

Reservation Station Architecture for Mutable Functional Unit Usage in Superscalar Processors

Yan Solihin, Kirk W. Cameron, Yong Luo, Dominique Lavenier, Maya Gokhale

Los Alamos National Laboratory

{solihin,kirk,yongl,lavenier,maya}@lanl.gov

Abstract

One major bottleneck of a superscalar processor is mismatch of instruction stream mix with functional unit configuration. Depending on the type and number of functional units, the performance loss caused by this mismatch can be significant. In this paper, we introduce mutable functional units (MFU) that enable floating point units to serve integer operations, and propose a novel architectural solution to this mismatch problem to enhance the performance of integer-intensive applications while not adversely affecting the performance of floating-point-intensive applications. Modifications to a base MIPS R1000-like architecture include the MFU, an additional reservation station dedicated to the MFU, and a steering logic. We obtain a speedup ranging from 8.3% to 14.3% for integer application, while keeping the hardware cost resulting from the architecture modification minimal (<1 % die area). In addition, our modification does not affect clock frequency of the processor and maintains binary compatibility.

1 Introduction

Superscalar microprocessor architecture attempts to exploit *instruction level parallelism (ILP)* by fetching and issuing multiple instructions every cycle. Since the issued instructions can have any combination of integer, memory, and floating point instructions, the functional unit configuration has to take that into account. Providing as many copies of each functional unit type as the issue width will provide the best ILP by avoiding any stalls due to unavailable functional units. However, this approach increases the die area occupied by the functional units, increases the complexity of the bypass network¹, and increases power consumption. Although bypass network is an important component in a superscalar architecture, it is also a source of complexity as it contains long wires. Subbarao et al. [17] pointed that the bypass delay in a bypass network (data bypass logic) grows quadratically with issue width. Since bypass network is used to forward the result values from one functional unit to another, it is more accurate to say that the bypass delay and the number of bypass network is proportional to quadratic value of the number of functional units². The paper concluded that for an 8-way superscalar processor implemented on a 0.18 μm process, the bypass network is the most important factor that limits clock frequency.

Thus, reducing the number of functional unit is desirable from hardware point of view. However, the resulting increase in instruction stalls due to unavailable functional units, which may decrease

¹ Bypass network is a network connecting functional units. It is used to forward result values of completing instructions to dependent instructions, bypassing the register files. Thus, instead of wasting cycles by checking the register files for availability of operands and reading from them, dependent instructions are issued to functional units and use the bypassed values.

² Subbarao et al. uses an assumption that the number of functional units is proportional to the issue width, hence the bypass delay and the number of bypass paths grow quadratically with issue width.

performance significantly. To solve this trade-offs, the common practice is to use a set of applications in determining which functional configuration yields good performance while maintaining the hardware complexity in an acceptable range. However, applications have different instruction stream mix. This compromise often leads to a mismatch in instruction stream mix with the functional unit configuration, causing a performance bottleneck in a superscalar processor.

A PowerPC 620 study identified this instruction mismatch as the highest contributor to the loss of *Instruction per cycle (IPC)* in the Spec92 benchmark, roughly ranging from 0.5 to 1.8 [5]. Since this type of stall is clearly determined by the functional unit configuration on a specific microprocessor implementation, we repeated this experiment on an architecture that is similar to the MIPS R10000 using the Spec95 benchmark suite. Figure 1 displays the average number of cycles that a ready instruction stalls due to unavailable functional units. The average stall for floating-point applications (swim, wave5, and su2cor) is 1.11 cycles while for integer applications (compress, ijpeg, li, and kmeans) it is 1.73 cycles. This is clearly a serious performance bottleneck.

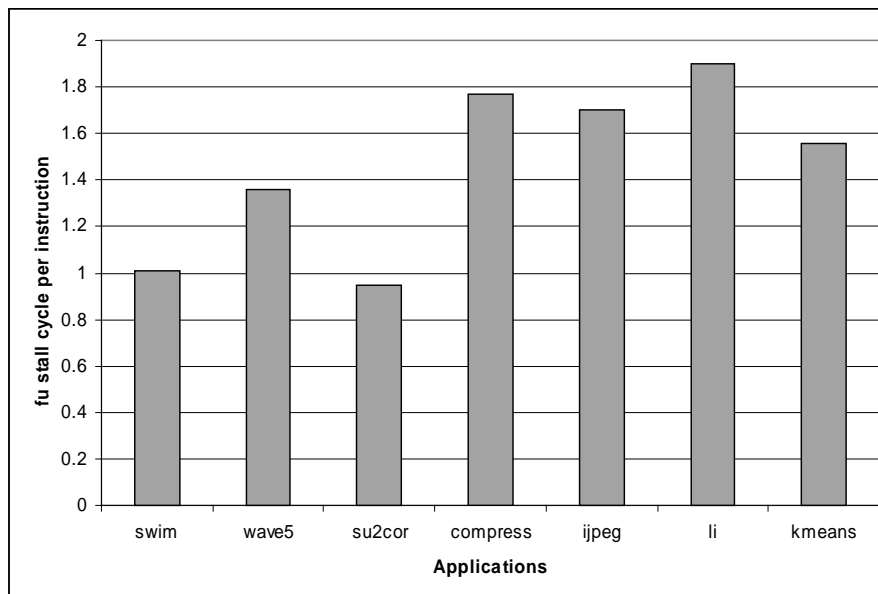


Figure 1: Average number of stall cycle per instruction due to

A more common performance measure in a superscalar processor is the degree of *instruction level parallelism (ILP)*, measured as the average number of committed instructions per cycle. Figure 2 shows the loss of IPC due to unavailable functional units ranging from 0.08 to 0.76. Note that the IPC loss for integer applications is generally higher than that of floating-point applications. This indicates that the R10000 lacks integer execution bandwidth much more than floating-point execution bandwidth.

