Mutable Functional Units and Their Applications on Microprocessors *

Yan Solihin[1], Kirk W. Cameron[2], Yong Luo[3], Dominique Lavenier[4], Maya Gokhale[5]
[1] *University of Illinois at Urbana-Champaign*
[2] *University of South Carolina*
[3] *Intel Corporation*
[4] *IRISA/CNRS*
[5] *Los Alamos National Laboratory*
solihin@cs.uiuc.edu, kcameron@cse.sc.edu, yong.luo@intel.com, lavenier@irisa.fr, maya@lanl.gov

Abstract

Functional units are the heart of microprocessors as they execute binary instructions of a program. Current microprocessors typically have several types of functional units. In this paper, we propose a new functional unit that combines a floating-point adder and an integer arithmetic and logic unit into a single unit. This functional unit reconfigures itself at run-time to serve different instructions from the program instruction stream. We call such units mutable functional units *or MFUs. MFUs can be used in microprocessors to improve functional unit utilization, reduce power consumption, and to improve performance without adding extra functional units. MFUs only require minor modifications to the existing floating-point adder design. We show that overheads of reconfiguration are small, typically 0 to 1 clock cycle, and at most 2 clock cycles. We demonstrate how integration with a typical current microprocessor can be achieved. This integration allows speedups of non-numerical applications by 8% to 14% while keeping the number of functional units constant. We also show that various enhancements to the base architecture that increase the instruction fetch rate affect the speedups positively.*

1 Introduction

Functional units are the heart of microprocessors as they execute binary instructions of a program. Current microprocessors typically have several types of functional units to execute different types of instructions: integer arithmetic and logic unit (ALU), integer multiplier, integer shifter, floating-point adder, multiplier/divider, square root units, etc. Today microprocessors exploit instruction level parallelism (ILP) by fetching and executing several instructions in the same clock cycle at different functional units. Functional units may consume a significant portion of power consumption in current microprocessor chips. These drawbacks will be even more pronounced in future processors where there may be a large number of functional units per chip such as the IBM BlueGene which may have well over 300 functional units per chip [2].

*This work is supported by Los Alamos National Laboratory under grant LDRD ER 2000022. Preliminary idea of this paper appears in [3]

One type of resources that is often underutilized are floating-point related resources, which include floating-point units, registers, and reservation stations. Non-numerical applications usually have very few floating-point computation, and, as a result, the floating-point resources are mostly idle. When idle, the floating-point resources consume power and waste die area. To reuse floating-point resources, the first modification is to enable the floating-point units to execute integer instructions. In this paper, we propose to extend floating-point adders with the capability to execute integer instructions. Floating-point adder already contains most of the hardware needed for integer execution. Simply extending the operand width and adding a few switches enable the units to be "reconfigured" to execute different types of instruction. We call these units *mutable functional units* or MFUs, and the reconfiguration process as *mutation.* To exploit MFUs, we need to mutate them to adapt to the program's needs during its execution. Thus, the mutation must happen at run-time with low overhead. We show that this is the case, where the mutation overhead is at most 2 clock cycles.

We also demonstrate how integration of the MFU with microprocessors is possible, and evaluate it on a specific platform of a popular superscalar processor, the MIPS R10000. In this platform, we keep the number of functional units constant, and replace the processor's floating-point adder by an MFU to increase its utilization for non-numerical applications. We show that the integration can be very simple and require relatively little new hardware without impacting the clock cycle frequency of the processor. Using a cycle accurate microprocessor simulator, the extra resource provided by the MFU for non-numerical applications allows speedups of the applications by 8% to 14%. We also show that various enhancements to the base architecture that increase the instruction fetch rate affect the speedups positively.

The rest of this paper is organized as follows: Section 2 describes related works. Section 3 describes the design of the mutable functional units. Section 4 describes the the integration of the MFU in microprocessors, with a specific example of the MIPS R10000. Section 5 describes evaluation environment of the study and Section 6 discusses the evaluation results on both base and enhanced R10000 architectures. Finally, Section 7 presents conclusions and future work.

2 Related Work

To achieve reconfigurability, people have proposed using reconfigurable fabric to augment the capability of a conventional processor [7, 6, 13, 10, 11]. Using reconfigurable fabric significantly increases the complexity of the compiler, the hardware integration, and the synchronization between the fabric and the main processor. Since our reconfigurability feature only involves switching between floating-point and integer execution, we choose to use a few simple programmable switches in the functional unit instead of using reconfigurable fabric.

