Managing Interconnect Delay With Architectural and Compiler Techniques

Walt Fish, Chris Flesher, David Suksumrit, Allen Wan,

Abstract

Interconnect delay is becoming an increasingly dominant constraint in modern processor design. Already, several modern processors require extra pipeline stages to account for interconnect delay, and a signal crossing the entire chip can require several cycles to propagate. Until recently, the interconnect delays between ALUs and the register file were dwarfed by logic delay. However, techniques in managing delay due to interconnects will become more crucial as technology scaling causes interconnect delay to account for an increasing portion of functional unit execution time.

We propose to address the problem of interconnect delay through the use of register bank/ALU clusters, created by partitioning the register file into separate banks, each associated with a nearby functional units. This means that instructions whose operands are stored in registers adjacent to their intended functional unit do not suffer additional interconnect delay due to long propagation distance, while instructions whose operands are in a separate cluster will suffer a longer interconnect delay penalty. We further propose to create compiler optimizations to ensure that operands produced and consumed by functional units will be in registers close to the local ALU cluster whenever possible, thus ensuring that a minimum of instructions will have the penalty of the longer, inter-cluster communication delay.

I. Introduction

As technology features scale in size, the delay due to logic in gates improves by a factor relative to the feature scaling, but the delay due to interconnect decreases only slightly. Gate delay is due to the width of the transistors involved, and improves by a factor relative to feature scaling. Interconnect delay is modeled by an RC time constant, with R being the resistance of a wire and C the lumped coupling capacitance with other metal features. As features scale downward in size, C improves by a factor of the scaling, but making wires smaller increases R by the factor of the scaling. This effect causes interconnect delay to have poor improvement relative to the rest of technology. With modern superscalar processors attempting faster clock speeds, the portion of the chip that a signal can travel across via interconnect decreases. With current organizations of the register file and execution units, we will soon reach a point where any operation involving the register file will require an additional clock cycle for signal transfer. The limiting factor in such a case would be the delay of the worst-case scenario in which operands must be transmitted between registers located on the edges of the register file and an ALU that is physically far away.

As interconnect delay has grown relative to logic delay, circuit designers have had to add extra pipeline stages to allow for data to travel between the register file and ALUs. At 250nm, logic delay equals interconnect delay as a source of latency in circuit design[1]. Requiring an extra cycle for operations or adding another pipeline stage could be seen as a step backward in superscalar processing.

In a 2002 update of the International Technology Roadmap for Semiconductors (ITRS), interconnect delay was identified as an area where "design and layout solutions are needed"[2]. One emerging technique for addressing this hurdle is the notion of exposing elements previously hidden by the ISA[3]. Exposing these elements gives compilers and programmers the ability to explicitly account for and manage these obstacles. Our approach extends this concept by proposing a new architecture combined with compiler optimizations that exploit low-level manipulation of system latencies, allowing faster execution time.

Our paper addresses a hypothetical situation where interconnect delay has grown to the point where an integer operation can take several cycles due to interconnect delay. Current superscalar techniques such as Tomasulo's algorithm[4], reorder buffer and multiple ALUs will not be able to hide the increased delays without further advances. In order to prevent this performance degradation, we need to limit the total amount of wire used by the path of an ALU instruction. To achieve this end, we intend to place ALUs and reservation stations closer to the register file by partitioning the file into smaller pieces, each with its own dedicated ALUs. Data within a cluster can originate from the register file, undergo computation and be written back to the register file in one cycle. To ensure that the maximum portion of instructions remain inside one cluster and do not suffer the penalty of interconnect delay due to inter-cluster communication, we propose a compiler optimization which attempts to reassign all source and destination operands by their location into one of the clusters.

Motivation

As interconnect delay has increased in relation to logic delay, the time that it takes for data to travel from the registers to the ALU have risen proportionally to become equal or greater than the time required to operate on the data in the ALUs. Since variations in this propagation delay are not exposed, a processor must allow for the worst case delay on all ALU operations even if the operands are coming from the physically closest registers. We propose to separate the register file into banks, each with its associated functional unit so that only a portion of total instructions must suffer the delay from operands being communicated from one cluster to another. When an instruction has both source and destination operands within one bank, the instruction can complete significantly faster in its associated ALU than in the case of an architecture where operations have to take into account the worst-case delay. If an instruction uses operands from different banks, it will have more delay than the optimal case due to longer wire paths for inter-bank communication from the registers to the ALUs.