It becomes essential to increase the execution bandwidth without increase in the number of functional units or the die area for functional units. This motivates us to design a mutable functional units, which has the capability of serving different types of instructions, requiring mutation at run time, to better match the incoming instruction stream mix.

The idea of a functional unit that can serve both integer and floating point instructions was proposed by Subbarao and Smith [18]. By augmenting integer execution capability to floating

point units, they show that the units speed up integer-intensive applications [19] by off-loading the integer instructions to the floating-point functional units. However, both papers do not

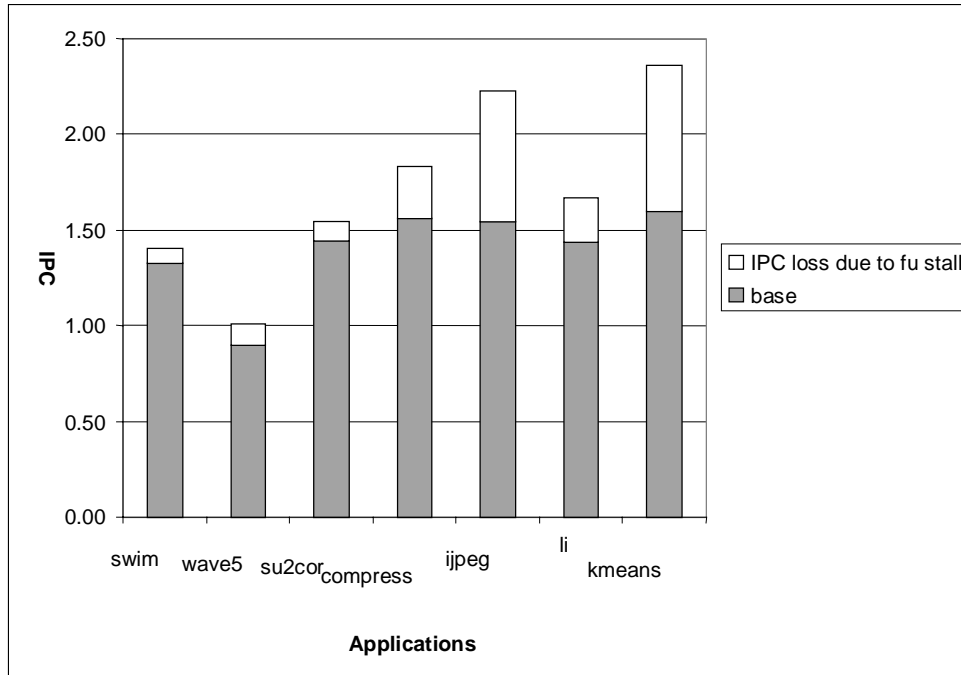


Figure 2: IPC loss due to unavailable functional units

provide details of how the floating point units are augmented with integer execution capability, the latencies, and the implications on die area.

In this paper, we present one design of an augmented floating-point unit and call it a mutable functional unit (MFU), since a switch of functionality at run time (mutation) is required to serve different types of instructions. It is based on the floating point adder of MIPS R10000 [3]. The design shows that the size of an MFU is roughly equivalent to the size of a floating point adder, the latencies of integer operations performed by the MFU are the same as the latencies that are executed by an ordinary ALU. In addition, we take into account the mutation penalty of an MFU into our simulation. To exploit the MFU, we propose novel architecture modifications to the base R10000-like architecture.

S. Subramanya et al [19] proposes an architecture and compiler scheme to exploit an integer-augmented floating-point units. Their scheme focuses on compiler transformation of source codes to identify integer instructions that can be executed by the augmented floating-point units. This is achieved through adding 22 new opcodes to tag integer instructions that are to be executed by the floating-point units. By adding extra opcodes, this approach sacrifices binary compatibility and requires code recompilation. Furthermore, it requires an additional compiler instruction analysis and partitioning.

In our approach, we take a hardware approach to detect when integer and memory instructions can be executed by the MFU with minimal hardware cost (< 1% of die area), while achieving comparable performance. Since we do not introduce new opcodes, code recompilation is not needed, although a new compiler will reduce the hardware cost further by performing analysis at compile time.

The rest of the paper is organized as follows: Section 2 describes the implementation of an MFU, alternative architectures that exploit it, and the comparison among the architectures. Section 3

describes the simulation tool, testbeds, and discuss the results. Section 4 summarizes our efforts and future directions.

2 Architectures for Mutable Functional Unit

This section begins with describing our choice of design of a mutable functional unit, and the implication on mutation penalty. Alternative architectures that exploit the MFU are discussed next. We evaluate three architectures depending on when the analysis of instruction stream is performed and when the MFU mutation is carried out: fetch profiling (FProf), reservation station monitoring (RSMon), and dispatching to and issuing from a dedicated reservation station for the MFU (RS-MFU). Advantages and drawbacks of each architecture scheme are discussed. And finally, we present a steering logic algorithm for the RS-MFU scheme.