A hardware component that can execute both integer and floating-point instructions was mentioned in [9]. No functional unit design was presented and it is not clear whether the component is a single functional unit or a group of integer and floating-point units. In a further study [5], the existence of such functional or group of functional units were assumed and performance was improved on superscalar architectures utilizing these units. Modifications to the instruction sets were added, and the compiler must perform grouping of integer instructions: one group, containing instructions with their dependence chain, is routed to these units; and another group goes to traditional integer units. In our study, we present the actual design of functional units that are able to serve both integer and floating-point instructions and discuss the reconfiguration overheads. No modifications to the compiler and instruction sets are needed because the hardware steer instructions to the functional units transparently.

3 Mutable Functional Unit Design

This section describes the design of a mutable functional unit (MFU) and its mutation penalties.

3.1 MFU Design

In designing a mutable functional unit (MFU), we would like to make use of existing hardware in the floating-point adder and minimize modifications. We restrict the MFU to perform integer arithmetic, logic, shift, and floating-point addition, and can be configured in one of two modes. In the *integer mode*, it can perform 64-bit integer addition, subtraction, shift, and logic operations. In the *floating-point mode*, the MFU can perform double precision standard 754 floating-point addition.

We base our MFU design on the R10000's floating-point adder [14]. The floating-point adder has three pipeline stages: align, add, and pack. The first pipeline stage (align) contains a right shifter, the second (add) contains a 54-bit adder, and the third (pack) contains a left shifter. The modifications that we do to the floating-point units are minor and do not consume much die area. They are:

- extension of the adder from 54 bit to 64 bits. The adder serves integer addition/subtraction and floating-point addition/subtraction.
- substitution of the 53-bit right shifter by a 64-bit barrel shifter. The shifter performs both left and right shift integer instructions and alignment for floating-point addition/subtraction.
- insertion of 4 programmable switches along the data-path

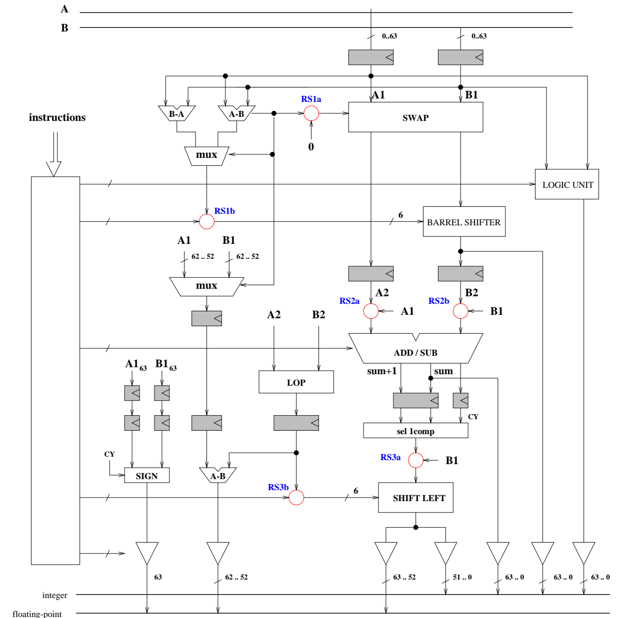

Figure 1: Double precision IEEE 754 floating point adder

- addition of a logic unit to execute integer logic instructions.

The resulting design is shown in Figure 1. The circles represent the programmable switches. The MFU takes two operands as input, and two outputs are dedicated respectively to integer and floating-point results. In the *integer mode*, the swap unit is disabled (switch RS1a). Hence, the input of the barrel shifter is B1. The two inputs of the adder are respectively connected to A1 and B1 by the two switches RS2a and RS2b. In the *floating-point mode*, the inputs of the adder come from the first stage of the pipeline. The barrel shifter is controlled by the operations performed on the exponents. In that scheme, only the 54 least significant bits of both the adder and the barrel shifter are used. To simplify the design, at most one instruction can enter or exit the MFU in a single cycle.