We examined other architectural features to determine whether or not they had a large effect on solving the problem of interconnect delay. We analyzed various configurations for the SimpleScalar Register Update Unit (RUU). As can be seen in figure1, as we increased RUU size for vpr (a SPECint2000 benchmark), there was very little performance benefit. Even after increasing the RUU size until the RUU was never full, we still saw very little benefit. At the same time, increasing ALU latency caused a linear increase in execution time. This shows that increasing interconnect latency cannot be hidden by the traditional methods of using reservation stations or large reorder buffers.

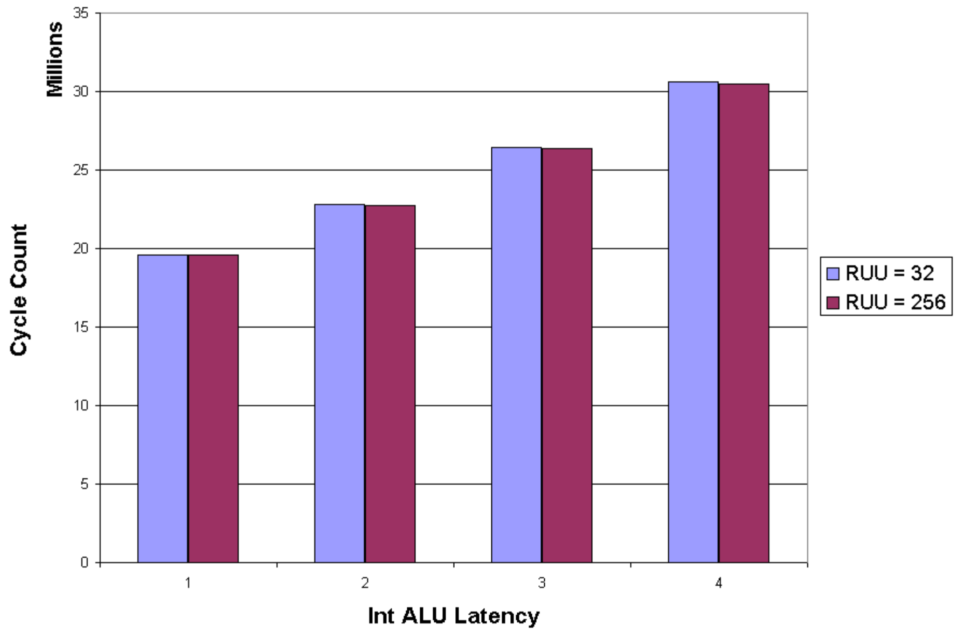

Figure 1.
Effects of RUU size on execution time

Hypothesis

We believe that we can address the issue of increasingly dominant interconnect delay by statically scheduling instructions in ALUs that are physically closer to their source and destination registers.

We employed the SimpleScalar tool set[5] for our project due to its flexibility and extensibility[6]. This allowed us to simulate architecture changes in the partitioning of the register file and ALUs as well as simulating new instructions in a large superscalar machine via SimpleScalar annotations[7]. Annotations are useful for synthesizing new instructions without having to change and recompile the assembler. This allows us to modify the simulator and 'tag' instructions based on their predicted delay. These annotations allow us to examine register usage and assign each instruction a 'delay class' based on which banks an instruction's source and destination registers involve. We also used the gcc compiler that came with SimpleScalar (ss-gcc) in order to output the assembly code that we analyzed and optimized.

II. Architecture

Register files are traditionally one piece of unified memory, with several access ports. Every data transfer between an ALU and the register file must allow enough time for the propagation between the farthest ALU and register combination.

Our proposed banked register file with close functional units is illustrated in Figure 2. The banking of the file is chosen such that ALU instructions that have source and destination operands entirely within one bank will have a one-cycle delay. If an instruction has operands involving banks farther away, there is a one-cycle penalty for each 'bank' of distance between the farthest two operands.