2.1 Mutable Functional Unit

One realization of a mutable functional unit is within an embedded reconfigurable fabric using Field Programmable Gate Array (FPGA) technology. Such a technique has been proposed as a mechanism for creating application-specific hardware circuits to augment the capability of a conventional processor [6,7,11,14,15,21]. To take advantage of the reconfigurable fabric, the compiler must:

- Identify portions of code that can be accelerated within the reconfigurable fabric;
- Synthesize hardware circuits to realize those code fragments;
- Generate code to transfer state between the processor and reconfigurable fabric;
- Synchronize transfer of control between processor and reconfigurable fabric.

Although there is potential for speedup with this approach, there are also several drawbacks. Performance characterization must be done to locate likely candidates for migration to reconfigurable fabric. In order to synthesize application-specific circuits, a CAD tool chain must be invoked as part of the compilation process. Even more significantly, the reconfigurable fabric runs at a fraction of the clock rate of a modern microprocessor, so there must be considerable instruction level parallelism in the circuit to compensate for the slower clock rate. Finally, the architecture requires complex synchronization between processor and reconfigurable fabric to load configurations dynamically and to load and recover state from the reconfigurable fabric.

Our goal is to augment a superscalar processor with reconfigurability without requiring specialized compilers, large investment in custom fabrication technology, and complex synchronization between subsystems running at very different clock rates. Thus, we do not choose FPGA to implement the MFU.

We modify the R10000 floating-point adder so that it is able to additionally perform integer operations [3]. In choosing which integer operations are to be executed by the MFU, our priority is to accommodate frequently executed integer instructions. Instruction profiles of Spec95 show that integer addition, followed by integer shift and logic operations, are the most frequent integer instructions using SimpleScalar compilation. In addition, memory instructions are as frequent as integer operations. This approach identifies the relative frequency of certain instruction type over the entire code. However, we are also interested in profiling instructions within the instruction window to give indications of optimum mutation frequency and general code characteristics, to establish the MFU functionality. In [9], results of characteristic profiling are given. In particular,

we identify the clustering of instructions by distance between two consecutive instructions of the same type. The results confirm our simple frequency profiling showing integer-additions and memory operations are the most “clustered” and dominant instructions for the codes in this study. This provides further evidence that only limited modification of the original floating point adder is necessary to achieve performance improvement.

Thus, based on both profiling methods, we designed the MFU to be able to execute integer addition, shift, and logic operations, plus address generation³ for memory operations. The design requires widening the adder data path to 64 bits and adding a few switches to the floating-point adder to enable the unit to mutate into an integer adder/shifter.

Succeeding hardware design revealed that such kind of MFU costs roughly the same number of gates as a floating-point adder [3]. Another important metric of the MFU is its mutation penalty. The design also revealed the maximum mutation penalties shown in Table 1. The penalty is particularly high (2 cycles) when we switch from floating-point capability to integer addition capability. This penalty is due to the need to wait for the floating-point pipeline to drain before we are able to use it for integer addition. Thus, it is important to reduce the frequency of mutation so as to avoid such high mutation penalties.

Table 1. MFU Mutation Penalty

Mutation Category	Current Instruction	Next Instruction	Instruction After Next	Cycles
Integer to FP mutation	Logic / Add	FP-add	FP-add	0
	Shift	FP-add	FP-add	1
FP to Integer mutation	FP-ADD	Logic	Not Add	0
	FP-ADD	Logic	Add	1
	FP-ADD	Shift	All Integer	1
	FP-ADD	Add	All Integer	2

³ Address generation is basically an integer addition.

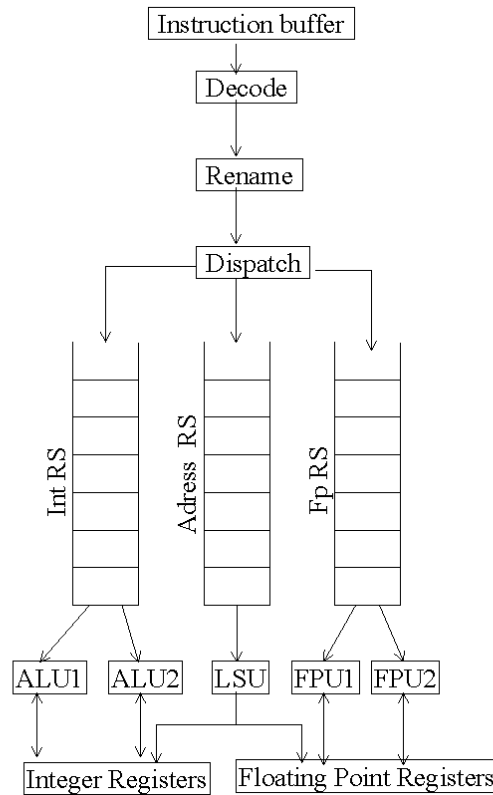


Figure 3: MIPS R10000 functional unit and reservation station configuration

2.2 Alternative Architectures

We chose to use the MIPS R10000 architecture as the basis of our study. As a reference, the architecture of MIPS R10000 is shown in Figure 3. The functional units consist of 2 integer ALU's. One is capable of performing basic operations (add/sub, logic) plus branch and shift operations, and the other is capable of performing basic plus integer multiplication and division. There is one address generation unit which is embedded into the load store unit. Finally, there are 2 floating-point units (FPUs). FPU1 is capable of performing addition, and FPU2 is capable of performing multiplication, division, and square root operations. There are three reservation stations: integer, floating point, and memory/address reservation stations. Each reservation station has 16 entries, and issues instructions in an out-of-order manner to the respective functional units.

