Los Alamos National Laboratory Unclassified Technical Report LA-UR #99-6768

Boosting the Speedup of Future Processor Architectures by Using Mutable Functional Units

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

† Los Alamos National Laboratory
{solihin, kirk, lavenier, maya}@lanl.gov

‡ Intel Corporation
yong.luo@intel.com

Abstract

One major bottleneck of a superscalar processor is the mismatch of instruction stream mix with functional unit configuration. The resulting "unavailable functional unit" stalls can be a significant factor of performance loss. Recently, mutable functional units (MFUs) - functional units that can serve both floating-point and integer operations - have been proposed to reduce this type of stall. The benefit to our implementation of an MFU is increased integer execution bandwidth without increased die area or power consumption. In this paper, we show that the ILP gain, as well as the speedup, provided by the MFU increases in many architecture modifications expected in the future. The speedup ranges from 8% to 22%. We conclude that MFUs show promise in improving the performance of future architectures.

1 Introduction

Superscalar microprocessor architecture attempts to exploit instruction level parallelism (ILP, measured as instructions per cycle or IPC) by fetching and issuing multiple instructions every cycle. Since the issued instructions may be of 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 IPC by avoiding any stalls due to unavailable functional units. However, this approach increases the die area occupied by the functional units and increases the complexity of the bypass network[1], increases the complexity of superscalar architecture because it contains long wires. Subbarao et al. [16] pointed out that the bypass delay in a bypass network (data bypass logic) grows quadratically with issue width, which also means that it is proportional to the quadratic value of the number of functional units[2]. The paper concluded that for an 8-way superscalar processor implemented on a 0.18 μm process, the bypass network is the most significant factor that limits clock frequency. Thus, reducing the number of functional units is desirable from a hardware point of view.

[1] The 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.

Hardware complexity performance trade-offs are contemplated by architecture designers when determining functional unit configurations that have acceptable complexity without sacrificing too much performance. However, this compromise results in instruction and functional unit mismatch that can be a significant factor in decreased instructions per cycle (IPC) [12, 11]. To solve this problem, recently floating-point functional units that have integer execution capability have been proposed [15, 9, 11]. The units are called mutable functional units in [11] because the units can mutate (with mutation latency) to serve floating-point or integer instructions. The studies differ in the approach taken to exploit the extra integer execution bandwidth provided by MFUs. In [9], the authors use a compiler approach and add to the instruction set architecture. In [11], our approach is transparent to the ISA and requires no compiler modifications. Instead, we use a minimal hardware modification to implement a mutable functional unit. Both studies show that the MFU usage boosts the IPC of integer applications significantly while neither discusses whether a particular architecture modification increases or decreases the benefit of using MFUs.

Thus, we focus on likely architectural modifications and what effect, if any, they have on performance when combined with an MFU design. We present two significant contributions in this context:

- We present a study of the achieved IPC gain over the base non-MFU architecture on various future architecture modifications. Results of this study provide indications of increasing IPC gain provided by MFU on future architectures.
- We present an explanation of why the IPC gain increases on the architecture modifications that we simulate. We also discuss applicability of MFU to real near-future architectures.

The rest of this paper is organized as follows: Section 2 provides the implementation of an MFU and an architecture that exploits it. Section Section 3 presents the architecture modifications we choose to simulate and their simulation parameters. Section 4 presents and discusses the simulation results. Section 5 explains why the IPC gain increases on the architecture modifications and discusses other future architecture modifications. Finally, Section 6 summarizes the study.

2 Architectures for a Mutable Functional Unit

We begin this section by describing briefly the design of a mutable functional unit (MFU), including the mutation penalty. Next we present an architecture scheme that exploits the usage of MFU with minimal hardware modification.