3.2 Mutation Penalties

Switching the MFU mode involves a penalty. Due to the 3-stage pipeline when configured as a floating-point unit, and only 1-stage pipeline when configured as an integer ALU, the time for switching from integer to floating-point and vice versa is not constant. This is shown in Table 1. Given an incoming instruction to be served (column 2 in Table 1), the mutation penalty depends on the current instruction (column 1) and the next instruction that follows the incoming one (column 3). Note that the penalties remain the same for highly pipelined adder, except in Case 6 where the penalty would be equal to the number of pipeline stages for "align" stage + 1. However, since the "add" stage latency is the most significant one, this penalty remains very small compared to the number of pipeline stages.

To illustrate the mechanism and timing of MFU mutation, here we discuss two situations. The first deals with a sequence of an integer addition followed by floating-point addition (Case 1) which does not exhibit mutation penalty. This process is illustrated in Figure 2-(a).

Table 1: MFU mutation penalty

Case		current	incoming	next	Penalty
1	int to	logic/add	fp-add	fp-add	0
2	float	shift	fp-add	fp-add	1
3	float to	fp-add	shift	not add	0
4	int	fp-add	logic	add	1
5		fp-add	shift	all int	1
6		fp-add	int-add	all int	2

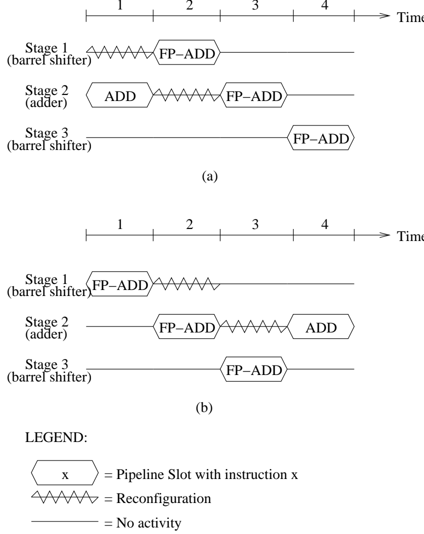

Figure 2: Mutation mechanism and penalty for a sequence of instructions: {ADD, FP-ADD} (a), and {FP-ADD, ADD} (b)

The figure shows the three pipeline stages in the MFU (shifter, adder, shifter) and for each stage, what instruction is executed or reconfiguration is performed. Each time step corresponds to one clock cycle. At time 1, the integer addition (ADD) is served by the adder in the Stage 2 of the pipeline. Since the following instruction is a floating-point adder (FP-ADD), Stage 1 is being reconfigured to accept the FP-ADD. At time 2, ADD has finished execution and FP-ADD can now use Stage 1 while Stage 2 is being reconfigured. At time 3, FP-ADD moves to Stage 2. By this time, reconfiguration has completed. Thus at time 4, the FP-ADD simply flows through the pipeline to Stage 3. Since ADD and FP-ADD are issued to the MFU back-to-back at time 1 and 2 respectively, the mutation penalty is fully hidden.

The second situation deals with a sequence of floating-point addition followed by an integer addition (Case 6) which requires 2-cycle mutation penalty. At time 1, FP-ADD is served by the shifter in Stage 1. At time 2, FP-ADD has moved to Stage 2 of the pipeline and Stage 1 can now be reconfigured to accept ADD. However, the adder needed by the integer addition is still occupied by FP-ADD. At time 3, FP-ADD moves to Stage 3 while Stage 2 is being reconfigured. At time 4, FP-ADD has completed execution while ADD is now being served by the adder. Since FP-ADD is issued at time 1 and ADD is issued at time 4, for two clock cycles (time 2 and 3) the MFU cannot accept new instruction.

Although the mutation penalty is low (at most 2 cycles), we need to make sure that mutation does not happen very frequently.

4 MFU Integration

Microprocessors that will likely benefit the most from an MFU integration is the ones that have many functional units per chip. These microprocessors are likely to be partitioned into clusters, with each cluster consisting of an issue path, functional units, and register files. This architecture is illustrated in Figure 3. In Figure 3-(a), instruction decoding is performed globally, followed by a hardware logic to route the decoded instructions into the clusters. Alternatively, this routing can also be performed statically by the compiler in a similar fashion to a VLIW machine. Figure 3-(b) shows the hardware does not contain steering logic, and instruction decoding can now be performed locally in each cluster. The latter approach is in line with Sun's MAJC processor architecture [8]. In both cases, the communication bus allows communication across different register files. The last stage is instruction retirement. The MFU can be integrated in each cluster to support both integer and floating-point execution.