The basic modification needed is to replace the floating point adder (FPU1 in R10000) with an MFU. Thus, the MFU is able to perform floating point addition, integer addition, logic, and shift operations. Then we design the architecture that exploits the MFU. First, we must determine the pipeline stage in which it should perform analysis of the instruction stream and the actual mutation of the MFU. Based on this, we consider three schemes as shown in Table 2.

Table 2 : Architectures based on when the analysis and mutation take place

Tasks	Fetch Profiling (FProf)	RS Monitoring (RSMon)	Dedicated RS (RS-MFU)
Analysis	Fetch	Dispatch	Dispatch
Mutation	Fetch	Dispatch	Issue

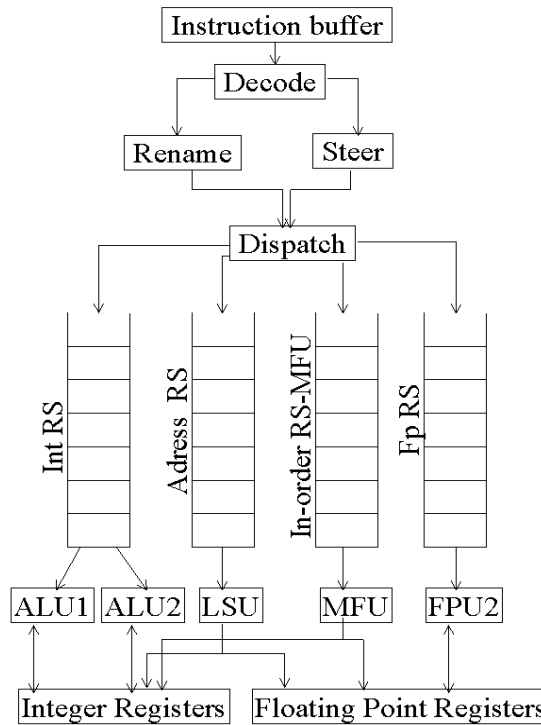


Figure 4: RS-MFU architecture modification to R10000

In fetch profiling (FProf), analysis is performed to the instructions that are fetched from the instruction cache. FProf tries to minimize the functionality change of the base architecture (but the hardware cost may not be minimized). In this scheme, first we simply replace the fixed functional unit FPU1 with MFU, which can be mutated to perform either a floating-point addition or an integer addition/shift/logic.

In Reservation Station Monitoring (RSMon), the analysis is performed by inspecting the fullness of the reservation stations. If one reservation station is full of instructions, more bandwidth is needed to service instruction types of that reservation station, thus the MFU is mutated to serve the reservation station. The architecture modification needed for this scheme is minimal, as it only needs to detect whether one reservation station becomes full, and reacts to it. However, the

drawback is high switching frequency. When the MFU is mutated to serve a reservation station, it reduces execution bandwidth of other reservation stations, which may quickly become full, necessitating a new mutation.

In both FProf and RSMon schemes, there is a possibility of starving floating point additions which reside in pipeline stages that are later than the analysis phase as the MFU is the only unit that is capable of executing floating point additions. This happens when there is a change of instruction stream from floating point operation intensive to integer operation intensive. When we detect such a change, we mutate the MFU to serve integer operations. However, it is possible that there are floating point addition operations that reside in the floating point reservation station which are waiting for execution that have to stall because the MFU is serving integer operations. Stalling these instructions in the end stall other instructions that have finished execution but cannot be committed. Thus, we add floating point addition capability to the original FPU2 (multiplier/divider) to avoid starving.

In the third scheme (RS-MFU), analysis of instructions is performed at dispatch stage right after fetched instructions are decoded for operands. After decoding the operands, register renaming is performed in parallel with a steering logic that selects a subset of instructions to be executed by the MFU. The steering logic is aware of the instruction types and infers whether an instruction can be sent to the RS-MFU. If it can be sent, the logic next decides whether or not to send to RS-MFU. The steering is performed in parallel with register renaming to avoid any effect on clock frequency. At issue stage, instructions that are sent to the RS-MFU are issued only to the MFU for execution, in the order of their arrival. If the MFU detects that the new instruction has a different type compared to the one it is serving, it performs the mutation and executes that instruction.

Architectural change needed for this scheme includes a reservation station that is dedicated to the MFU as illustrated in Figure 4. Steering logic is added to the dispatch stage to decide to which reservation station instructions are sent. All instructions steered to RS-MFU will be issued to the MFU for execution. With this scheme, the MFU mutates according to the type of instruction it will receive next.

Note that we transform the FPU1 of the R10000 into an MFU and add extra read and write ports to the integer register file. Thus, all floating-point additions have to be executed by the MFU. Since we add integer execution bandwidth and not floating-point execution bandwidth, we expect to see performance improvement for integer applications, but we have to demonstrate that the performance of floating-point applications does not suffer.

An issue arises whether the reservation station (RS-MFU) should implement in-order or out-of-order issue. The potential benefits of using an out-of-order RS-MFU include reduction of stalls due to larger instruction issue window (compared to the earliest instruction in in-order issue) and reduction of mutation frequency by prioritizing the issue of instructions of the same type.

However, we think that out-of-order issue advantages are not worth the added complexity. First, reducing mutation frequency by prioritizing instructions of the same type has a potential drawback of delaying the execution of an earlier instruction in the RS-MFU, causing all dependent instructions in other reservation stations to stall. Furthermore, reducing mutation frequency can be achieved more effectively by tuning the steering logic. Second, the advantage of reducing stalls with large instruction window can also be approximated by embedding some heuristics into the steering algorithm, and possibly by reordering instructions before entering the RS-MFU. This is beyond the scope of our paper.