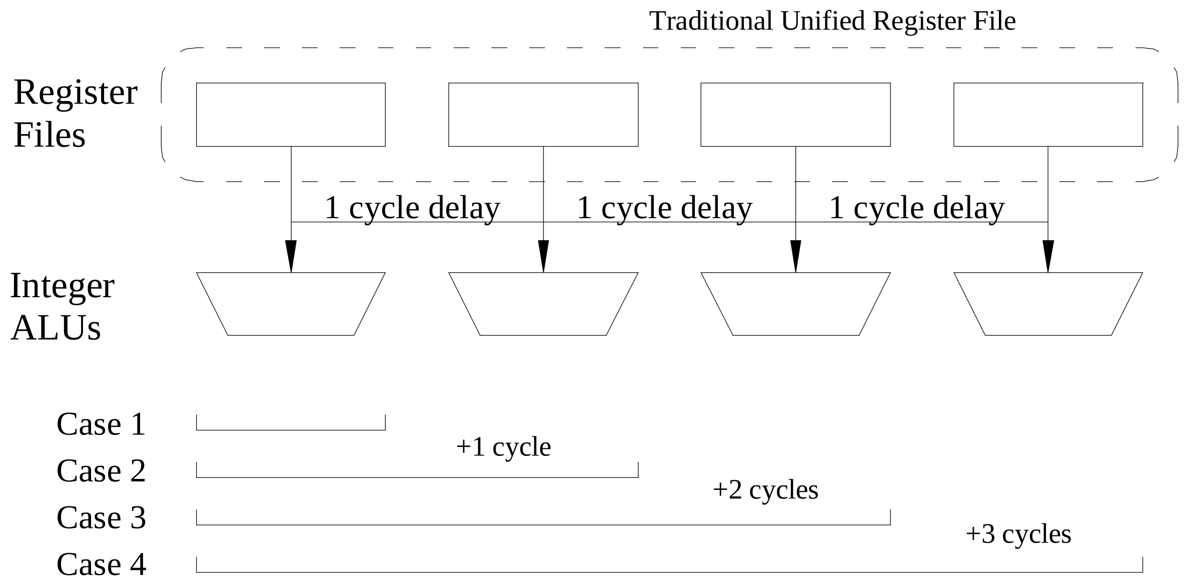

Figure 2.
Banking of a register file to exploit physical proximity to functional units. Different cases illustrate the delay associated with instructions communicating between distant banks for operands.

We used many of the default architecture features in SimpleScalar. The most important change is the banking of the original 32-entry register file. We separated the file into four banks (one bank for each ALU). The registers are divided evenly between banks, with eight registers per bank plus a local copy of the stack pointer and frame pointer because these two values are accessed often. Since they occur in every register bank, if the stack pointer or frame pointer is written to, these operations will suffer the worst-case penalty as the values are propagated throughout all banks. The duplicated registers are read from far more often than they are written to, so their duplication results in a net improvement in execution time.

We chose to use a perfect branch predictor because the small size of our micro-benchmarks allowed the two-level branch predictor insufficient time to warm up. This caused an unusually low prediction accuracy which we believe is not representative of true scientific code. Using the two-level branch predictor would have reduced the improvement by only about one percent for most of the benchmarks. Furthermore, we used a 128-entry reorder buffer which never completely filled, and we experienced extremely low cache misses due to the recursive nature of our benchmarks.

III. Experimental Methodology

To model the different interconnect delays, we first modified SimpleScalar to execute integer instructions with specified latencies. As SimpleScalar parses through a program's machine code, it assigns a class to each instruction that specifies the latency and execution time of that type of instruction. To model the interconnect delays associated with receiving data from registers at different distances, we created four types of integer ALU classes, each with a different latency.