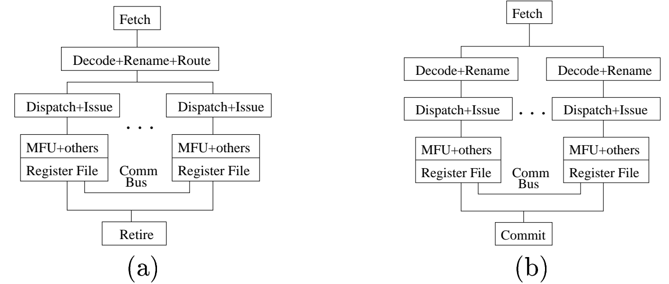

Figure 3: MFU integration in multi-cluster microprocessors

As a first step in the evaluation of MFU integration in microprocessors, in this section we only focus on integrating the MFU in a single cluster environment that is based on a current superscalar microprocessor, the MIPS R10000.

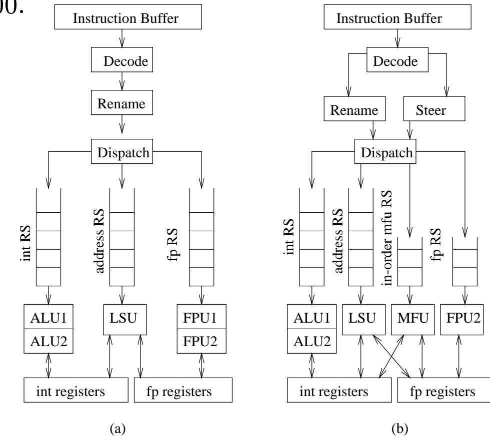

Figure 4: (a) Base R10000 (b) Base R10000 with integrated MFU.

4.1 Integration on the R10000 architecture

We base the architecture on a widely studied processor architecture, the MIPS R10000. It is a superscalar

processor that can fetch and dispatch four instructions every clock cycle. The instruction flow is partially shown with the functional units in Figure 4-(a). The R10000 has 2 integer ALUs, ALU1 is capable of performing basic operations (add/sub, logic) plus branch and shift operations, and ALU2 is capable of performing basic plus integer multiplication and division. There is one Address Generation Unit (AGU) which is embedded in the Load Store Unit (LSU). 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. Analysis of instructions is performed at the dispatch stage right after fetched instructions are decoded for operands. After decoding the operands, register renaming is performed to remove write-after-write and write-after-read dependences. Then instructions are dispatched to various reservation stations depending on their types. When all the operands of an instruction is available, the instruction is issued to a functional unit, even if earlier instructions have not been issued. This issuing scheme is called out-of-order. The execution results are then stored to the registers and also put on the forwarding bus to notify other functional units and reservation stations.

Our initial evaluation of the processor reveals that the R10000 functional unit configuration does not give optimal performance, where non-numerical applications suffer significantly from lacking integer units. The IPCs are reduced by 14% to 32% because of this problem. By converting the FPU1 to MFU, the functional unit can now provide extra integer execution bandwidth which would otherwise be idle.

Our MFU integration is shown in Figure 4-(b). It has an extra processing after decoding stage called *instruction steering*, which is performed in parallel with the renaming stage, thus avoiding possible increase in clock latency. The steering stage selects instructions to be executed by the MFU. The selected instructions will be dispatched to a new reservation station dedicated to the MFU, that will issue the instructions only to the MFU. In contrast to other reservation stations, this new reservation station issues instructions in a strict FIFO (in-order) manner. This makes the reservation station simpler and smaller to implement. When the MFU detects that the new instruction that is issued to it has a different type compared to the one it is serving, it performs mutation, then it executes that instruction. As with other functional units, the results of the MFU computation are also put on the forwarding bus.

In the evaluation section (Section 6) we evaluate how many entries the new reservation station needs to have. We find that 8 entries, half the size of other reservation stations in R10000, is sufficient. Furthermore, since the reservation station will now accommodate all floating point addition operations, the floating-point reservation station now only contains floating point multiplication, division, and square root operations. Thus, the number of entries in the floating-point reservation station can also be reduced from 16 entries to 8 entries. Thus, we have kept the total number of reservation station entries constant.