Since a clear-cut advantage of using an out-of-order RS-MFU is not clear, for this paper we decided to use an in-order issue RS-MFU. However, further study is planned so as to evaluate the trade-offs of performance versus hardware complexity for an out-of-order RS-MFU.

2.3 Comparisons of the Alternative Architectures

In FProf and RSMon schemes, the hardware modifications needed are replacing FPU1 with an MFU, augmenting an adder into FPU2, and wiring all reservation stations to the MFU. The hardware increase of augmenting an adder into FPU2 seems non-trivial. It may in fact be comparable to a brute-force approach of adding extra functional units. In addition, requiring the MFU to be connected to all reservation stations to enable the MFU to receive instructions from any reservation stations is not a scalable approach.

The RS-MFU scheme has two advantages compared to FProf and RSMon schemes. First, it does not add die area to the existing functional unit area because the FPU2 is not augmented with a floating point adder. Second, it is more scalable since the MFU is connected only to one reservation station. With these advantages, it does not have the added complexity of extending wiring to the MFU and arbitration logic that selects which functional unit must execute a particular instruction.

Another potential advantage of the RS-MFU is it is more extendable. For example, instruction pre-processing in trace cache's fill unit has been suggested in [2,13,22]. The advantages of using the fill unit is that it is not in the critical path, thus, there is plenty of time for processing. Binding instructions to reservation stations (which we call instruction steering) was first suggested by [22]. With the trace cache, depending on implementation trade-offs, it is possible to implement the steering logic in the fill unit of a trace cache, instead of at the dispatch stage as suggested in this paper. However, whether implementing the steering is beneficial is beyond the scope of this paper.

In relation with the compiler, instruction steering can be implemented by the compiler by adding a tag to an instruction to indicate which reservation station the instruction must go to. This approach, however, requires code recompilation.

Finally, in relation to a VLIW architecture, an MFU gives the compiler more flexibility in bundling instructions, as the instruction slot for the MFU may contain arbitrary types of instructions. Again, this is beyond the scope of this paper.

2.4 Steering Logic Algorithm

There are many options for steering algorithms. We choose a simple algorithm that is shown in Figure 5. Basically the algorithm uses a saturating counter, Cfp, to detect whether there is a need for floating-point addition bandwidth. If there is a need, indicated by a positive value of Cfp, then the steering logic only dispatches floating-point addition operations into the RS-MFU. If there is no such need (Cfp = 0), the steering logic starts to dispatch integer and memory operations into the RS-MFU in a n-chunk round-robin manner, which is basically a round-robin with a granularity of n instructions. For example, a 4-instruction-chunk round robin waits until there are 12 (3x4) integer or memory operations sent to other functional units (ALU1, ALU2, LSU) before the RS-MFU can take 4 integer or memory operations. The counter Cfp is controlled by 2 parameters: Cfp_increment, and Cfp_max. For the n-chunk round robin scheduling, a counter Crr is used, which is controlled by n. This algorithm allows changes in the granularity that will directly affect mutation frequency after minor adjustment.

For our study, the parameters used in the steering logic algorithm are Cfp_max = 16, Cfp_increment = 4, and n = 4.

The hardware cost for the RS-MFU scheme is a new reservation station, steering logic, and extra read and write ports in the integer register file. The cost of new reservation station with 16 entries is estimated, as 3% of the total die size. However, as described in Section 4, we obtain the same performance by using an 8-entry RS-MFU and reducing the floating-point reservation station to 8 entries. Thus there is no die area increase with the use of RS-MFU. In addition, the steering logic is simple and does not add a lot of die size. It also performs in parallel with register renaming so that clock frequency is not affected.

```

Steering logic algorithm:
If (fp_addition instruction)
  Dispatch to RS-MFU
  Cfp ← min(Cfp + Cfp_increment, Cfp_max)
Else
  Cfp ← max(Cfp-1, 0)
  If (Cfp == 0) /* no fp addition bandwidth needed */
    Crr++;
    If (Crr >= n)
      Crr ← Crr - 4 * n;
    If (Crr >= 0)
      Dispatch to RS-MFU
  Else
    Dispatch to other RS

```

Figure 5 Algorithm for the steering logic

3 Simulation

3.1 Tools and Testbeds

We use 3 integer and 3 floating point applications from Spec95 benchmark [19] plus kmeans [10]. Kmeans is basically an iterative clustering algorithm. Clustering algorithms are often used for image processing or computer vision applications. For Spec95 applications, we use the training data set. For kmeans, we use “-D3 -N10000 -K30 -n50” as the parameters.

Table 3: Simulation parameters

Parameter	Value
Fetch width	4
Issue	Out of order
Branch prediction	Bimod, 512 entries
Number of registers	32 int + 32 fp
Functional units	ALU1, ALU2, LSU, FPU1, FPU2
ROB entries	64
Reservation stations	16 entries int, 16 entries addr, 16 entries FP
L1-cache	2-way, 32 KB-I + 32 KB-D, 1 cycle hit

L2-cache	2-way, 4MB, 11 cycle hit, 69 cycle miss
----------	---

SimpleScalar simulator [4] is used for the experiments. We made modifications to SimpleScalar to partially simulate a MIPS R10000. We chose R10000 because it is a well understood RISC architecture. We model the R10000's reservation stations, instruction latencies, and functional unit configuration. The simulator is different from the R10000 in a couple of aspects: the renaming scheme uses a reorder buffer, thus, we set the number of registers to 32 int + 32 fp (instead of 64+64 in R10000). We set the ROB entries to 64 so that only the number of entries of the reservation stations limits instruction dispatch. There is no checkpoint repair mechanism for branch misprediction. So when a branch misprediction after an execution of a branch, immediately the pipeline is flushed and the fetch is redirected. And finally, some parameters shown in Table 3 are not the same with those of R10000. This representative version of the MIPS R10000 will be referred to as our "base architecture" from this point forward.

