Figure 4. Register multimapping technique for banked register file.

Figure 4(a), which shows an example of read conflicts, is identical to Figure 3(b), and is presented here for the purpose of comparison. The idea of RM is illustrated in Figure 4(b). In this figure, the architectural register which is mapped to R3 (main register) in bank 0 is also mapped to register R3' (auxiliary register) in bank 1. This duplicates the two registers in different banks. The probability finding a free read port to read the operand for the MUL instruction is thus increased as it can be read from either R3 or R3'. Although the ADD instruction has used up all the read ports in bank 0, a free read port is found at bank 1, so data is read for the MUL instruction from R3' instead of R3, and the instruction can start being executed in the same cycle with the ADD instruction.

Because an architectural register is mapped to multiple physical registers in RM technique, one might suspect that it will reduce the number of free registers when extra registers are used as auxiliary registers, and more write ports are needed to write data into more registers. Our technique avoids this issue by exploiting free resources when they are not used, as shown in Section 2.2. The auxiliary register is allocated only when the freelist is not empty. Also, it may be taken out of the pool of allocated registers and then mapped to a new architectural register as a main register to avoid rename stalls. Similarly, a write to auxiliary registers is done only when there are free write ports. If a free write port is not found, the auxiliary register is deallocated to avoid delays due to write port conflicts.

Our BRM technique is proposed to resolve read conflicts. Similarly, it can be used to resolve write conflicts by writing results that can be written into either register (main or auxiliary register). However, this is not necessary, as we use LB [13] to resolve write conflicts. In fact, LB is a good technique to resolve write conflicts. A similar technique cannot be used to reduce read conflicts, as information about read conflicts is not known at the time registers are about to be allocated. That is the reason why BRM is proposed for read conflicts, while LB is employed for write conflicts, and ERM combines both of them.

We have discussed BRM with distinguished main registers and auxiliary registers. As mentioned before, this is necessary because the index of the main register is used during the execution, and the index of the auxiliary register is used when reading or writing data. However, with the combination with LB, the concepts of main registers and auxiliary registers are not different, as the index of the virtual register is used in place of the main register, and indices of the physical main and auxiliary registers are used to access appropriate values inside the register file.

Several issues need to be resolved in order to make register multimapping work:

1. Microarchitecture changes to support storing information about auxiliary registers.
2. When and how to map/unmap an auxiliary register. Auxiliary registers, similarly to the main registers, are allocated and deallocated frequently. Allocation and deallocation policies must be determined.
3. How to read/write operands with multiple registers. Because multiple registers contain a value for register multimapping, there must be mechanisms to read and write these registers.

3.1.2 Architecture support

A new table, called the *multimapping table* (MT), is added into the main pipeline. Each entry of the table contains multimapping information for a physical register. The number of entries is equal to the number of physical registers. Each entry contains the following fields:

- R_M Index of the main physical register.
- S_M If this bit is set, the physical register contains a valid value. This bit is used to determine whether data has been written into the main register.
- I_A If this bit is set, the auxiliary register has been mapped. This bit is used to determine whether the main register has an attached auxiliary register.
- R_A Auxiliary register index (valid only when I_A is set).
- S_A If this bit is set, the auxiliary register contains a valid value (valid only when I_A is set). This bit is used to determine whether data has been written into the auxiliary register.

Entries are accessed through register index R_M . In order to support the search for auxiliary registers (to be discussed later), the table is organized as a CAM to search for an I_A bit whose value is 1.

Figure 5 shows the position of the MT in the processor pipeline, and the communication between it and different stages. The table is accessible from the following processor structures:

1. *Rename logic*, when renaming registers and freeing registers, as main and auxiliary registers are allocated/deallocated.

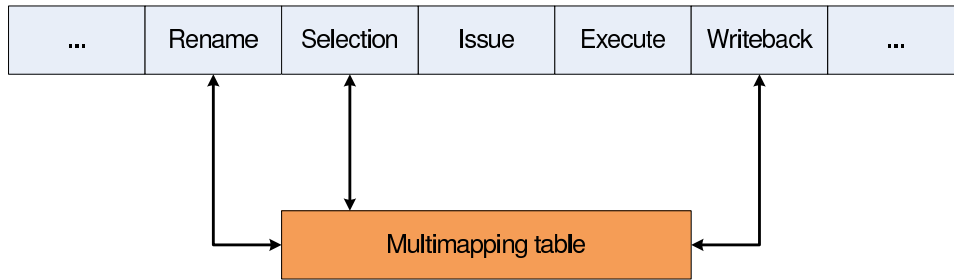


Figure 5. Register multimapping table.

```

allocate()
  if (the freelist is not empty) then
    Get the register index from the freelist to  $R_M$ ;
    Unset  $S_M$ ;
    if (the freelist is not empty) then
      Get the register index from the freelist to  $R_A$ ;
      Unset  $S_A$ ;
      Set  $I_a$ ;
    end if
    The main register index is  $R_M$ ;
  else if (found an allocated auxiliary register  $R'_A$ ) then
    // Deallocate the auxiliary register
    Unset  $I_a$  of the entry which contains  $R'_A$ ;
    The main register index is  $R'_A$ ;
  else
    Stall the rename stage;
  end if
end

```

Figure 6. Allocating physical registers.

2. *Selection logic*, when determining resource availability for ready instructions before issuing them into functional units. The table is accessed to get information about read ports.
3. *Writeback*, when results are written into the register file from finished instructions in the functional units. Auxiliary registers may be deallocated according to a particular writeback policy. This will be discussed in detail in later sections.

3.1.3 Mapping (allocating) physical registers

Figure 6 shows the pseudo code of the procedure to allocate main and auxiliary registers. When allocating a main register, a register index is obtained from the freelist if it is not empty. In case the freelist is empty but there are allocated auxiliary registers, then one of the auxiliary registers is deallocated and used as the

```

deallocate( $R_M$ )
  Add register index  $R_M$  to the freelist;
  if ( $I_A$ ) then
    Add register index  $R_A$  to the freelist;
    Remove register index  $R_A$  from the auxlist;
    Unset  $I_A$ ;
  end if
end

```

Figure 7. Deallocating physical registers.

main register. To find an already allocated auxiliary register, a CAM search is done in the MT to find an I_A bit that is set to 1. When an I_A bit is set to 1 for a physical register, a corresponding auxiliary register has been allocated. After the first I_A bit is found, this bit is then unset, and the auxiliary register index R_A is used as the main register index. After the main register is allocated, the corresponding S_M is unset.

The allocation of auxiliary registers is also done during this time. When allocating an auxiliary register, only the freelist is considered. If the list is empty, then the auxiliary register is not allocated. Because the auxiliary register must reside in a different bank from the main register, it must be easy to quickly find a free register for any desired bank from the freelist. A simple way to do this is to split the freelist into sublists, each containing indices of registers for a bank. After an auxiliary register is found, the index of the register is written into R_A , the corresponding I_A bit is set, and the S_A bit is unset.

3.1.4 Unmapping (deallocating) physical registers

When the main register is about to be freed, its index R_M is added into the freelist. If its I_A bit is set (the main register has an attached auxiliary register), then the attached auxiliary register is also deallocated. To deallocate an auxiliary register, the index in R_A is added into the freelist, and the bit I_A is unset. The procedure is illustrated in Figure 7.

An auxiliary register is deallocated when (1) the freelist is empty while trying to allocate a main register as discussed in Section 6, or (2) when deallocating the main register as in this section. Besides, it may also be deallocated when (3) no free write ports are found to write data to that register. The third case is related to write policy, which is discussed in the next section.

```

write_result( $R_M$ )
  if (not  $S_M$  and found a write port for  $R_M$ ) then
    Write to the main register  $R_M$ ;
    Set  $S_M$ ;
  end if

  if ( $I_A$  and not  $S_A$  and found a write port for  $R_A$ ) then
    Write to the auxiliary register  $R_A$ ;
    Set  $S_A$ ;
  end if
end

```

Figure 8. Writing results into multiple registers.

```

writeback_csvt(instruction)
  with (destination index  $R_M$  of the instruction)
    write_result( $R_M$ );

    if ( $S_M$  and  $S_A$ ) then
      Commit the finished instruction;
    else
      The instruction is delayed at the output of the functional unit;
    end if
  end with
end

```

Figure 9. Writeback conservative policy.

3.1.5 Writing to physical registers

The result, after being produced, is written into the main register and also the auxiliary register if that register exists. Writing into the auxiliary registers used free write ports. Figure 8 illustrates the procedure to write results back to both registers. For writing to either a main register or an auxiliary register, the valid bits (S_M and S_A) are checked to make sure the result has not been written into the register. This helps avoid writing multiple times which wastes write bandwidth. If one of the above bits is not set, then the result is written into the corresponding register (main or auxiliary). The difference between writing back to the main and auxiliary registers is that the result is written to the auxiliary register only when I_A is set.