4.2 Steering Logic Algorithm

The algorithm that we use for instruction steering logic is shown in Figure 5. The algorithm uses a saturating counter, cfp, to detect whether there is a need for floating-point addition bandwidth. If there is, indicated by a positive value of cfp, the steering logic only dispatches floating-point addition operations into the MFU's reservation station. Otherwise (cfp = 0), the steering logic dispatches integer and memory operations into the MFU's reservation station in an n-chunk round-robin manner, which is basically a round robin with a granularity of n instructions. The counter cfp is controlled by 2 parameters: cfp_increment, and cfp_max. crr is used to control the round-robin scheduling, where the chunk size is specified by n. For our study, we use cfp_max = 16, cfp_increment = 4, and n = 4. The adders, comparators, multiplexers needed by the steering logic are small, since they only operate on 4-bit data.

```
if (fp_addition instruction) {
   Dispatch to MFU's reservation station
   cfp = min(cfp + cfp_increment, cfp_max)
} else {
   cfp = max(cfp-1, 0)

   /* no fp addition bandwidth needed */
   if (cfp == 0) {
       crr++;
       if (crr >= n)
           crr = crr - 4 * n;
       if (crr >= 0)
           Dispatch to MFU RS
       else
         Dispatch to other RS
   }
}
```

Figure 5: Steering logic algorithm

4.3 Integration on Enhanced R10000

We hypothesize that the performance improvement provided by the MFU is related to the utilization of the MFU. The best performance is obtained when there are enough integer instructions to dispatch to the MFU to keep the utilization high, up to a point where saturation in the utilization is reached. In particular, we enhance the R10000 architecture by increasing the ability of the processor to fetch more instructions per cycle. Although some of the enhancements are too optimistic for real implementation, they are suitable in evaluating our hypothesis. The enhancements are increasing the issue width of the processor to 8 and 16 instructions, larger on chip cache, perfect branch prediction, and processor-memory integration. All the enhancements increase the fetch or decode rate potential of the processor by increasing number of available instructions to dispatch (wider issue), less instruction and data misses (larger caches), lower cache miss penalties (processor-memory-integration), and more available instructions across branches (perfect branch prediction).

5 Evaluation Environment

Applications. We use numerical and non-numerical applications from Spec95 benchmark [1] plus kmeans [12]

for our study. These include perl, li, ijpeg, compress, swim, su2cor, and wave5. The description of the codes and the input sets used are shown in Table 2.

Table 2: Application and input sets description

App.	Description	Input Set
Swim	Shallow water simulation	Train
Wave5	Maxwell's equations	Train
Su2cor	Monte-Carlo method	Train
Compress	Lempel-Ziv file compression	Train
Ijpeg	Image compression/decompr.	Train
Li	Xlisp interpreter	Train
Kmeans	Iterative clustering	-D3 -N10000 -K30 -n50
Perl	Perl language interpreter	Train

Simulator. Our simulation is based on Simplescalar version 3.0 [4], a cycle-accurate superscalar processor simulator. We modify the simulator to simulate closely a MIPS R10000, except in the usage of reorder buffer and the mechanism of recovery from branch misprediction. The simulation parameters of the base R10000 and its enhancements are shown in Table 3. In the Enhanced R10000 and Integrated MFU, the table only shows the parameters that change. We integrate the MFU, new reservation station, and steering logic to the base as well as enhanced R10000 architectures.

Table 3: Simulation parameters.

Architecture	Parameters and Value
Base R10000	Fetch, decode, and commit 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 (R10000 latencies) ROB: 64 entries Resv stations: 16 entries int, addr, and fp L1: 2-way, 32 KB-I + 32 KB-D, 1 cycle hit L2: 2-way, 4MB, 11 cycle hit, 69 cycle miss
Enhanced R10000	*wider issue*: Fetch, decode width: 8-way and 16-way *Perfect branch prediction (pbp)* *Larger on-chip cache (loc)*:4-way, 128 KB-I + 4-way, 128 KB-D L1, 1 cycle hit *Integrated processor and memory (pim)*: no L2, mem access 5 cycles
Integrated MFU	Func units: ALU1, ALU2, LSU, MFU, FPU2 (R10000 latencies) Reservation stations 16-entry int and addr, 8-entry mfu and fp