2.1 Mutable Functional Unit

Our goal is to utilize the hardware that exists in floating-point units by augmenting them for integer execution capability. We base our design of MFU on the R10000's floating-point adder [5]. The floating-point adder has three pipeline stages: align, add, and pack, as illustrated in Figure 1. The first pipeline stage (align) contains a right shifter, the second (add) contains a 53-bit adder, and the third (pack) contains a left shifter. So the adder, with a few extensions,

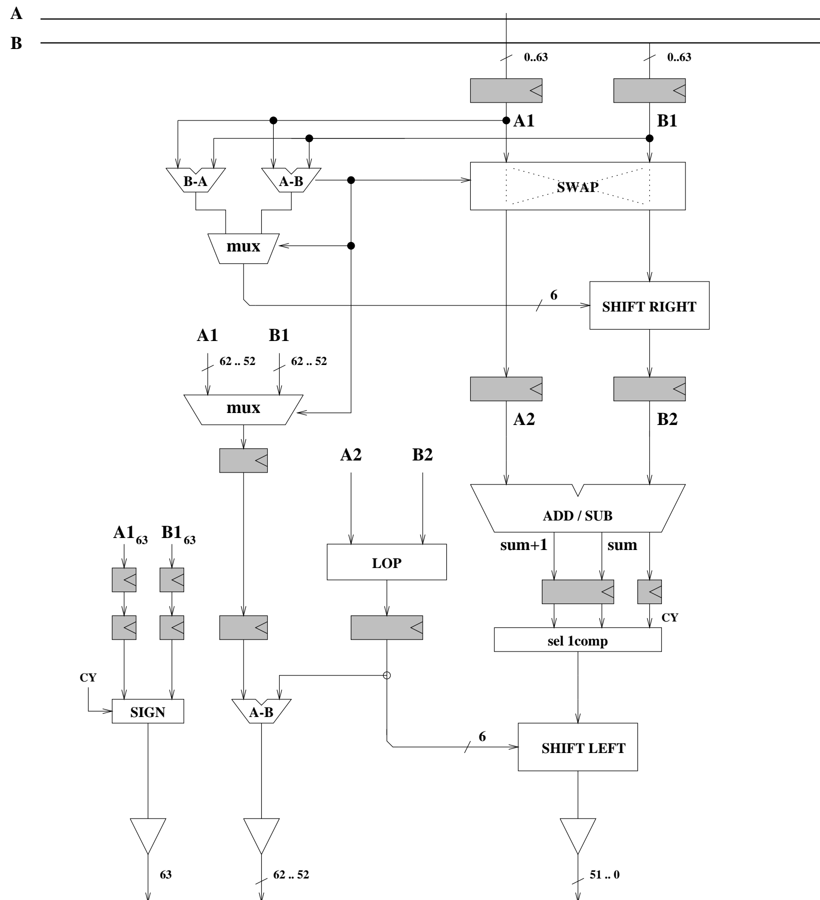

Figure 1: Double precision IEEE 754 floating point adder

has the hardware necessary to perform shift and 64-bit add operations. The idea of a mutable functional unit is to use the hardware in floating-point units to perform integer execution.

Thus, based on profiling results in [13], we designed the MFU to be able to execute integer addition, shift, and logic operations, plus address generation for memory operations. The floating-point adder hardware already provides 53-bit shift capabilities (mantissa). The adder, however, needs to be enlarged to 64 bits. It does not have a logic unit, thus we need to augment it. Finally, a few switches need to be added to change the data path for each of the operations that need to be performed in the MFU. The resulting hardware design revealed that an MFU roughly has the same number of gates as a floating-point adder plus a single logic unit. The details of the hardware design and timing can be found in [5].

One important aspect that affects the performance of the MFU is its mutation penalties, which are shown in Table 1. Mutation penalties are stall cycles due to the MFU changing its current functionality such as the ability to execute integer operations to another functionality such as floating-point operations. The penalty is paid when there are floating-point instructions in the pipeline just prior to a mutation that must be drained from the pipeline stages that the integer operations require. Although in general the mutation penalty is not high, an architectural scheme needs to avoid excessive mutations.

Table 1: MFU mutation penalty

Instruction	Next Instruction	Instruction After Next	Mutation Penalty
logic	fp-add	fp-add	0
shift	fp-add	fp-add	1
fp-add	shift	not int-add	0
fp-add	logic	int-add	1
fp-add	shift	all int	1
fp-add	int-add	all int	2

2.2 Architecture to Exploit MFU

Varying the pipeline stages in which the mutation analysis and the actual mutation are performed, we have studied several alternative architectures [11]. We will briefly mention a promising architecture, which we call RS-MFU.

We chose to use the MIPS R10000 architecture as the basis of our study (shown in 2a). The functional units consist of 2 integer ALUs. 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 (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, and issues instructions in an out-of-order manner to the respective functional units.