We studied two policies regarding writes:

1. *Write conservative (WC)*. After an instruction finishes execution, it is committed only after the result

```

writeback_aggs(instruction)
  with (destination index  $R_M$  of the instruction)
    write_result( $R_M$ );

    if ( $S_M$  and not  $S_A$ ) then
      deallocate_aux( $R_M$ );
      Commit the finished instruction;
    else if ( $S_M$  and  $S_A$ ) then
      Commit the finished instruction;
    else
      The instruction is delayed at the output of the functional unit;
    end if
  end with
end

```

Figure 10. Writeback aggressive policy.

is written to *both* registers. This increases the chance to reduce read port contention, but introduces write port contention (see Figure 9).

2. *Write aggressive (WA)*. If no free write port is found for an auxiliary register, the instruction still commits and the auxiliary register is deallocated. This not only eliminates write conflicts due to writing to auxiliary registers, but also reduces the chance of reducing read conflicts (see Figure 10).

3.1.6 Reading from physical registers

When an attempt to read the main register fails because corresponding read ports are busy, a second attempt is made to read the auxiliary register. If the attempt succeeds, data is read from this register. This is illustrated in Figure 11.

The selection logic in the issue window needs to be modified in order to support the above action. We modified the selection logic in [15] to support a banked register file with RM. We considered a baseline implementation similar to [3] where the priority encoder takes into account information about the availability of functional units and contention of register file ports to grant the issue of a ready instruction. In our work, the root cell also takes into account the information about the banks of the auxiliary registers.

```

read_source( $R_M$ )
  if ( $S_M$ ) then
    if (found a free read port for  $R_M$ ) then
      Read source from  $R_M$ ;
      Mark the instruction's source as read;
    else if ( $I_A$  and  $S_A$  and found a read port for  $R_A$ ) then
      Read source from  $R_A$ ;
      Mark the instruction's source as read;
    end if
  end if
end

```

Figure 11. Reading operands from multiple registers.

3.2 Enhanced Register Multimapping

The previous section describes the BRM scheme where read conflicts get minimized at the expense of increased write conflicts. In this section, we discuss how the effectiveness of BRM can be improved by delaying the allocation of physical registers until the writeback stage instead of allocating physical registers in the rename stage.

3.2.1 Late binding of physical registers

In a typical processor, an architectural register is mapped to a physical register at rename stage even though the physical register is written into only at writeback. This unnecessarily increases the lifetime of a physical register [13]. This work proposed using *virtual physical registers* for renaming while delaying the allocation of a real physical register until writeback to reduce the pressure on the register file. A modification of the same technique was used by Park et al. [16] to reduce bank write conflicts. If physical registers are allocated at the renaming stage, there is no information on when the instructions will finish executing (i.e., instructions renamed/issued in the same cycle may not finish executing in the same cycle) and therefore write conflicts can occur. At the writeback stage, we know the instructions that have finished executing. So, if the allocation of physical registers is delayed till the writeback stage, write port contention can be minimized by allocating a physical register in a bank which still has free write ports. In fact, if the number of write ports is equal to the issue width, there will be no write conflicts.

3.2.2 Combining late allocation with register multimapping

The effectiveness of RM depends on the ability to find an auxiliary register from which to read data in case the main register cannot be read due a conflict. The availability of an auxiliary register itself depends on whether an auxiliary register could be previously be written into (i.e., there was no write conflict when an attempt was made to write to the auxiliary register). Since LB of physical registers reduces write conflicts, combining LB with RM will increase the probability of finding available auxiliary registers during reads, thereby increasing the effectiveness of RM.

To combine LB with RM, allocation of main registers and auxiliary registers is delayed until writeback. After write ports for main registers are determined, auxiliary registers are written using the remaining free write ports.

To evaluate the impact of delaying physical register allocation, we studied and compared the following policies:

1. Both main registers and auxiliary registers are allocated at renaming. This is the case of basic register multimapping.
2. Main registers are allocated at rename, but auxiliary registers are allocated at writeback (*D1*).
3. Both main registers and auxiliary registers are allocated at writeback (*D2*).

Figure 12 shows how ERM works and contrasts it with LB and BRM. In BRM, both the physical registers (main and auxiliary) are allocated at the rename stage. For LB, the allocation of registers is delayed to writeback stage. For *D1*, the auxiliary register is allocated at writeback while the main register is allocated during renaming. For *D2*, both main and auxiliary registers are allocated at writeback.

Although the combination of two techniques makes the number of written auxiliary registers higher, write conflicts can still occur. We assume *write aggressive* for ERM.

3.2.3 Architecture and policy changes

To support ERM, modifications are made to the MT. The main register index R_M in the MT is now used as the virtual register, and a new field R'_M is added to store information about the index of the main physical register. A new bit, I_V , is also added to the MT to determine if a main physical register has been mapped for the virtual register. R'_M is valid only when I_V is set (note that the I_A bit and other auxiliary

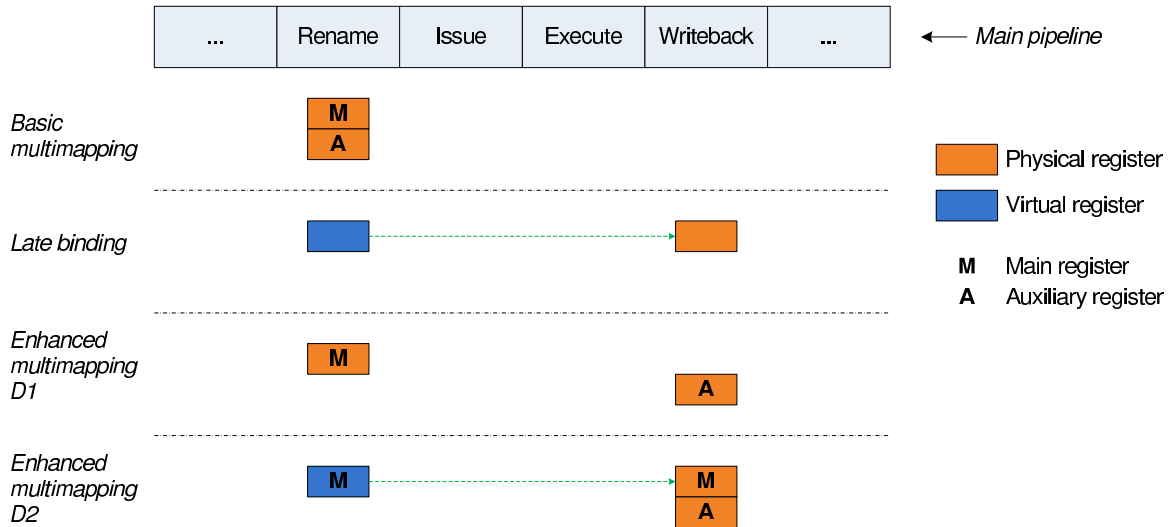


Figure 12. Enhanced multimapping as compared to late binding and basic multimapping.

```

enhanced_allocate()
  if (the freelist is not empty) then
    Get the register index from the freelist to  $R_M$ ;
    Unset  $S_M$ ;
    The main register index is  $R_M$ ;
  else if (found an allocated auxiliary register  $R'_A$ ) then
    // Deallocate the auxiliary register
    Unset  $I_A$  of the entry which contains  $R'_A$ ;
    The main register index is  $R'_A$ ;
  else
    Stall the rename stage;
  end if
end

```

Figure 13. Enhanced mapping registers.

register related fields are not changed). I_V is set during writeback, when the physical register is mapped to a virtual register, and is unset when freeing a register. The auxiliary register index R_A is used as normal and is allocated at writeback stage for both $D1$ and $D2$.

The allocation policy is simpler than for BRM, as an auxiliary register is not allocated at renaming. The procedure for allocation is depicted in Figure 13. In policy $D1$, R_M is used as both a virtual register and physical register, and in $D2$, R_M is the virtual register, but the allocation in Figure 13 can be used for both policies.

As *writeback conservative* increases write conflicts, we only study writeback aggressive for ERM. The

```

write_result( $R_M$ )
  if (not  $S_M$ ) then
    Allocate a physical register  $R'_M$  for the virtual register  $R_M$ ;
    Write to the main register  $R'_M$ ;
    Set  $S_M$ ;
  end if

  if ( $I_A$  and not  $S_A$ ) then
    Allocate an auxiliary register  $R_A$ ;
    if (found a write port for  $R_A$ ) then
      Write to the auxiliary register  $R_A$ ;
      Set  $S_A$ ;
    end if
  end if
end

```

Figure 14. Enhanced writing results back.

writing procedure of ERM is depicted in Figure 14. The allocation of physical registers R'_M and R_A can be done just before writeback, as they are done by finding physical registers in the freelist with free write ports only.