3.2 Results and Discussions

Figure 6 shows the IPC of the base architecture (first bar), FProf scheme (second bar), RSMon scheme (third bar), RS-MFU scheme with 8-entry RS-MFU and 8-entry floating point reservation station (fourth bar)⁴, and the addition of an integer ALU that also calculates addresses for memory operations (fifth bar). The fifth bar is provided for comparison to assess how effective the RS-MFU and FProf schemes to a less-scalable brute-force approach of adding an extra ALU unit that also performs address generation for memory operations (AGU).

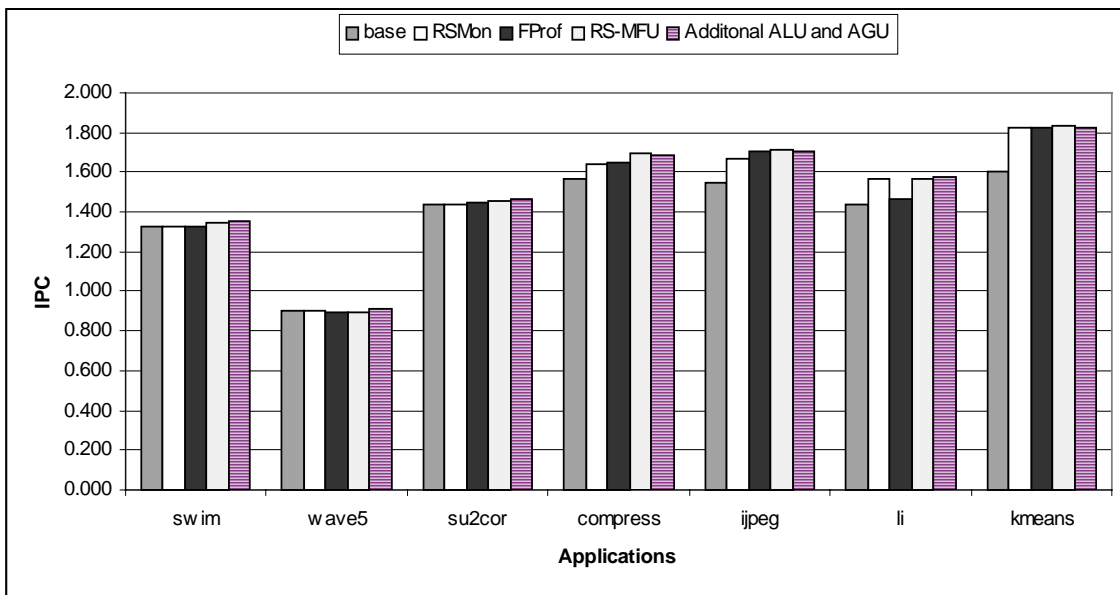


Figure 6: IPC gain for various schemes

⁴ The base R10000 architecture has 16-entry floating point reservation station. By having an 8-entry new RS-MFU and reducing the floating point reservation station to 8 entries, the total number of entries in all reservation stations is the same, thus the die area for reservation stations is roughly the same.

First, for all schemes, the IPC for floating point applications do not change. By comparing the base with the additional ALU/AGU scheme we know that floating point operations do not need additional integer execution or address generation bandwidth. Consequently, adding an MFU, which adds integer execution and address generation bandwidth, will have little impact.

For integer applications, we have interesting results. All schemes improve the IPC for all applications. However, the improvement of FProf and RSMon schemes are always lower than the improvement from the RS-MFU scheme, especially for compress and jpeg.

For RS-MFU scheme, the IPC of integer applications improves from 8.3% for compress to 14.3% for kmeans. The steering of integer and memory instructions, when floating point additions are not present, to the RS-MFU explains this performance improvement. Thus, the MFU will provide more integer execution bandwidth most of the time.

Note that memory instructions that are sent to RS-MFU are also sent to the address reservation station because the MFU only performs address generation and passes the results to the corresponding instructions in the address reservation station. The address reservation station actually holds the memory instructions until the actual load or store operations have been