A major modification needed to create RS-MFU scheme is to replace the floating-point adder (FPU1) with an MFU, which is able to perform floating point addition, integer addition,

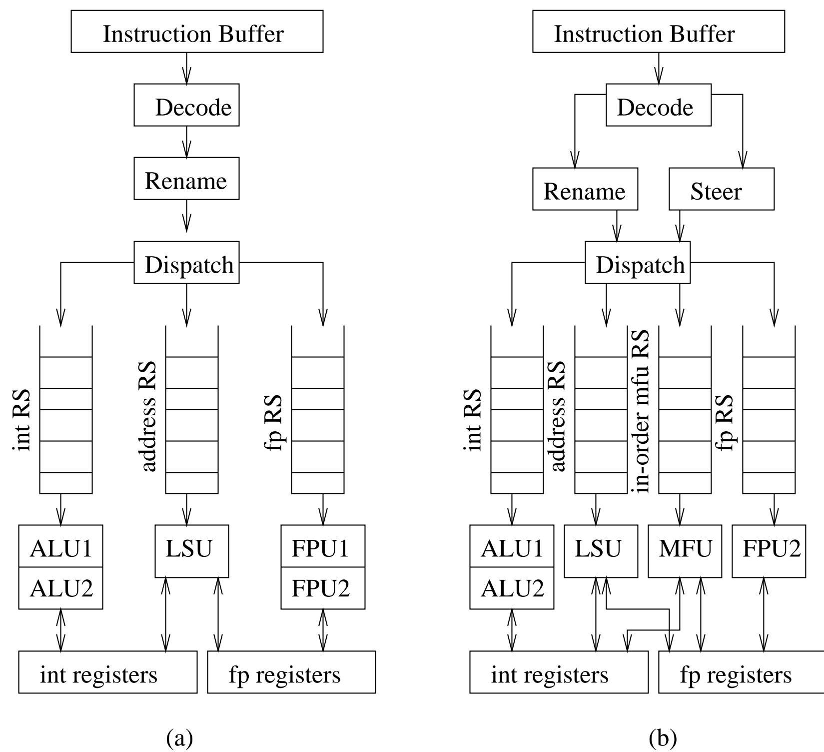

Figure 2: (a) R10000 architecture (b) RS-MFU architecture scheme. Modifications to the R10000 are a steering logic at the dispatch stage, MFU, new reservation station for MFU, and extra integer ports to the integer register file

logic, shift operations, and address generation[3]. 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 in parallel with a steering logic that selects a subset of instructions to be executed by the MFU. The selected instructions are dispatched to a new reservation station (MFU RS) that will issue the instructions only to the MFU, as illustrated in Figure 2b. 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. The steering is performed in parallel with register renaming to avoid any effect on clock frequency.

Since we replace FPU1 (floating-point adder) with an MFU, all floating-point additions have to be performed in the MFU. Floating- point applications have a lot of floating-point additions. Thus, the MFU will serve these instructions most of the time, providing little bandwidth left to serve integer instructions. The MFU may even reduce floating-point addition bandwidth if we overuse it for integer execution. In [11], we have shown large performance improvement for integer applications, ranging from 8-14%, and for floating-point applications, the scheme has successfully avoided this type of performance loss.

We implement the MFU RS to have 8-entries and issue instructions in an in-order manner for reasons fully explained in [11].

The real hardware costs for the RS-MFU architecture modification lie in the steering logic, extra read and write ports in the integer register file and the extra wiring. Providing data paths from both register files (int and fp) to MFU in real implementation may require layout change or multi-cycle read and write from one of the register file to MFU. In terms of die area, the steering logic should cost $< 1\%$ of current R10000 chip die area. We judge these hardware costs to be small, even more so when considering the performance gain for integer applications.

3 Simulated Architecture Modifications

We consider future architecture modifications that have one thing in common: increased ability of the processor to fetch and decode instructions. This type of modifications increases the rate instructions are sent to functional units, increasing the pressure on the functional units to consume instructions faster. We hypothesize that this pressure is associated with IPC gain provided by an MFU (which we prove in Section 4 and analyze in Section 5). We simulate the following architecture modifications:

- **Wider-issue superscalar architecture (8-way, 16-way)**. Suppose that we increase the fetch, decode and commit width to 8- or 16-way, but maintain the functional unit configuration of the base and RS-MFU scheme (as in Figure 2), then we can fetch more instructions per cycle, limited by branch prediction and instruction alignment in the I-cache.
- **Larger on-chip cache (loc)**. This modification reduces the number of instruction cache misses and thus allows the processor to have less vertical waste in the instruction fetch slots.