The enhanced *writeback aggressive* remains unchanged compared to basic *writeback aggressive* in the previous section. The only change for reads is that the read of main registers refers to R'_M as physical register index. This is done just before writeback.

3.3 Register Multimapping Parameters

RM operates by exploiting free resources (free physical registers and free write ports, as described in Section 2.2). Changing the number of physical registers and write ports will impact the effectiveness of RM. Both the number of used registers as well as the number of used read/write ports depend on the applications. Also, they both correlate to IPC. The higher the IPC, the higher the utilization of these resources. Hence, in this work, we studied these factors that impact the effectiveness of RM:

1. *IPC*. High IPC means that the number of issued and committed instructions per cycle is high. So the number of register file accesses for writes per cycle is also high. This can reduce write port availability for writing results to auxiliary registers. Because aggressive writeback deallocates auxiliary registers if a free write port is not found, the number of auxiliary registers that hold a value to support multimapping is also reduced, hence reducing the effectiveness of multimapping technique.

2. *The number of physical registers.* The higher the number of physical registers, the higher the probability of finding a free register to map as an auxiliary register is high. We studied the effectiveness of RM as the number of registers varies.
3. *The number of write ports.* While we originally assumed this number to be equal to the writeback width of the processor, we also studied a larger number (e.g., twice the writeback width) to determine the best choice.

4 Methodology

We employed the M5 simulator [4] to study RM. M5 is a detailed execution-driven simulator that allows modeling of detailed out-of-order superscalar processors as well as SMT processors. The simulator was extensively modified to model the banked register file (including integer and floating point register files) and delays due to bank conflicts. Further modification was required to model basic and enhanced multimapping.

We denote the organization of a banked register file by $\langle B, R, W \rangle$, where B is the number of banks, R is the number of read ports per bank, and W is the number of write ports per bank. E.g., a centralized register file will ideally have the bank organization $\langle 1, 8, 4 \rangle$ in a 4-way superscalar processor.

We used CACTI [22] to evaluate area, latency, and power consumption of different register file organizations.

4.1 Assumptions and Simulator Modifications

A new instance was added into M5 to simulate banks of the register file. This instance models different parameters of the banks, including the number of banks, the number of ports per bank, and the current busy/free ports. We are interested in experiments with homogeneous register file banks.

Ready instructions in the instruction queue compete for source operands and functional units. If a functional unit is available for an instruction, the instruction is issued into the functional unit. As we allow partial read, the instructions will hold the assigned functional units, and their available sources are latched at the input of the functional units until all sources are available. The functional units are then allowed to start executing. We employed oldest first policy, which means that an instruction which is delayed

Table 2. Baseline processor configurations of superscalar simulation.

Parameter	Value
Processor Width	4
Instruction Queue	32
Reorder Buffer	80
Number of integer physical registers	80, 96 and 112
Number of floating point physical registers	80, 96 and 112
Register access time	1 cycle
Instruction cache	32KB, 1 cycle, 64B/block
L1 Data cache	64KB, 1 cycle, 64B/block
L2 cache	2MB, 10 cycles, 64B/block
Branch Predictor	Tournament
BTB	4096

because no free read ports were found in the previous cycle will have higher priority over other younger instructions. This helps avoid deadlock when an issued instruction can never be executed.

After being executed, finished instructions compete for free write ports. If a free write port is found for a finished instruction, the result is written back, dependent instructions are woken up, and the instruction is allowed to commit. Otherwise, the instruction still holds the functional unit, and is delayed and competes for a free write port in the next cycle. We also employed oldest first policy for the pool of finished instructions.

Another instance was added to model the MT, and different policies discussed in Section 3 were implemented into the simulator.

4.2 Superscalar Simulation

The baseline processor organization that we modeled is similar to Alpha 21264. The target processor is superscalar with an issue width of 4. Various parameters are listed in Table 2. Note that the number of registers is variable only when evaluating the impact of number of registers on the technique; otherwise, it is 96 registers.

We evaluated our technique using a subset of the SPEC200 benchmark suite [9]. SPEC2000 benchmarks include 7 integer applications and 5 floating point applications, as listed in Table 3 [18]. Benchmarks were compiled using -O3 flag. Each SPEC2000 benchmark was run for about 500 million instructions after skipping initializations (a minimum of 100 million instructions).

In the case with no RM support, we collected simulation results for register utilization, read and write

Table 3. Description of SPEC2000 applications.

Application	Type	Description
mcf	Integer	Combinatorial optimization
eon	Integer	Computer visualization
ammp	Floating point	Computational chemistry
applu	Floating point	Parabolic / Elliptic partial differential equations
apsi	Floating point	Meteorology: Pollutant distribution
bzip2	Integer	Compression
twolf	Integer	Place and route simulator
vortex2	Integer	Object-oriented database
wupwise	Floating point	Physics / Quantum chromodynamics
parser	Integer	Word processing
swim	Floating point	Shallow water modeling
vpr	Integer	FPGA circuit placement and routing

port utilization, fraction of read and write port conflicts, cache miss rates (icache and dcache), fraction of register reads from the register file or bypass logic (including unique registers used by multiple operands), register lifetimes, IPCs, etc., to analyze and compare the behavior of processors under different experiments.

With multimapping support, we collected simulation results for register utilization (including the number of allocated auxiliary registers), fraction of register reads from main registers or auxiliary registers, fraction of a main register found from the freelist, the allocated auxiliary register pool, fraction of an auxiliary register found or not found, fraction of read and write port conflicts, IPCs, etc.

We then compared execution performance, read and write conflicts, etc., of different multimapping policies to the baseline case to evaluate the effectiveness of RM.

We ran experiments with different bank organizations and different registers to study their impact on performance of the whole system.

4.3 SMT Simulation

The parameters for the baseline processor that we used for SMT studies are shown in Table 4. SMT policies we employed in our experiments are ICOUNT for fetching [24] and round robin for load/store queue, instruction queue, reorder buffer, and committing.

We ran experiments for 2-threaded applications from SPEC2000. Each application was run until every workload reaches at least 500 million instructions after skipping initializations. Workloads were constructed using a sliding window methodology. We studied 17 different combinations of SPEC2000 benchmarks from

Table 4. Baseline processor configurations of SMT simulation.

Parameter	Value
Processor Width	4
Instruction Queue	32
Reorder Buffer	80
Number of integer physical registers	128
Number of floating point physical registers	128
Register access time	1 cycle
Instruction cache	32KB, 1 cycle, 64B/block
L1 Data cache	64KB, 1 cycle, 64B/block
L2 cache	2MB, 10 cycles, 64B/block
Branch Predictor	Tournament
BTB	4096

Table 3.

We employed the methodology proposed in [20, 14] to calculate weighted IPCs. The SMT speedup in [20] can be expressed as

$$S = \sum_j \frac{IPC_{SMTj}}{IPC_{Sj}}$$

where IPC_{SMTj} is the IPC of thread j when in SMT execution, and IPC_{Sj} is the IPC of thread j when in single-threaded execution.

To make it easy for explanation, we define the $IPC_{T,M,P}$ as a quantity of 3-tuple:

- T: Relative (R) or absolute (A).
- M: Single-threaded execution (SS) or SMT execution (SMT).
- P: Register multimapping policy. The policies include ideal (I), no support (N), and support policies (WC , WA , $D1$, $D2$) as discussed in Section 3.

We computed the relative IPC of multimapping policy P in SMT execution by

$$IPC_{R,SMT,P} = \frac{IPC_{A,SMT,P}}{IPC_{A,SMT,I}}$$

Because $IPC_{A,SMT,P} = IPC_{A,SS,P} \times S_P$, where S_P is the SMT-speedup of policy P as mentioned before, we have

$$IPC_{R,SMT,P} = \frac{IPC_{A,SS,P} \times S_P}{IPC_{A,SS,I} \times S_I} = IPC_{R,SS,P} \times \frac{S_P}{S_I}$$

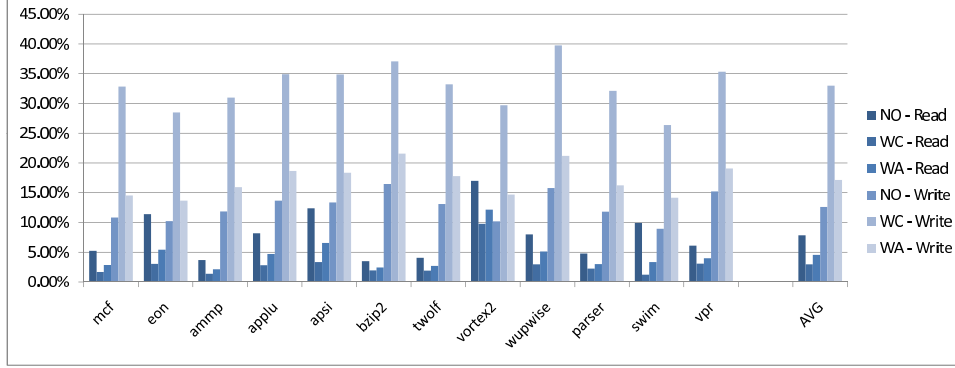


Figure 15. Percentage of register read and register write attempts that fail due to bank conflicts for $\langle 4, 2, 1 \rangle$ organization for basic register multimapping.