6 Evaluation

6.1 Results on Base R10000

Figure 6 shows speedups of R10000(base)+MFU over the R10000(base) for each applications. The speedups are calculated by comparing throughput or IPCs. Each application has four bars. The first three bars of each application show the speedups of the MFU integration with various number of reservation station entries: 4 entries (MFU-4 on the first bar), 8 entries (MFU-8 on the second bar), and 16 entries (MFU-16 on the third bar). The last bar of each application shows the "ideal" speedup, an upperbound performance case where the MFU has no mutation penalties and that is able to concurrently execute floating-point and integer instructions. The "ideal" MFU is an expensive implementation which requires an addition of an extra integer ALU to the FPU1 in the original R10000. First, the figure shows that the reservation station for the MFU needs to have at least 8 entries, as the speedups are noticeably lower with only 4 entries. Comparing 8 entries with 16 entries, however, indicates that there is virtually no difference in speedups, suggesting that 8 entries are sufficient.

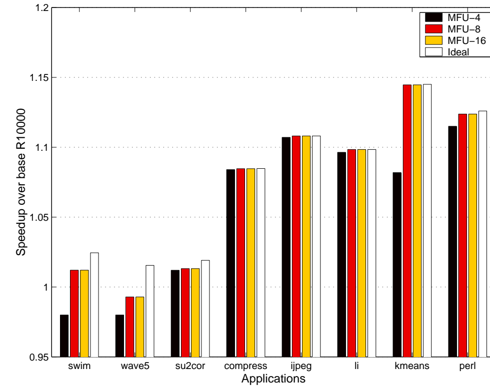

Figure 6: Speedup of R10000 with MFU over base R10000

Now, let us consider the 8-entry reservation station in Figure 6. The figure shows that IPC of non-numerical applications is improved significantly, with speedups ranging from 8.3% for compress to 14.3% for kmeans. These speedups are very close to the ideal speedups, showing the effectiveness of the MFU. For floating-point applications, there is virtually no speedups. This is expected as the MFU is always busy with floating-point add instructions, leaving no room for executing integer instructions. The ideal speedups, however, show minor speedups because the functional unit can execute both floating-point and integer instructions at the same time.

Table 4 shows the percentage of time that the reservation station dedicated to MFU is full for various number of entries. The averages are presented for numerical applications and non-numerical applications. Firstly, the table shows that with 16-entry, the reservation station is never full. Secondly, it shows that the averages for numerical applications are higher compared to non-numerical applications, 48.0% vs. 24.2% for 4-entry, and 8.8% vs. 1.1% for 8-entry. This is because numerical applications have a lot more floating-point additions that the reservation station has to handle. A 4-entry reservation station is clearly saturated. However, a 8-entry reservation station handles it almost as well as the 16-entry for non-numerical applications, in average it is only full for 1.1% of the time, confirming the results of Figure 6 where the speedups for 8-entry and 16-entry MFU's reservation station are almost identical.

Table 4: Percent of time the MFU's reservation station is full

Application	4-entry	8-entry	16-entry
Swim	57.0%	12.2%	0%
Wave5	33.3%	7.0%	0%
Su2cor	42.7%	7.3%	0%
Average	**48.0%**	**8.8%**	**0%**
Compress	22.2%	1.0%	0%
Ijpeg	25.2%	2.4%	0%
Li	18.1%	0.2%	0%
Kmeans	31.2%	0.6%	0%
Average	**24.2%**	**1.1%**	**0%**

To get deeper insights into the effects of mutation penalties, we profile the mutation distance (how many instructions are executed between two mutations). For non-numerical applications (compress, ijpeg, li, and kmeans), it ranges from 325 to 7.3M instructions, with an average of 3.5M instructions. This indicates that mutation is very seldom and the mutation penalties are insignificant. For numerical applications (swim, wave5, and su2cor) it ranges from 16.5 to 40.5 instructions, with an average of 26.5 instructions. Although it is more often, but it is still tolerable.

6.2 Results on Enhanced R10000

Table 4 shows that for non-numerical applications, the reservation station is very seldom full (< 2.5% for all applications). This suggests that probably the MFU is still under-utilized. However, for numerical applications, the reservation station is in average full for 8.8% of the time, which suggests that the MFU may already be saturated that the enhancements may not yield better speedups.