[3]Note that memory instructions that are sent to MFU RS are also sent to the address RS. MFU only performs the address generation, then passes the results to the address reservation station, which performs the actual loads and stores.

- **Processor-memory integration (pim)** [8, 10]. This modification decreases the time taken to satisfy instruction cache misses, resulting in less vertical waste in the instruction fetch slots.
- **Better branch prediction (pbp)**. Higher accuracy in branch prediction means we can afford to predict more branches per cycle. This allows us to fetch more instructions from several basic blocks. Thus, it reduces horizontal waste in instruction fetch slots, resulting in more instructions fetched per cycle.

In the simulation, we use integer and floating-point applications from Spec95 benchmark [2] plus kmeans [17] as the codes of interest. These include perl, li, ijpeg, compress, swim, su2cor, and wave5. Kmeans is an iterative clustering algorithm. Clustering algorithms are often used in 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. RS-MFU increases integer execution bandwidth thus IPC gain is expected for integer applications. Floating-point applications are included in the study just to show that for all architecture modifications that we simulate, the RS-MFU scheme does not reduce the IPC by any noticeable amount.

The parameters of the simulation are shown in Table 2. Parameters in the architecture modifications that are not listed in the table have the same values with the base R10000. All architecture modifications (8-way, 16-way, pbp loc, pim) are applied to both the base architecture (R10000) and to the RS-MFU scheme, and the two versions are compared to measure the IPC gain and speedup. We chose to implement single architecture modifications one at a time measuring the IPC gain obtained by using RS-MFU scheme over the base without MFU, so that we can isolate the performance effect.

Simplescalar simulator [6] 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. Some of Simplescalar features which are different from R10000 are left unchanged. Specifically, 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 occurs after the execution of a branch, the pipeline is immediately flushed and the fetch is redirected. And finally, some parameters shown in Table 2 are different from the R10000.

4 Simulation Results

Each architecture modification given in Table 2 results in higher IPC because they increase the fetch/decode rate (Figure 3). However, the magnitude of the improvement is code dependent. In integer applications, perfect branch prediction provides the highest IPC boost. Integer applications on average have 1 conditional branch in 5 instructions. Thus, boosting branch prediction accuracy has a high impact on the IPC. In floating-point applications, processor-memory integration (pim) provides the highest IPC boost. This is due to the large working sets in the floating-point applications, which cause large L1 data cache and L2 cache miss rates. Miss rates are low in integer applications for our choice of input set, since the working set fits in the cache. Thus, pim benefits floating-point application performance the most. An anomaly

Table 2: Simulation parameters

Parameters		Values
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 with R10000 latencies
	ROB	64 entries
	Reservation stations	16 entries int, addr, and 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
R10000 with RS-MFU	Functional units	ALU1, ALU2, LSU, MFU, FPU2 (R10000 latencies)
	Reservation stations	16-entry int and addr, 8-entry mfu and fp
Architecture modifications, applied to both base R10000 and R10000 with RS-MFU		
8-way	Fetch, decode, and commit width	8
16-way	Fetch, decode, and commit width	16
Perfect branch prediction (pbp)	Branch prediction	perfect
Large on chip cache (loc)	L1 cache	4-way, 128 KB-I + 4-way, 128 KB-D, 1 cycle hit
Processor-memory integration (pim)	L2 cache	none
	Processor-memory	memory bus 32 bytes, access latency 5 cycles

in our observed codes is kmeans, which while primarily an integer application, has quite many floating-point additions and a large working set. Thus, it benefits both from pim and pbp, as shown in the figure. The IPCs for RS-MFU scheme with each of the architecture modifications are similar to Figure 3, only higher. Now that we understand the relative performance improvement for these types of architectural modifications, we wish to observe whether the RS-MFU implementation will give even higher IPC gains.