$IPC_{R,SMT,P}$ is used to evaluate the effectiveness of register multimapping policy P on SMT execution mode. The ratio $\frac{S_P}{S_I}$ can also be used to compare the effectiveness of multimapping policy P on SMT execution over single-threaded execution.

5 Results and Analysis

For both superscalar simulation as well as SMT simulation, we present our results for $\langle 1, 8, 4 \rangle$, $\langle 4, 2, 1 \rangle$, and $\langle 4, 1, 1 \rangle$ for SPEC2000 benchmarks. $\langle 4, 1, 2 \rangle$ and $\langle 4, 2, 2 \rangle$ are also included when comparing different port organizations.

5.1 Performance

5.1.1 Basic register multimapping

In order to evaluate the effectiveness of BRM, we consider two bank organizations: $\langle 4, 2, 1 \rangle$ and $\langle 4, 1, 1 \rangle$. The first organization has the same total number of ports as in a centralized register file ($\langle 1, 8, 4 \rangle$). In the $\langle 4, 1, 1 \rangle$ organization, the number of read ports is halved and hence the chances of read port conflicts are higher.

Figures 15 and 16 show the percentage of register read and register write attempts that fail due to port conflicts for each organization. The percentage of failed write attempts does not change much when the number of read ports changes, while failed read attempts in $\langle 4, 2, 1 \rangle$ are much smaller than in $\langle 4, 1, 1 \rangle$. Figure 17 shows the impact of each policy on the whole IPC of the system.

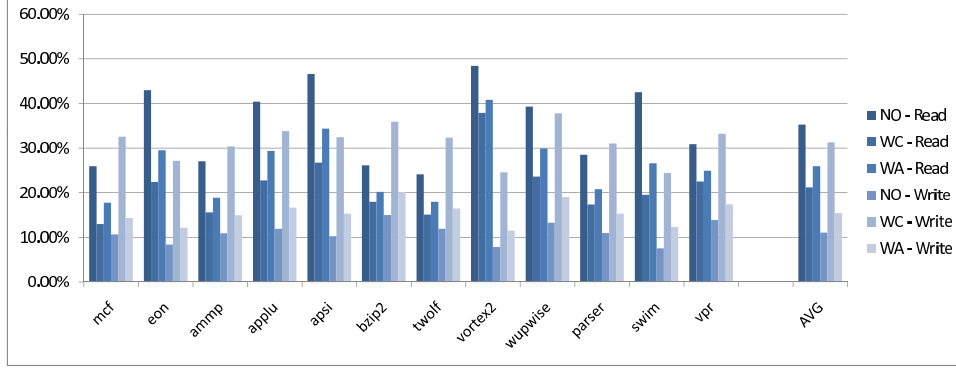


Figure 16. Percentage of register read and register write attempts that fail due to bank conflicts for $\langle 4, 1, 1 \rangle$ organization for basic register multimapping.

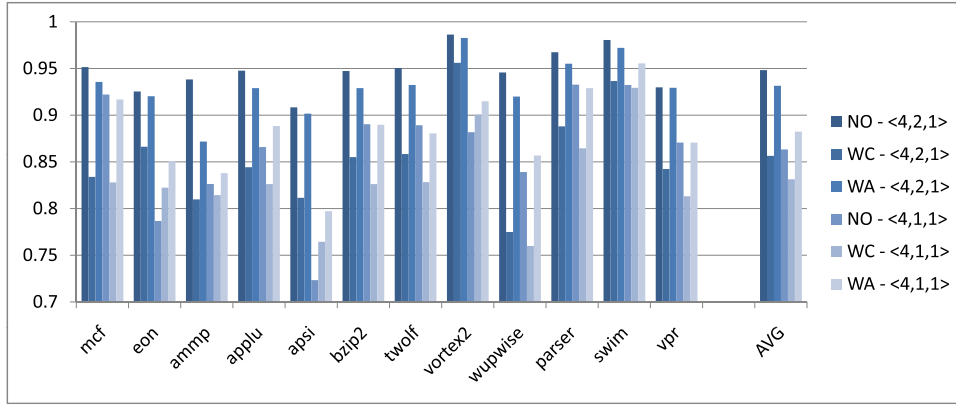


Figure 17. Normalized IPC for basic register multimapping.

For each organization, we compared the effectiveness of each write policy (*write conservative* - *WC* and *write aggressive* - *WA*) to the case without any support (*NO*). As we expected, the amount of failed read attempts decreases significantly for *WC* (about 4.9% for $\langle 4, 2, 1 \rangle$ and 14.0% for $\langle 4, 1, 1 \rangle$) when compared to the case without support. Failed write attempts, however, increase to 20.3% for both organizations. This is because a finished instruction can only commit when its auxiliary register has been written back. Because the increase of delays due to write conflict is higher than the decrease of delays due to read conflict, performance of the whole system degrades, for both $\langle 4, 2, 1 \rangle$ and $\langle 4, 1, 1 \rangle$, as shown in Figure 17 in the case of *WC*. Performance improves by 1.9%, however, for $\langle 4, 1, 1 \rangle$ in the case of *WA*, and is as high as 7.4% for *apsi* benchmark. This benefit comes from reduction in read conflicts (about 9.3%), while the increase in write conflicts is smaller (about 4.4%). For $\langle 4, 2, 1 \rangle$, read conflict reduction for *WA* is 3.3%, but write conflict increase is 4.5%. IPC degrades by 1.6%.

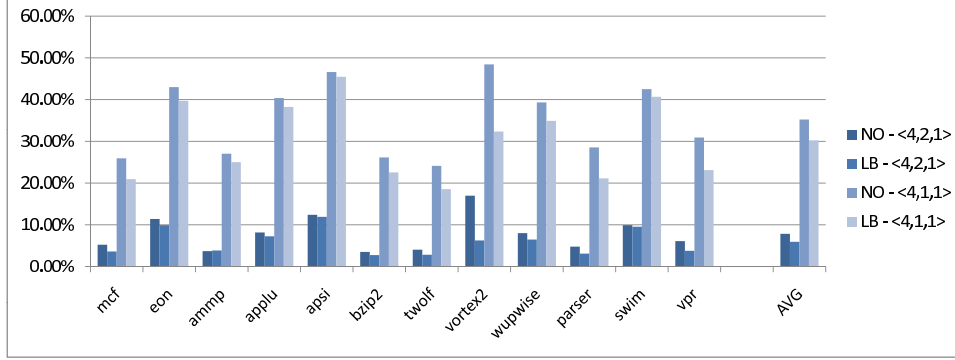


Figure 18. Percentage of register read attempts that fail when the physical register is allocated at writeback.

Significant reduction in read conflicts, but increase in write conflicts shows that register multimapping should be used in conjunction with the late allocation technique that can decrease write conflicts.

5.1.2 Late allocation of physical registers

Late allocation of physical registers effectively reduces write port conflicts. In fact, for two organizations $\langle 4, 2, 1 \rangle$ and $\langle 4, 1, 1 \rangle$, there are no instructions delayed due to write conflict for 4-way superscalar processors as the number of write ports is equal to the commit bandwidth and each instruction only produces at most one register result (and banks can be selected in the writeback stage in such a way that there are no conflicts). Figure 18 shows the effect of late allocation on read conflicts. Late allocation decreases even read conflicts slightly as the registers get more evenly distributed across banks than in the baseline case where a physical register is picked off the freelist and can belong to any arbitrary bank.

The combined effect of alleviation of write and read pressure make late allocation a very effective technique for IPC improvement. Figure 19 shows the normalized IPC for each organization. The technique reduces the IPC degradation due to banking by about 4%.

We also observed that average write port utilization when using the late allocation technique for a 4-way processor with 4-write-port register file is only about 28% and about 44% when at least one instruction is issued (this will be discussed in more detail in Section 5.3.2). This demonstrates that the remaining free write ports can be used to support register multimapping by using them to write data into auxiliary registers.