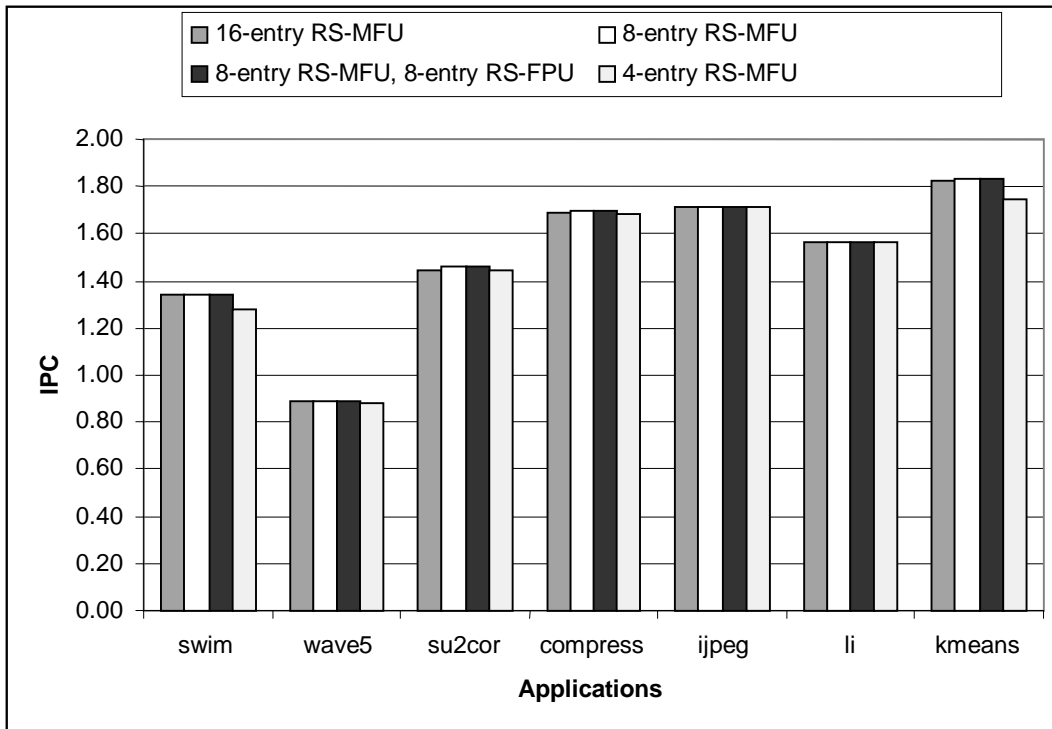


Figure 7 : IPC comparison for varying number of entries of RS-MFU

completed and committed. Thus memory instructions enter RS-MFU at the dispatch stage, and leave the RS-MFU at the issue stage.

For floating-point applications, the IPC is either increased or decreased by a very small amount. This is due to the fact that the MFU does not add any additional floating-point execution bandwidth. Thus, the MFU is only beneficial when there is no floating-point addition, for example, during initialization phase. This extra bandwidth gives swim a little bit of IPC improvement (0.8%). However, for wave5 and su2cor, the additional IPC is offset by the cost of mutation. The mutation frequency for wave5 and su2cor is high, causing high total mutation

penalty. Table 4 shows that for floating-point applications, the mutation frequency translates directly into IPC gain: the more frequent the mutation occurs, the lower the IPC gain.

Table 4: Mutation frequency

Application	Avg. #instructions per mutation
102.swim	40.5
146.wave5	16.5
103.su2cor	21.1
129.compress	370
132.jpeg	7219576
130.li	7065704
Kmeans	325

The mutation frequency for jpeg and li is very low because there are no floating-point addition instructions. Kmeans and compress, however, have 0.7% and 0.5% floating-point addition instructions, thus the mutation frequency is higher than jpeg and li.

Figure 6 also shows that RS-MFU scheme achieves comparable IPC with a brute-force approach of adding an extra ALU unit. The figure shows there is virtually no IPC difference between RS-MFU scheme and the brute force approach for integer applications. For floating-point applications, the RS-MFU scheme achieves over 97% of the IPCs of the brute-force approach (98.9% for swim, 97.7% for wave5, and 98.3% for su2cor). Thus, the RS-MFU scheme achieves comparable performance compared to the brute-force approach. Overall, we have also shown that the RS-MFU improves the performance of integer applications significantly while maintains the performance of floating point applications without adding an extra functional unit.

The effect of the size of RS-MFU (number of entries) is shown in Figure 7. In addition to the 8-entry RS-MFU with 8-entry floating point reservation station that is shown in Figure 6 (base RS-MFU scheme), Figure 7 shows IPC of a 16-entry RS-MFU, an 8-entry RS-MFU, and a 4-entry RS-MFU, all with the original 16-entry floating-point reservation station. We found that there is virtually no difference in IPC between the base RS-MFU scheme with the 16-entry RS-MFU. While we estimate that a 16-entry RS-MFU increases the R10000 die size by 3%, an 8-entry RS-MFU with 8-entry floating point reservation station will not result in an increase of die size, due to the same total number of entries in all reservation stations compared to the base R10000 architecture. Actually the hardware area for reservation stations decreases a little bit because the RS-MFU is using in-order issue, while the original 16-entry floating point reservation station implements out-of-order issue. The reason that we don't lose performance when reducing the number of entries in the floating point reservation station to 8 is that the reservation station now only holds multiplication, division, and square root instructions, with all additions sent to RS-MFU.

One interesting thing to note is that for su2cor, the 8-entry RS-MFU achieves a little better IPC than a 16-entry RS-MFU. The reason for this is that the 8-entry RS-MFU provides better instruction distribution across the reservation stations. When there are no floating-point instructions, placing an integer or memory operation in RS-MFU or other reservation stations (because the RS-MFU is full) can make a difference in performance. And in this case, the performance is higher when we dispatch the operations to other reservation stations. For a 4-entry RS-MFU, however, IPC is lost compared to a 8-entry RS-MFU. The reason for this is that

instruction distribution worsens when we can only put 4 instructions in the RS-MFU. Thus, more instructions are sent to other reservation stations giving other functional units increased loads. Table 5 shows the percent of execution time the RS-MFU is full. Swim and kmeans are the applications that lost IPC the most when using a 4-entry RS-MFU. Not coincidentally, for swim and kmeans, the percent of time the RS-MFU is full are also the highest among floating point and integer codes respectively: 56.96% and 31.15%.