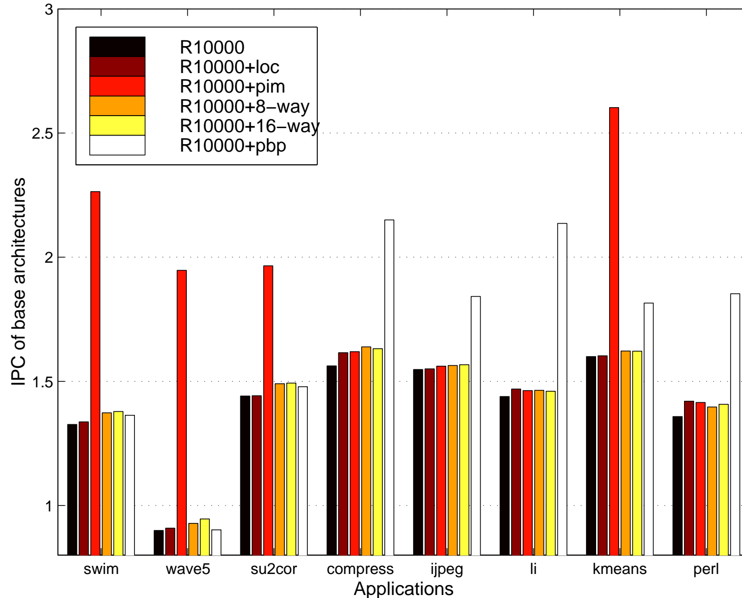

Figure 3: IPC of base architectures: R10000 + its modifications

IPC gain is calculated as the difference in measured IPC values between the schemes (with and without MFU), as shown in Figure 4. For floating-point applications, the MFU provides little benefit since most of the time the MFU has to serve floating-point additions. The resulting potential IPC gain is usually offset by the mutation penalties. Wave5 has been shown to have high mutation frequency, thus in most architecture modifications, the IPCs are reduced, by a small amount. Swim and su2cor provide IPC gain, albeit small.

For integer applications, all IPC gains are positive, indicating that for all architecture modifications and all integer applications tested, the RS-MFU scheme offers better IPC than the base scheme. The question is how the IPC gain varies with architecture modifications. The first bar of each grouping of Figure 4 shows RS-MFU gains in IPC over the base architectures, ranging from 0.13 to 0.23.

With large-on-chip cache applied to the R10000 and RS-MFU (the second bar), the RS-MFU gains more IPC on some applications (compress, li, and perl) ranging from 0.15 to 0.19, the same IPC on kmeans, and loses a little bit of IPC gain on ijpeg. The decrease in IPC gain (0.003) in ijpeg is statistically insignificant.

In processor-memory integration (pim), L2 cache is eliminated, processor-memory bandwidth is increased, and memory access latency is decreased. It results in a faster supply of instructions

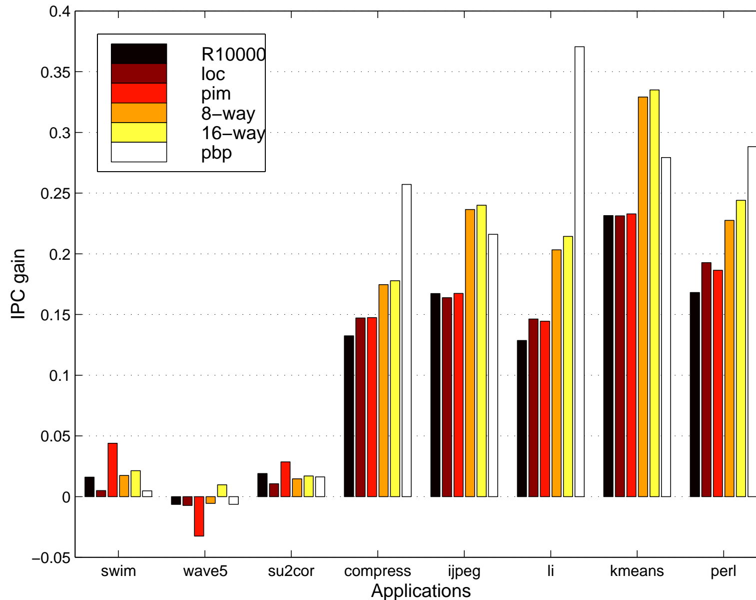

Figure 4: IPC gain of RS-MFU scheme over the base architecture with various architecture modifications. IPC gain = IPC(RS-MFU with mod) - IPC(R10000 with mod).

in the case of instruction cache misses. This gives higher fetch/decode rates for all applications, resulting in higher IPC gain, especially for compress, li, and perl.