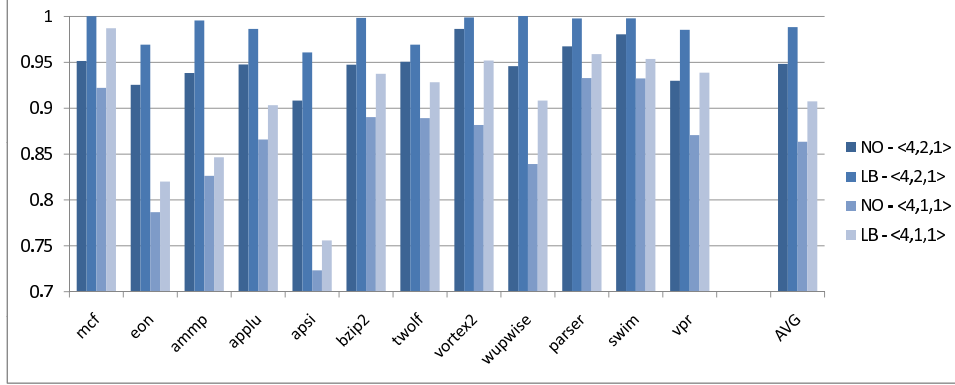


Figure 19. Normalized IPC for late allocation of physical registers.

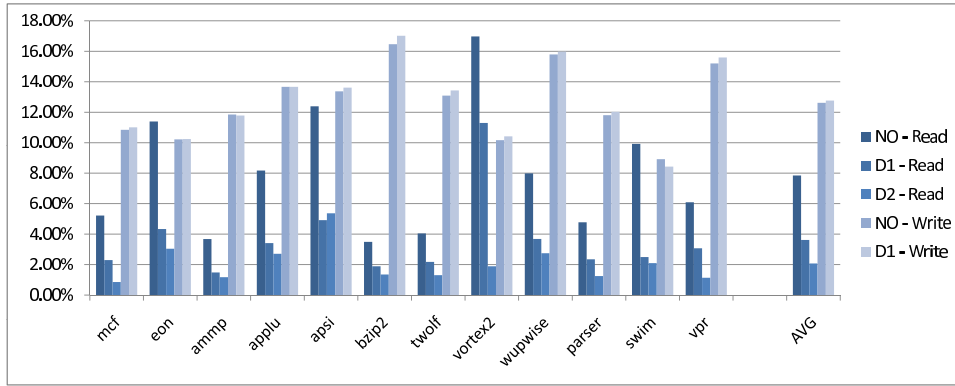


Figure 20. Percentage of read/write attempts that fail for $\langle 4, 2, 1 \rangle$ for enhanced register multimapping.

5.1.3 Enhanced register multimapping

Figures 20 and 21 show the percentage of failed attempts due to read and write port conflict for each register organization when the physical registers are allocated in the writeback stage. *D1* represents the case where only the auxiliary register is allocated in the writeback stage, whereas the main register is allocated at the rename stage. *D2* represents the case where both main and auxiliary registers are allocated in the writeback stage.

In this section, we consider only the *write aggressive* policy. Note that there are no delayed instructions due to writes for *D2* for a *write aggressive* policy as the number of write ports is equal to the commit bandwidth. Using a *write conservative* policy can result in write conflicts (though it can still result in higher performance improvement due to a greater reduction in read conflicts).

Our results show that *D1* does not significantly improve performance for the $\langle 4, 2, 1 \rangle$ organization. This is because the number of write port conflicts is not reduced. For the $\langle 4, 1, 1 \rangle$ organization, however, *D1*

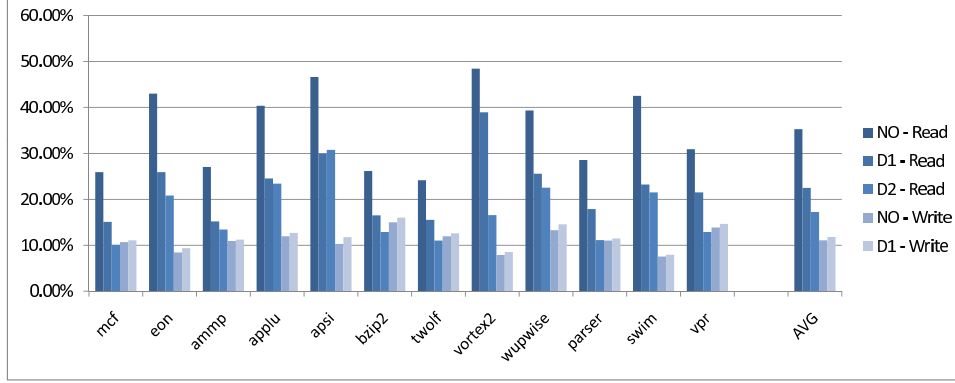


Figure 21. Percentage of read/write attempts that fail for delays for $\langle 4, 1, 1 \rangle$ for enhanced register multimapping.

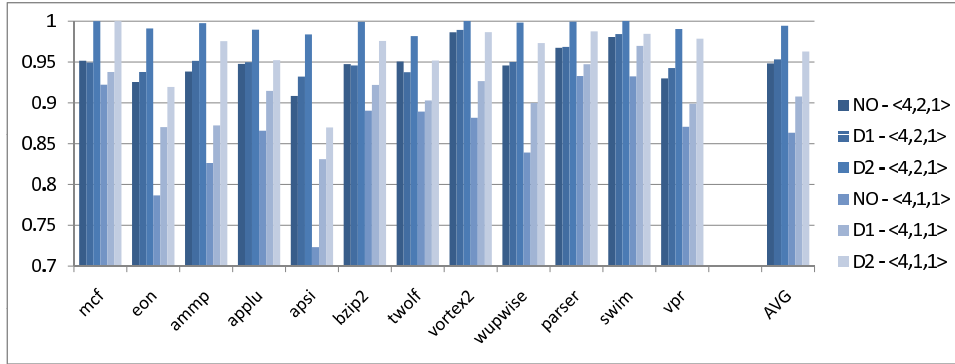


Figure 22. Normalized IPC for enhanced multimapping.

decreases the number of read conflicts significantly (about 12.7%) while the write conflicts increase only modestly (0.8%). Therefore, we see an average IPC improvement of 4.4% (Figure 22). Improvements of up to 10.7% are seen.

For the best combination $D2$ (allocation of both registers is delayed until writeback), writes into auxiliary registers are done using the free write bandwidth (as there are no write conflicts for the WA policy). IPC increases significantly as a result for both register file organizations. Performance improvements are up to 7.5% for $\langle 4, 2, 1 \rangle$ and up to 14.9% for $\langle 4, 1, 1 \rangle$. Average improvements are 4.6% and 10.0%, respectively. Comparing $D1$ and $D2$, we also see a significant reduction in read conflicts, because registers are distributed better over banks for $D2$ than $D1$.

Another way to interpret these results is that IPC degradation due to banking can be reduced to only 0.6% for $\langle 4, 2, 1 \rangle$, and only about 3.8% for $\langle 4, 1, 1 \rangle$ by an application of enhanced multimapping.

Comparing enhanced multimapping against basic multimapping, enhanced multimapping outperforms basic multimapping by 8% and 6.3% for $\langle 4, 1, 1 \rangle$ and $\langle 4, 2, 1 \rangle$, respectively. Enhancements are up to 13.8%

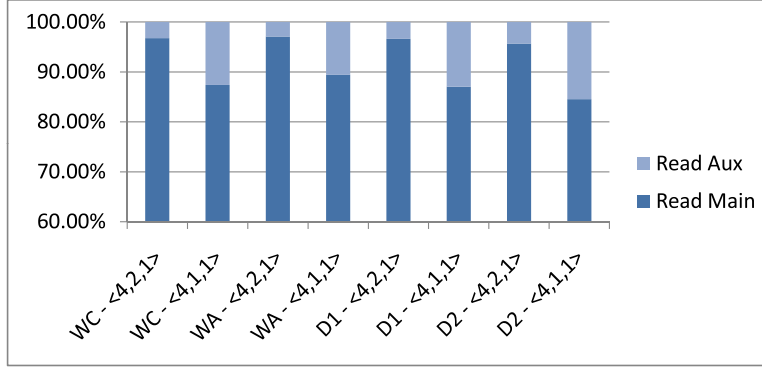


Figure 23. Breakdowns of register reads.

and 12.3%, respectively. This shows that there can be considerable value to combining multimapping with late allocation of physical registers. Comparing enhanced multimapping to late allocation of physical registers, an average performance improvement of 5.5% is seen for $\langle 4, 1, 1 \rangle$. Individual improvements are up to 12.9%. This shows that even the effectiveness of late allocation of physical registers can increase significantly due to register multimapping. Improvements are insignificant for $\langle 4, 2, 1 \rangle$ as late allocation reclaims all the performance degradation due to banking.