Table 5: Percent of time RS-MFU is full

Applications	16-entry RS-MFU	8-entry RS-MFU	4-entry RS-MFU
swim	0	12.24 %	56.96 %
wave5	0	6.97 %	44.33 %
su2cor	0	7.26 %	42.67 %
compress	0	0.96 %	22.15 %
jpeg	0	2.44 %	25.19 %
li	0	0.24 %	18.09 %
kmeans	0	0.61 %	31.15 %

4 Conclusions and Future Work

We have presented an architecture that exploits extra bandwidth provided by a mutable functional unit by adding a new reservation station (RS-MFU). The performance gain is practically equivalent to adding an additional ALU and AGU into MIPS R10000-like architecture. We have shown through simulation that the new architecture can speedup integer applications ranging from 8.3% to 14.3%. For floating-point applications, the new architecture has no impact on the performance, i.e. speeding up or down less than 1%. Because we add a separate 8-entry reservation station while reducing the floating point reservation station to 8 entries, the only hardware cost is due to extra integer register ports and a steering logic, which we estimate to be less than 1% of the die area. Furthermore, since the steering logic is performed in parallel with register renaming, clock frequency is not affected.

Further work will focus on three objectives. The first objective, to be done in very near future, is to use more applications, including non-scientific codes, to validate whether the performance gain of our architecture schemes in exploiting the MFU also apply to broader classes of applications. The second objective is to evaluate the integration of our architecture ideas with a trace cache, in the context of a wider issue superscalar processor architecture. The final objective is to evaluate hardware/compiler hybrid scheme to exploit the MFU, especially with relations to a VLIW architecture and compiler. As pointed in section 2, one potential advantage of the MFU to VLIW architectures is it gives more flexibility to the compiler in bundling instructions as the MFU can accept almost all types of instructions. This flexibility may result in higher ILP. This requires further study.

Bibliography

- [1] C. Rupp et. al., "The NAPA Adaptive Processing Architecture", IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, California, April, 1998.
- [2] Daniel H. Friendly, Sanjay J. Patel, Yale N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors", Proceedings of the 31st ACM/IEEE International Symposium on Microarchitecture, Dallas, Texas, Dec 1998.
- [3] Dominique Lavenier, Yan Solihin, Kirk W. Cameron, "Integer/Floating-point Reconfigurable ALU", Technical Report LA-UR #99-5535, Los Alamos National Laboratory, Sep 1999.
- [4] Doug Burger and Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0", Technical Report #1342, University of Wisconsin-Madison Computer Sciences Department, June 1997.
- [5] John L. Hennessy and David A Patterson, "Computer Architecture: a Quantitative Approach", pp 341, Morgan Kaufmann Publishers, Inc., 2nd ed., 1996.
- [6] John R. Hauser and John Wawrzynek, Garp: A MIPS Processor with Reconfigurable Coprocessor, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, pp. 24-33, April 16-18, 1997.
- [7] Kenneth C. Yeager, "The MIPS R10000 Superscalar Microprocessor", Proceedings of the 27th Annual International Symposium on Microarchitecture, pp. 28-40, 1996.
- [8] Kirk W. Cameron, Yong Luo, "Instruction Level Modeling of Scientific Applications", Proceedings of the ISHPC 99, May 1999.
- [9] Kirk W. Cameron, Yan Solihin, Yong Luo, Workload Characterization Via Instruction Clustering Analysis", Technical Report LA-UR ____, Los Alamos National Laboratory.
- [10] "K-Means Algorithm for unsupervised classification", <http://www.ece.neu.edu/groups/rpl/kmeans>
- [11] M. Gokhale and J. Stone, "Compiling for Hybrid RISC/FPGA Architecture", IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, California, April, 1998.
- [12] Mikko H. Lipasti and John Paul Shen, "Superspeculative Microarchitecture for Beyond AD 2000", IEEE Micro, pp. 59-66, Sep 1997.
- [13] Q. Jacobson, J.E. Smith, "Instruction Pre-processing in Trace Processors", Proceedings the 5th International Symposium on High Performance Computer Architecture, January, 1999.
- [14] Rahul Razdan and Michael D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", Proceedings of the 27th Annual International Symposium on Microarchitecture, 1994.
- [15] Ralph D. Wittig and Paul Chow, "OneChip: An FPGA Processor with Reconfigurable Logic", Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, 1996.
- [16] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, Jeffrey P. Kao, The Chimaera reconfigurable Functional Unit", IEEE Symposium on Field-Programmable Custom Computing Machines, 1997.
- [17] Subbarao Palacharla, James E. Smith, "Complexity-Effective Superscalar Processors", ISCA, 1997.
- [18] Subbarao Palacharla, J.E. Smith, "Decoupling Integer Execution in Superscalar Processors", Proceedings of the 28th Annual International Symposium on Microarchitecture, 1995
- [19] Standard Performance Evaluation Corporation. www.spec.org.
- [20] S. Subramanya Sastry, Subbarao Palacharla, James E. Smith, "Exploiting Idle Floating-Point Resources for Integer Execution", ACM SIGPLAN Conference on Programming Language Design and Implementation, 1998.
- [21] Synopsis. <Http://www.darpa.mil/ito/psum1998/G052-0.html>.

[22] Yale N. Patt, Sanjay J. Patel, Marius Evers, Daniel H. Friendly, Jared Stark, "One Billion Transistors, One Uniprocessor, One Chip", *IEEE Micro*, pp. 51-57, Sep 1997.