With 8-way and 16-way, all applications get much larger IPC gain ranging from 0.17 to 0.34. The increase in fetch/decode rate resulting from a wider fetch is limited by branch prediction and instruction cache line size. Thus, although the 16-way case provides even larger IPC gain than the 8-way case, the increase is not as large as in 8-way versus 4-way [4].

Finally, in perfect branch prediction (pbp), all applications get higher IPC gain with the use of RS-MFU scheme. The increase is very large for compress, li, and perl, but more moderate for ijpeg and kmeans.

Thus, all applications generally enjoy the higher IPC gain provided by the RS-MFU. However, is it also the case with the speedup, i.e. does RS-MFU enhance the speedup of applications on the architecture modifications tested. Speedup increase is harder to obtain than an increase in IPC gain, because increases in IPC gain do not necessarily mean increased speedup. An example is shown in Table 3. The table shows that although IPC gain provided by RS-MFU increases from 0.1 (base) to 0.15 (with architecture modification), the speedup decreases from 1.1 to 1.075. To maintain the speedup, the IPC gain has to be increased by the same factor as the increase of IPC of the base with architecture modification over the base without architecture modification (a factor of two in our example).

Figure 5 gives the resulting speedup. Despite the previous example, amazingly the speedup of integer applications increases whenever IPC gain increases (compared to Figure 4). Out of all

[4]These improvements may be artificially inflated since we don't change the number of functional units as done in [9]. We believe this underscores the potential use of MFUs (as is our goal) by giving larger execution bandwidth per functional unit.

Table 3: IPC gain versus speedup example

Modification	Base	RS-MFU	IPC Gain	Speedup
R10000 (no mod)	1	1.1	0.1	1.1
with mod	2	2.15	0.15	1.075

architecture modifications, wider fetch (8 and 16-way) increases the speedup the most, followed by perfect branch prediction (pbp) and large-on-chip cache (loc). Processor in memory (pim) offers the least increase in speedup. To summarize, for all architecture modifications, RS-MFU provides speedup ranging from 8% to 21%, a significant speedup considering the small hardware cost mentioned. Floating-point applications enjoy a very small positive or negative speedup (within $\pm$ 2%) for all cases.

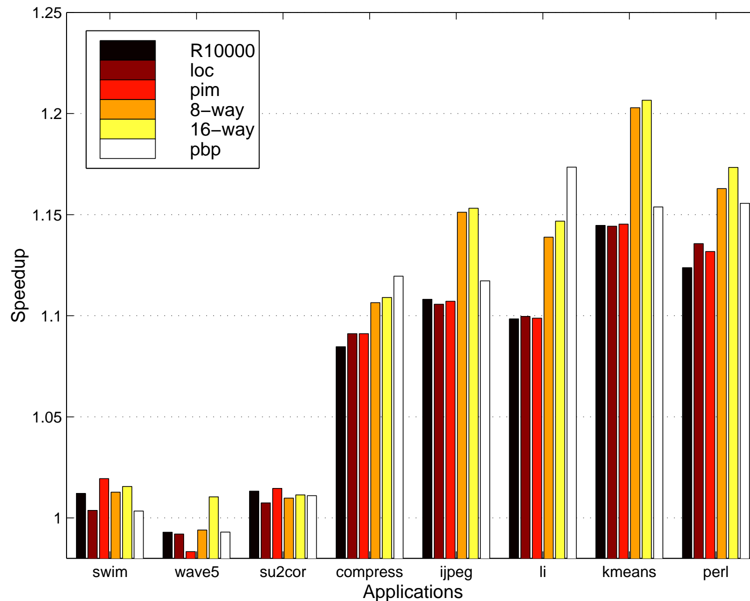

Figure 5: Speedup of RS-MFU scheme over the base architecture with various architecture modifications. Speedup = IPC(RS-MFU with mod) / IPC(R10000 with mod).

5 Analysis and Discussions

5.1 Rates Analysis

The potential IPC gain provided by the RS-MFU is related to the stall time due to unavailable functional units. This type of stall is caused when the speed capability of the processor to fetch/decode[5] instructions exceeds the speed capability of the processor to execute the instruc-

[5] including rename and dispatch.

tions. It is natural to think of the processor as having a front-end (fetch and decode stages, with rate f) and a back-end (issue and execute stages, with rate e). At a given time during the execution, we may see the cases as shown in Table 4.