Note that register multimapping reduces pressure on read/write ports of the register file, so it effectively also dampens the negative performance effects of reducing the number of ports. This may allow the designer to save significant area and power by designing register files with fewer ports without much degradation in performance. Our results in Figure 22 show that we can halve the number of read ports ($\langle 4, 2, 1 \rangle$ to $\langle 4, 1, 1 \rangle$) without losing more than 3% in performance. Using CACTI, we found that halving the number of ports will result in an area savings of 46% for 65-nm technology while energy per access will go down by 48%.

5.2 Effectiveness of Register Multimapping

5.2.1 Breakdown of register reads

Benefits of RM come from alternative reads of auxiliary registers when there are no free read ports for main registers. A high fraction of reads from auxiliary registers means an effective RM. Figure 23 shows the breakdown of reads for each register type.

Results show that RM in $\langle 4, 1, 1 \rangle$ performs better than $\langle 4, 2, 1 \rangle$. This is because the incidence of read

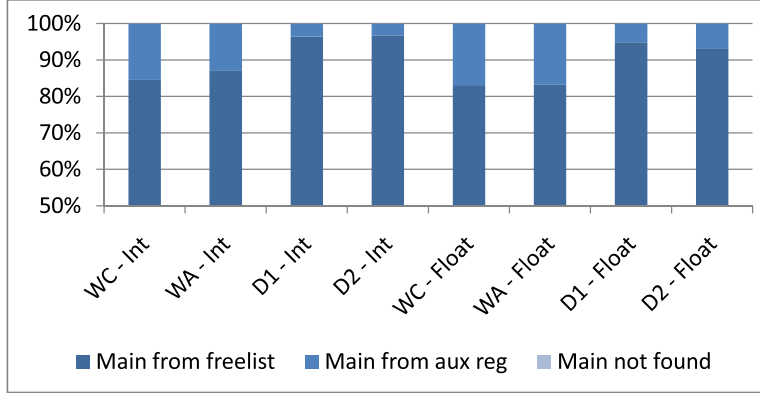


Figure 24. Fraction of main register allocation sources.

port conflicts is higher in $\langle 4, 1, 1 \rangle$. An average of 13% are reads from auxiliary registers for $\langle 4, 1, 1 \rangle$, and about 3% for $\langle 4, 2, 1 \rangle$.

In $\langle 4, 1, 1 \rangle$, *WC* has a high percentage of auxiliary register reads, as the number of auxiliary registers is high. *D1* shows benefit similar to *WC*, but it has a smaller number of write conflicts. For *D2*, the fraction of auxiliary register reads is highest. A similar trend can also be seen for $\langle 4, 2, 1 \rangle$.

5.2.2 Breakdown of main register allocation

As discussed previously, when a main register is about to be allocated, the freelist is looked up first, and if the freelist is empty, a current auxiliary register will be deallocated and then allocated as the main register to avoid renaming stall. The smaller the number of auxiliary registers deallocated for this reason, the better RM will do. This is reflected in Figure 24, which shows the breakdown of integer and floating point main register sources for $\langle 4, 1, 1 \rangle$ (note that results for the floating point registers are collected from floating point applications only). Because the fraction of stalling time when there are no free registers and no auxiliary registers is very small (less than 0.1%), the sources of main registers come from the freelist and allocated auxiliary registers. From the figure, ERM is seen to be more effective than BRM, as most of the time the main register is obtained from the freelist (about 97% of the time). This introduces an optimization opportunity for allowing stalls when the freelist is empty. In this case, a small performance degradation can be seen, but the additional logic to deallocate auxiliary registers at renaming is not required.

The reason why ERM is more effective than BRM in terms of main register allocation is because the lifetime of physical registers is smaller. By delaying the allocation of auxiliary registers until writeback for

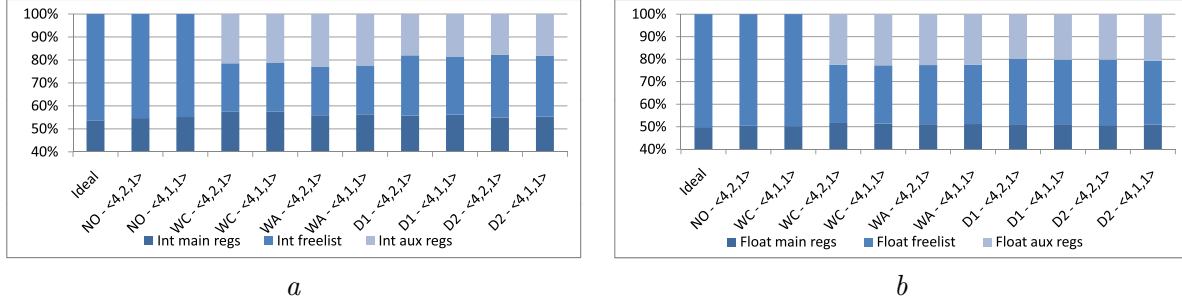


Figure 25. Physical register utilization.

$D1$ and both auxiliary and main registers for $D2$, the average number of free registers (or freelist size) is higher. This helps reduce the consumed energy not only for LB, but also for RM.

5.3 Resource Utilization

In this section, we investigate the effect of RM on the utilization of registers and write ports.

5.3.1 Register Utilization

Figure 25 shows the average number (in terms of percentage) of physical registers as main registers, the number of registers allocated as auxiliary registers, and the number of “unused” registers. Results show that, in the case without any support, register utilization is relatively small (about 54% for integer registers and 50% for floating point registers). The unused registers can be used in RM.

RM increases the number of used registers, as an average of about 20% of registers are used as auxiliary registers, and the number of main registers is slightly increased. In fact, the number of registers used as main registers and auxiliary registers does not change much over different policies for both basic and enhanced multimapping. This is because this number depends mainly on the allocation and deallocation of main registers, which are hardly impacted by the RM technique. This emphasizes one of the goals of RM: using free resources to improve performance, while not impacting the main behavior.

Compared to BRM, ERM has fewer average auxiliary registers, while it performs better. This is because the lifetime of registers is smaller.

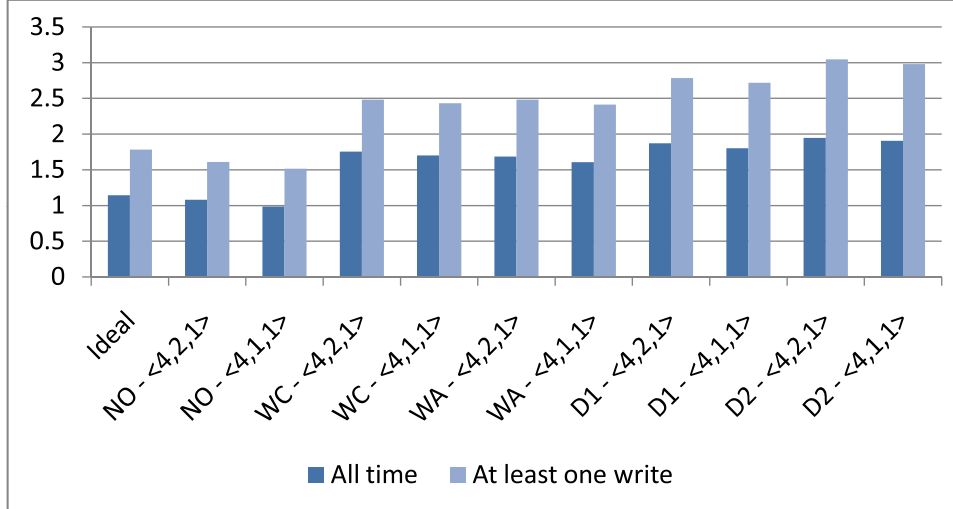


Figure 26. Average number of write ports used.

5.3.2 Write port utilization

Another resource exploited for RM is free write ports. As mentioned before, the use of write ports without RM is small. Figure 26 shows that for a physical register file with 4 write ports in a 4-way processor, the average number of used write ports is small: 1.14 for all execution time, and 1.78 when at least one write port is busy during the execution, while the register file has 4 write ports. We exploit the remaining free write ports to write results to auxiliary registers.

Among policies, *D2* has the best utilization of write ports. Write ports utilization is as high as 75% when at least one write port is busy. It is slightly lower for *D1* (67%). *WC* and *WA* have similar utilization (about 62%).