Figure 7 shows the resulting speedups of MFU integration to various enhanced R10000 architectures. There are six bars for each application. For reference, the first bar for each application shows the speedups of MFU integration in the base R10000 architecture (same as the second bar in Figure 6) The next five bars show the speedups of MFU integration on enhanced R10000 architectures that include larger on chip cache (second bar), processor memory integration (third bar), 8-way superscalar (fourth bar), 16-way superscalar (fifth bar), and perfect branch prediction (sixth bar).

For all non-numerical applications, the MFU integration in the enhanced architectures generally provide better speedups compared to the base R10000 architecture, although in varying degrees. Wider fetch (R10000(8-way) and R10000(16-way) in the figure) enhance the speedups the most, followed by perfect branch prediction (pbp), large on chip cache (loc), and processor-memory integration (pim). In all cases, the speedups range from 9.1% to 20.7%. As previously suspected, numerical applications' speedups are generally not affected.

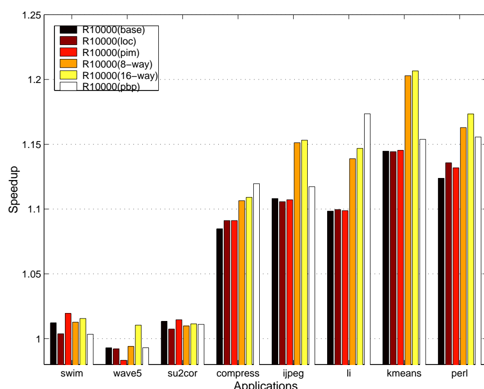

Figure 7: Speedups of MFU integration in the base and enhanced R10000 architectures.

7 Conclusions and Future Work

Conclusions. In this paper, we propose a *mutable functional unit* (MFU) that combine a floating-point adder and an integer arithmetic and logic unit (ALU) into a single functional unit, which reconfigures itself at run-time to serve different types of instruction from a program instruction stream. MFUs can be used in microprocessors to improve functional unit utilization, reduce power consumption, and to improve performance without increase in die area. MFUs only require minor modifications to the existing floating-point adder design. Overheads of reconfiguration are small, typically 0 to 1 clock cycle, and at most 2 clock cycles. We demonstrate how simple integration with a microprocessor based on the R10000 can be achieved. This integration allows speedups of non-numerical applications by 8% to 14% while keeping the number of functional units constant. We also show that various enhancements to the base architecture that increase the instruction fetch rate affect the speedups positively.

Future Work. For future study, we plan to evaluate the integration of the MFU on multi-cluster architecture, especially those with a high number of functional units per chip, such as the IBM BlueGene [2]. Other aspects, including hardware cost and power consumption will also be investigated.

References

[1] Standard Pefromance Evaluation Corporation. http://www.spec.org.

[2] IBM corporations. Bluegene project. *http://www.research.ibm.com/bluegene.*

[3] Dominique Lavenier, Yan Solihin, and Kirk W. Cameron. Reconfigurable arithmetic and logic unit. *SympA'6: Symposium sur les Architecture Nouvelle de Machine*, Besancon, France, June 2000.

[4] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *Technical Report 1342, University of Wisconsin-Madison Computer Science Department*, June 1997.

[5] S. Subramanya Sastry et al. Exploiting idle floating-point resources for integer execution. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.

[6] Maya Gokhale and J. Stone. Compiling for hybrid rsic/fpga architecture. *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[7] John R. Hauser and John Wawryznek. Garp: a mips processor with reconfigurable coprocessor. *IEEE Symposium on FPGAs for Custom-Computing Machines*, pages 24–33, April 1997.

[8] Sun Microsystems. Majc-5200: a high performance microprocessors for multimedia computing. *http://www.sun.com/microelectronics/majc.*

[9] Subbarao Palacharla and J.E.Smith. Decoupling integer execution in superscalar processors. *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995.

[10] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. *Proceedings the 27th Annual International Symposium on Microarchitecture*, 1994.

[11] Synopsis. http://www.darpa.mil/ito/psum1998/g052-0.html.

[12] James Theiler and G. Gisler. A contiguity-enhanced k-means clustering algorithm for unsupervised multispectral image segmentation. *Proceedings of the SPIE 3159, web http://www.ece.new.edu/groups/rpl/kmeans*, pages 108–118, 1997.

[13] 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.

[14] Kenneth C. Yeager. The mips r10000 superscalar microarchitecture. *Proceedings the 27th Annual International Symposium on Microarchitecture*, pages 28–40, 1996.