Table 4: Behavior of Fetch/Decode Rate (f) and Issue/Execute Rate (e) at Run-time

Case	Probability	Caused by
$f < e$	$1 - \alpha$	I-cache/tlb number and latency of misses, branch prediction limitation, non-optimal instruction alignment in cache, fetch width limitation
$f \geq e$	α	unavailable functional units, data dependency

An architecture modification that reduces the items listed in the "Caused by" column in Table 4 will decrease the associated probability. For example, reducing instruction cache misses increases the probability (α) of $f \geq e$. When α increases, instructions are dispatched into the reservation stations at a faster rate (up to the point where reservation stations are full, and we have $f = e$). This increases the stall time of an instruction due to unavailable functional units. Thus, larger values of α indicate higher potential IPC gain that can be provided by the MFU.

All architecture modifications that we experiment with in Section 4 increase α in one of the following ways: 8-way and 16-way fetch increases the fetch width; large on-chip cache reduces the number of I-cache misses; processor-memory integration reduces latency of cache misses; and finally perfect branch prediction solves the limitation in fetch rate imposed by branch predictor. Consequently, the IPC gain obtained by the RS-MFU is generally higher for those architecture modifications.

In real processor implementation, there will be many architectural modifications to improve IPC combined together. Although it will be difficult to determine the effect of combined architecture modifications on IPC gain provided by RS-MFU, as long as we can analyze how this combination affects α, we can predict whether the IPC gain of RS-MFU will increase or decrease, although currently there is no model to predict the magnitude of the increase/decrease.

5.2 Theoretical Foundation

f and e are hard to quantify in real processors. However, they can be quantified with some assumptions, as in [1], which we will discuss briefly here. The approach is based on queuing theory and provides the stall time due to unavailable functional units in IPC.

We are interested in the relationship between fetch/decode and issue/execute rates as architecture modifications are made. The system can be modeled as a simple queuing system where customers are instructions, servers are functional units, customers arrive at the fetch/decode rate, and customer are serviced at the issue/execute rate. Now, let us introduce some variables to allow definition of our queuing system. Full details of this analysis technique are discussed in [1].

$\bar{t}$	average inter-arrival distance between instructions (quantity without unit)
$\bar{x}$	average service time in cycles for each instruction
β_{eff}	effective number of instructions fetched and decoded per cycle. While queuing model allows us to limit the reservation station size of the buffer between arrival and service, in this discussion we assume an infinite size reservation stations to decouple β_{eff} from m_{eff} for more straightforward analysis.
m_{eff}	effective number of instructions that can be executed per cycle by the functional units.

Fetch/decode rate is thus calculated as $f = \frac{\beta_{eff}}{\bar{t}}$ where β_{eff} is architecture and code dependent and $\bar{t}$ is measured from the code. Issue/execute rate is calculated as $e = \frac{m_{eff}}{\bar{x}}$, where both m_{eff} and $\bar{x}$ are architecture and code dependent.

All of $\bar{t}$, $\bar{x}$, β_{eff}, m_{eff} can be calculated if synthetic codes with uniform instruction streams are used, allowing for direct validation of the model. Synthetic codes allow us to control the number of instruction cache misses, data cache miss effect, number and location of conditional branches, instruction alignment, and data dependency. Thus, the non-overlapped stall time due to unavailable functional units can be calculated because all parameters are known. And this gives indication of how much IPC gain is to be expected from RS-MFU scheme for the various architecture modifications. While quantitative analysis can be provided using this technique on synthetic codes, using this model in the context of the previous subsection provides insight to RS-MFU performance on real codes.

5.3 Other Architectures and Other Benefits of MFUs

Other architectures that can potentially increase the IPC gain provided by the MFU are:

- **Simultaneous multithreading (SMT)**. Since we can fill instruction slots with instructions from different threads, it is more likely that all slots are filled with instructions. Thus, it increases fetch/decode rate.
- **Trace cache**. Trace cache allows fetching traces instead of instructions. In the original design, a trace cache can supply up to 16 instructions or 3 basic blocks [7] per fetch, providing much more instructions per cycle than the traditional instruction cache-only approach. Thus, fetch/decode rate increases.
- **Superspeculative architecture**. Predicting load values allow instructions to be dispatched without waiting for their operands [14]. Although this architecture does not necessarily increase fetch rate, it can increase decode rate[6], resulting in more instructions ready to execute per cycle.