While write port utilization is high when at least one write is performed, utilization is only about 50% over the entire execution for ERM when zero writes are counted. Zero writes happen at exceptions such as branch misprediction or misspeculation, when no write ports are used. This introduces an opportunity for optimization when write ports can be exploited during zero-write period. Possible optimizations include using a buffer for writeback of result, or allocating auxiliary registers while handling exceptions. We leave this for future work.

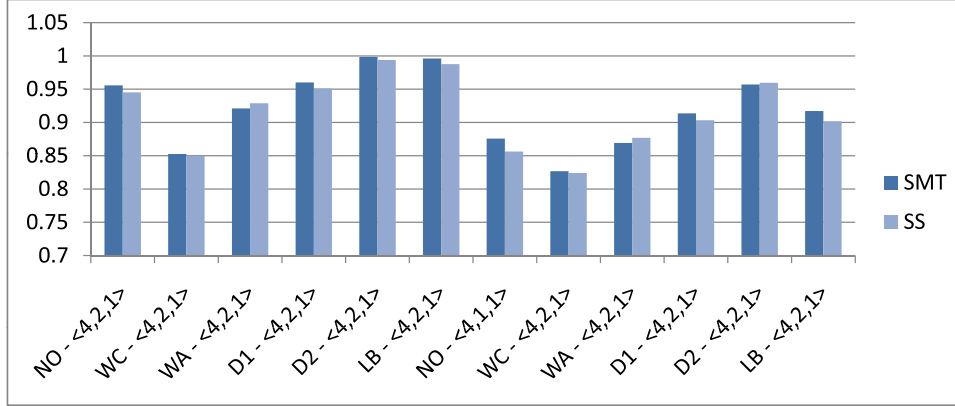


Figure 27. Normalized IPC of SMT execution compared to superscalar execution (SS).

5.4 Register Multimapping for SMT Execution

For SMT execution presented in this section, average IPC is about 1.7 (average of single-threaded executions is 1.35). Figure 27 shows relative IPC for 2-workload execution in the comparison with single-threaded execution. Results reveal that SMT execution is more sensitive with banked register file. It is more obvious for $\langle 4, 1, 1 \rangle$, when performance degradations of all cases are all higher in single-threaded execution. However, the good news is that the amount of degradation is smaller for *D2* (0.3%) when compared to the case without support (2%). This means that the penalty caused by register bank conflicts is alleviated better for SMT execution. For $\langle 4, 2, 1 \rangle$, however, we see a better performance of *D2* for SMT execution. This comes from the fact that resources are exploited better in SMT execution, while penalty due to high IPC has not impacted performance of *D2*.

In this section, the benefit of RM in SMT execution is compared to processors with 1 cycle access time centralized register files. This may not be a fair comparison because in SMT processors the number of physical registers is high, and access time may be as high as 2 cycles [25]. Meanwhile, the register file, as a critical storage of temporary data, requires high speed. This will further make RM a good choice, when the access time is reduced to just one cycle, when the register file is organized into banks, which has less complexity. Although we assume a centralized register file with single cycle access time, we still see the benefit of RM for SMT processors.

From the above observation, one can recognize the correlation between IPC and performance of register multimapping. In the later section, we will discuss more about this correlation.

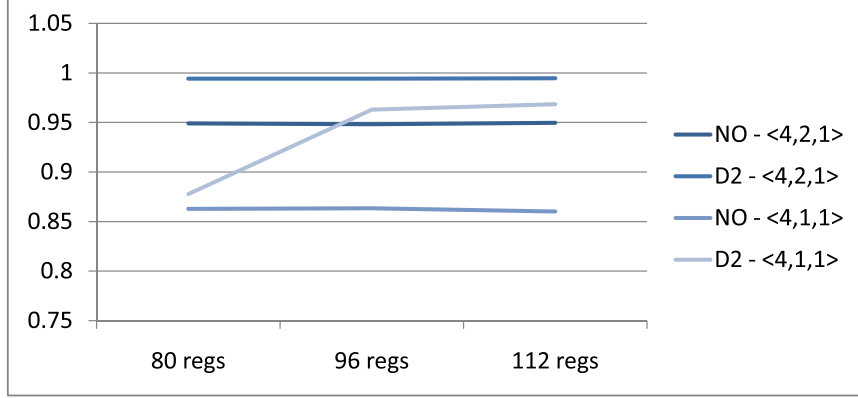


Figure 28. Register multimapping for different number of physical registers.

5.5 Impact of Parameters

We now discuss how various parameters impact the effectiveness of RM.

5.5.1 Impact of number of registers

Figure 28 shows the relationship between the effectiveness of RM ($D2$) and the number of registers. In previous sections, we assumed 96 integer registers and 96 floating point registers for single-threaded execution. When the number of registers of each type is increased by 16 to 112, performance change is relatively small. For $D2$ in $\langle 4, 1, 1 \rangle$, improvement in performance is only 0.6%. Performance degradation when the number of physical registers is reduced to 80 is as high as 8.5%. This is due to a lack of physical registers to support register multimapping. The figure suggests the choice of $\langle 4, 1, 1 \rangle$ organization with 96 physical registers, and multimapping support is attractive because performance degradation is modest, and the register file simpler than $\langle 4, 2, 1 \rangle$.

5.5.2 Impact of number of ports

Figure 29 shows the impact of port organizations in the effectiveness of RM. The two organizations $\langle 4, 2, 1 \rangle$ and $\langle 4, 1, 1 \rangle$ have been studied and compared in previous sections. In this section, two more organizations, $\langle 4, 1, 1 \rangle$ and $\langle 4, 2, 2 \rangle$, are studied for comparison.

We see that a change in the number of write ports from 1 to 2 for each bank does not have much impact on $D2$ if the number of read ports is kept the same. For example, $\langle 4, 1, 1 \rangle$ and $\langle 4, 1, 2 \rangle$ have the same performance, and $\langle 4, 2, 1 \rangle$ and $\langle 4, 1, 2 \rangle$ have the same performance (while we still can see performance

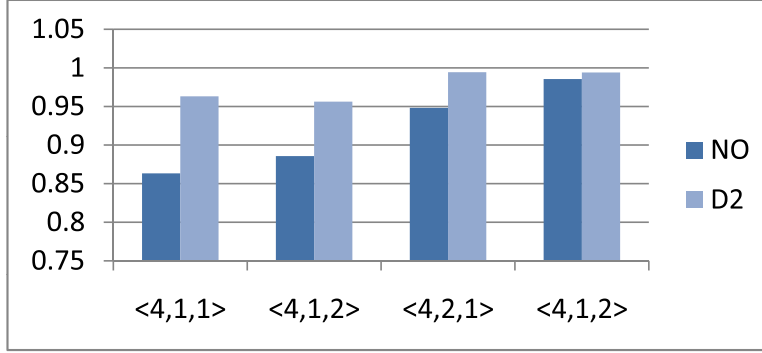


Figure 29. Register multimapping performance for different port organizations.

change for the case with no support). This means that 4 write ports are enough in this scenario, and they are not in the critical path when considering the performance of RM. $\langle 4, 1, 1 \rangle$ organization is an attractive choice as performance degradation is small compared to $\langle 4, 2, 1 \rangle$ (about 3%).

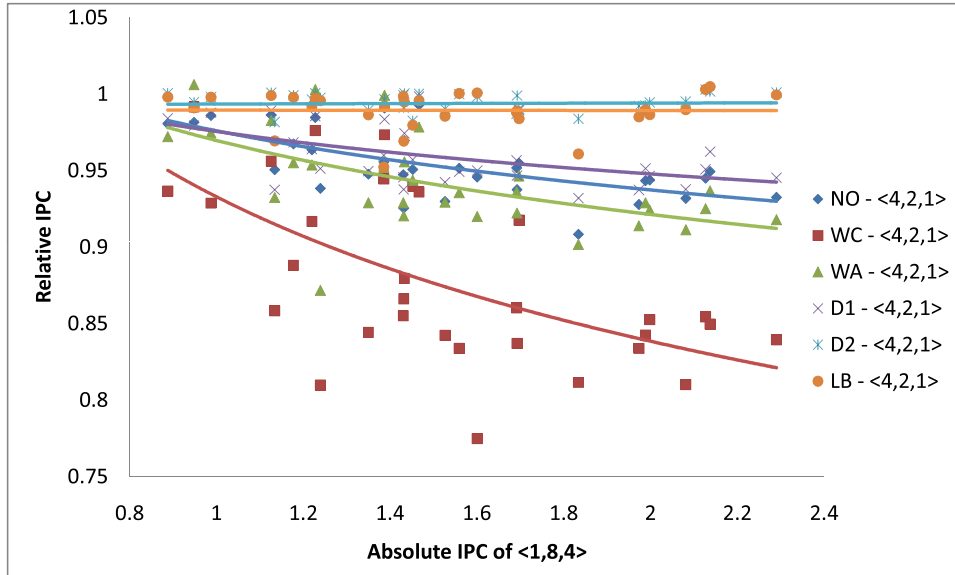
5.5.3 Correlation with IPC

The final set of results that we present is the correlation between IPC and performance of RM. Figure 30 shows this correlation for each port organization. To produce these graphs, we combined all results from single-threaded executions and 2-threaded executions.

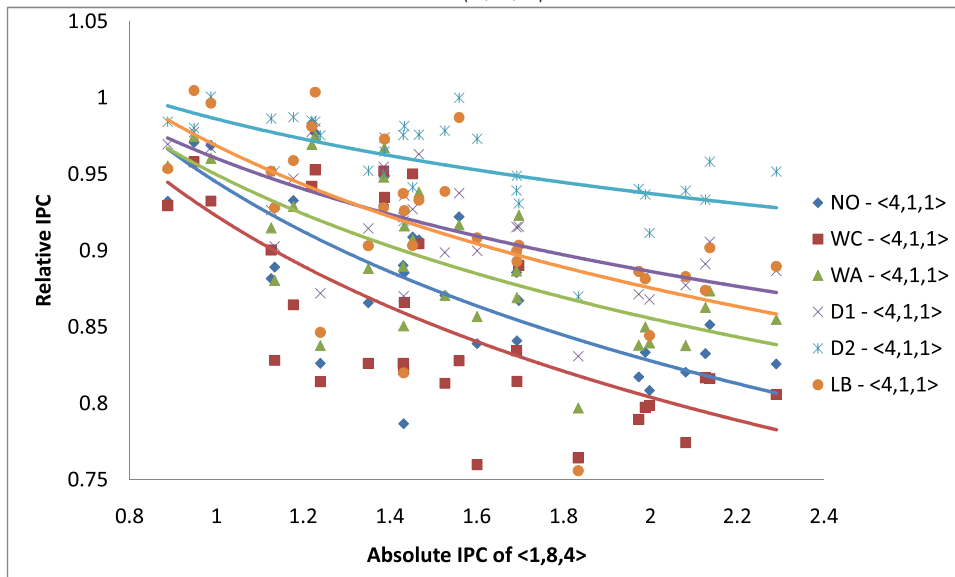
There are two things to note in these graphs:

1. Relative IPCs degrade for all policies in $\langle 4, 1, 1 \rangle$ and many configurations in $\langle 4, 2, 1 \rangle$ with increasing absolute IPCs.
2. Performance increases for *D2* and *LB* in $\langle 4, 2, 1 \rangle$ when IPC increases to 2.4.

There are two factors that impact the performance of RM as IPC changes: (1) Probability of bank conflicts increases along with IPC; the more the conflicts, the higher the benefit of RM. (2) When IPC increases, more instructions are issued and written back, and hence there are more accesses to the register file. This reduces the number of available ports available to support RM. These two factors produce the trends we observe above. For $\langle 4, 2, 1 \rangle$, while other policies see performance degradation as IPC increases, *D2* and *LB* still enjoy performance improvements. This is because relative IPC is a ratio of a policy's absolute IPC divided by the ideal case's absolute IPC, and adding the same amount to the smaller-than-1 ratio makes the ratio increase.



a. $\langle 4, 2, 1 \rangle$



b. $\langle 4, 1, 1 \rangle$

Figure 30. Correlation between register multimapping and absolute IPC.

It can be projected that the increasing trend continues until some IPC, and then starts to degrade. At that point, read conflicts are high enough to pull down performance of the whole system. It can be projected that the relative IPCs of all policies will eventually approach some lower boundary. Compared to the case without support, *D2* and *LB* have higher boundaries, as conflict probabilities of these policies are always lower. *D2* will have a higher boundary than *LB*, as in reality IPC could never reach 4 for a 4-way superscalar, hence there are always free write ports for RM.

5.6 Summary

In this section, we show the benefits of RM over the case without support in a banked register file. Different policies of RM are also studied in terms of performance and effectiveness. In the best case, performance improvement is up to 14.9% (10% on average). We also study the reasons why RM is a benefit. We found that resource utilization when using RM is much higher than the case without supports. We compared and contrasted the difference between single-thread executions and SMT executions when applying RM. Various parameters that impact performance of RM are also studied.

6 Summary and Conclusions

Register banking is a popular technique to reduce area, power, and latency overheads of register files. However, banking results in conflicts that threaten to reduce the effectiveness and efficiency of register files.

In this paper, we discussed register multimapping as a technique to reduce register bank conflicts. Multimapping involves mapping an architectural register to multiple physical registers and then avoiding read conflicts by consuming the value available in any of the mapped physical registers. We considered both basic multimapping, which can reduce only read conflicts, and enhanced multimapping, which can reduce write conflicts as well.

Our evaluation of the two multimapping schemes showed performance improvements up to 14.9% (10% on average). Results also indicate that register multimapping can allow port reduction at minimal cost. Halving the number of read ports resulted in register area savings of 46% and register power savings of 48% for only 3.2% degradation in performance.

As area and power increasingly become zero-order design constraints, the effectiveness and applicability

of techniques like register multimapping will only increase.

References

- [1] J. Abella and A. González. On reducing register pressure and energy in multiple-banked register files. In *ICCD '03: Proceedings of the 21st International Conference on Computer Design*, page 14, 2003.
- [2] S. Balakrishnan and G. S. Sohi. Exploiting value locality in physical register files. In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 265, 2003.
- [3] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *MICRO 34: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 237–248, 2001.
- [4] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, July-Aug. 2006.
- [5] J.-L. Cruz, A. González, M. Valero, and N. P. Topham. Multiple-banked register file architectures. *SIGARCH Comput. Archit. News*, 28(2):316–325, 2000.
- [6] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *MICRO 30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 149–159, 1997.
- [7] J. A. Fisher. Very long instruction word architectures and the ELI-512. In *ISCA '83: Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 140–150, 1983.
- [8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 4 edition, 2006.
- [9] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.
- [10] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.
- [11] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.
- [12] N. S. Kim and T. Mudge. Reducing register ports using delayed write-back queues and operand pre-fetch. In *ICS '03: Proceedings of the 17th Annual International Conference on Supercomputing*, pages 172–182, 2003.
- [13] T. Monreal, V. Vinals, J. Gonzalez, A. Gonzalez, and M. Valero. Late allocation and early release of physical registers. *IEEE Trans. Comput.*, 53(10):1244–1259, 2004.
- [14] D. W. Oehmke, N. L. Binkert, T. Mudge, and S. K. Reinhardt. How to fake 1000 registers. In *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 7–18, 2005.
- [15] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *SIGARCH Comput. Archit. News*, 25(2):206–218, 1997.
- [16] I. Park, M. D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *MICRO 35: Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 171–182, 2002.
- [17] S. Park, A. Shrivastava, N. Dutt, A. Nicolau, Y. Paek, and E. Earlie. Bypass aware instruction scheduling for register file power reduction. *SIGPLAN Not.*, 41(7):173–181, 2006.
- [18] S. Sair and M. Charney. Memory behavior of the SPEC2000 benchmark suite. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, (undated).
- [19] R. Sangireddy. Register port complexity reduction in wide-issue processors with selective instruction execution. *Microprocess. Microsyst.*, 31(1):51–62, 2007.
- [20] Y. Sazeides and T. Juan. How to compare the performance of two SMT microarchitectures. In *IEEE International Symposium on ISPASS*, pages 180–183, 2001.
- [21] E. J. Tan and W. B. Heinzelman. DSP architectures: Past, present and future. *SIGARCH Comput. Archit. News*, 31(3):6–19, 2003.
- [22] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, HP Laboratories, Palo Alto, CA, 2006.
- [23] J. H. Tseng and K. Asanovic. A speculative control scheme for an energy-efficient banked register file. *IEEE Trans. Comput.*, 54(6):741–751, 2005.
- [24] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. *SIGARCH Comput. Archit. News*, 24(2):191–202, 1996.
- [25] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *SIGARCH Comput. Archit. News*, 23(2):392–403, 1995.
- [26] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, page 179, 1996.
- [27] S. Wang, H. Yang, J. Hu, and S. G. Ziavras. Asymmetrically banked value-aware register files. In *ISVLSI '07: Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 363–368, 2007.
- [28] C. Zang, S. Imai, and S. Kimura. Duplicated register file design for embedded simultaneous multithreading microprocessor. In *ASICON 2005*, volume 1, pages 90–93, 24-27 Oct. 2005.
- [29] H. Zeng and K. Ghose. Register file caching for energy efficiency. In *ISLPED '06: Proceedings of the 2006 International Symposium on Low Power Electronics and Design*, pages 244–249, 2006.