In addition to potential IPC gain increase in the architecture modifications listed above, there are other benefits of MFU in SMT and trace processor architecture.

In a six-issue superspeculative trace processor of Hal SparcV 64 [4], the MFU enhances the packet construction flexibility and consequently average packet size. In their design, a packet in the trace cache is broken when the instructions in the packet do not match the functional unit

[6] more precisely, dispatch and issue rates also increase, but execute rate is limited by the functional unit configuration.

configuration. Converting the 2 FPUs to MFUs will provide flexibility in packet construction, for example it can now have any combination of 2 fp + 2 int, 1 fp + 3 int, and 0 fp + 4 int instructions, which traditionally requires breaking the packet.

In the design of future 8-way SMT Alpha 21464 [3], instructions from up to 4 threads can share functional units. It is likely that functional units are clustered because of the high clock rate requirement. With the MFUs providing higher integer execution bandwidth while maintaining the same number of functional units, it may reduce the number of clusters needed to serve the 4 threads, resulting in lower inter-cluster communication penalties.

Judging from these IPC gain benefits and other benefits, we believe that MFUs will even be more important in future processor architectures.

6 Conclusions

We have described briefly the hardware design of a mutable functional unit and an architecture scheme (RS-MFU) with small hardware cost that exploits the MFU to speedup integer applications. We have shown that many architecture modifications expected in the future increase IPC gain provided by the MFU. Our study also points to the increase in speedup with the architecture modifications. We show the results for 7 Spec95 applications (perl, ijpeg, li, compress, swim, su2cor, wave5) and kmeans with architecture modifications such as large on-chip cache, processor-memory integration, wider issue superscalar architecture (8 and 16-way), and better branch prediction (perfect). We also point to other future architecture modifications (SMT, trace cache, and superspeculative architecture) and how IPC gain provided by the MFU may increase in these architectures. In addition, we discuss other benefits of MFUs.

References

[1] Kirk W. Cameron and Yong Luo. Instruction level modeling of scientific applications. *Proceedings of the ISHPC 99*, May 1999.

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

[3] Keith Diefendorff. Compaq chooses smt for alpha: Simultaneous multithreading exploits instruction- and thread-level parallelism. *Microprocessor Report 13(16)*, Dec 6, 1999.

[4] Keith Diefendorff. Hal makes sparc fly: Sparc64 employs trace cache and superspeculation for high ilp. *Microprocessor Report 13(15)*, Nov 15, 1999.

[5] Dominique Lavenier, Yan Solihin, and Kirk W. Cameron. Integer/floating-point reconfigurable alu. *Unclassified Technical Report LA-UR 99-5535, Los Alamos National Laboratory*, Sep 1999.

[6] 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.

[7] Eric Rotenberg, Steve Bennett, J.E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. *29 Annual International Symposium on Microarchitecture*, Dec 1996.

[8] David Patterson et. al. A case for intelligent ram: Iram. *IEEE Micro*, April 1997.

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

[10] Yi Kang et. al. Flexram: Toward an advanced intelligent memory system. *International Conference on Computer Design*, Oct 1999.

[11] Yan Solihin et.al. Reservation station architecture for mutable functional unit usage in superscalar processors. *Unclassified Technical Report LA-UR 99-6234, Los Alamos National Laboratory*, 1999.

[12] John L. Hennessy and David A. Patterson. Computer architecture: a quantitative approach. *Morgan Kaufmann Publisher inc., 2nd ed*, page 341, 1996.

[13] Kirk W. Cameron, Yan Solihin, Yong Luo. Workload characterization via instruction clustering analysis. *in preparationf or Unclassified Technical Report LA-UR, Los Alamos National Laboratory*, 1999.

[14] Mikko H. Lipasti and John Paul Shen. Superspeculative microarchitecture for beyond ad 2000. *IEEE Micro*, pages 59–66, Sep 1997.

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

[16] Subbarao Palacharla and J.E. Smith. Complexity-effective superscalar processors. *The 27th Annual Internationa Symposium on Computer Architecture*, 1997.

[17] 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.