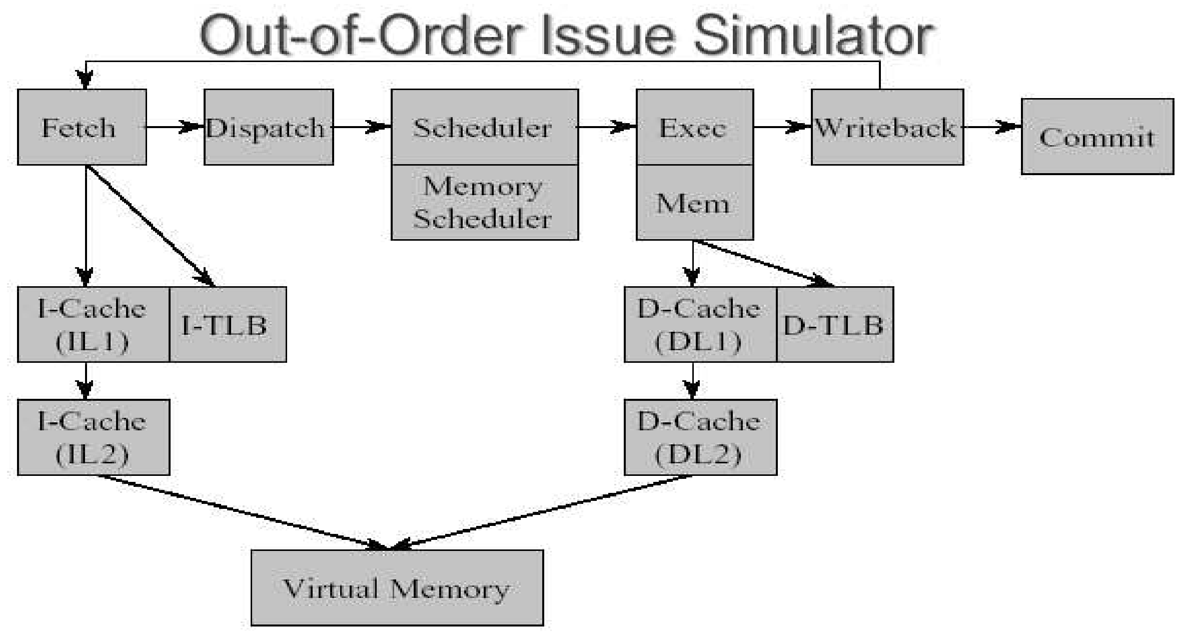

Figure 3.
Block diagram of the SimpleScalar simulator [taken from SS Hacker's Guide[7]]

For integer instructions, our modified version of SimpleScalar reassigns the instructions class before issuing. Annotations inserted into the code indicate which class the simulator should reassign to the instruction.

Because testing involved hand-reordering and annotating assembly code, small snippets of scientific code, each a few hundred lines, were used as micro-benchmarks. Arguably, if the code can be optimized by hand, a compiler that accounts for interconnect delays specified in the ISA can optimize performance as well. We decided to use mathematically-intensive C code that might be used in the scientific community as our micro-benchmarks.

Benchmark	Purpose
mm.c	Matrix Manipulation
gray.c	Gray Coding
spigot.c	Calculating Pi
sub_lex.c	Lexiographic Ordering
perm.c	“Short and Bewildering Recursive Method”

Table 1.
Microbenchmarks used for simulation

For our base architecture, we executed the micro-benchmarks without any annotations. When our simulator executes an instruction without an annotation, operands for that instruction take the worst-case number of cycles to arrive at the functional unit. These simulations represent a traditional architecture that issues to its functional units according to the worst case delay.

For the second set of simulations, each instruction was annotated according to the spatial proximity of the operands assigned by the compiler. For example, if an instruction has its operands in the same bank, then the instruction is assigned the shortest latency. If an instruction contains operands in separate banks, the annotation reflects the time necessary for both instructions to arrive at the functional unit. This arguably is analogous to creating hardware to determine dynamically the necessary delay for instructions to arrive at their functional units.

The third set of simulations involved moving operands to different registers and reordering instructions to achieve the best gain from exposing the interconnect delays to the ISA. To achieve this performance, operands were moved to different banks to maximize the number of instructions that use registers from one bank. The same code transformations should be possible with an ISA and a compiler that account for interconnect delay.

Although our experiments involved only integer operations, we annotated load and store operations as well. When the assembler translates a memory access instruction into machine code, it converts the instruction into an integer operation to compute the memory address and a memory operation to access that address.

```
sll/b    $9,$15,2
lw/a     $15,32($fp)
addu/b   $9,$9,$15
lw/a     $9,0($9)
lw/a     $16,16($fp)
sw/b     $9,0($2)
sll/b    $8,$16,2
lw/a     $15,32($fp)
lw/a     $16,16($fp)
addu/b   $8,$8,$15
addu/a   $15,$16,1
sw/b     $15,0($8)
j        $L6
```

IV. Experimental Analysis

Our experimental results show that we get a significant improvement through banking the register file and attempting to statically schedule the operands of individual instructions into the same bank. Figure 4 displays the normalized execution results for each of our five micro-benchmarks.

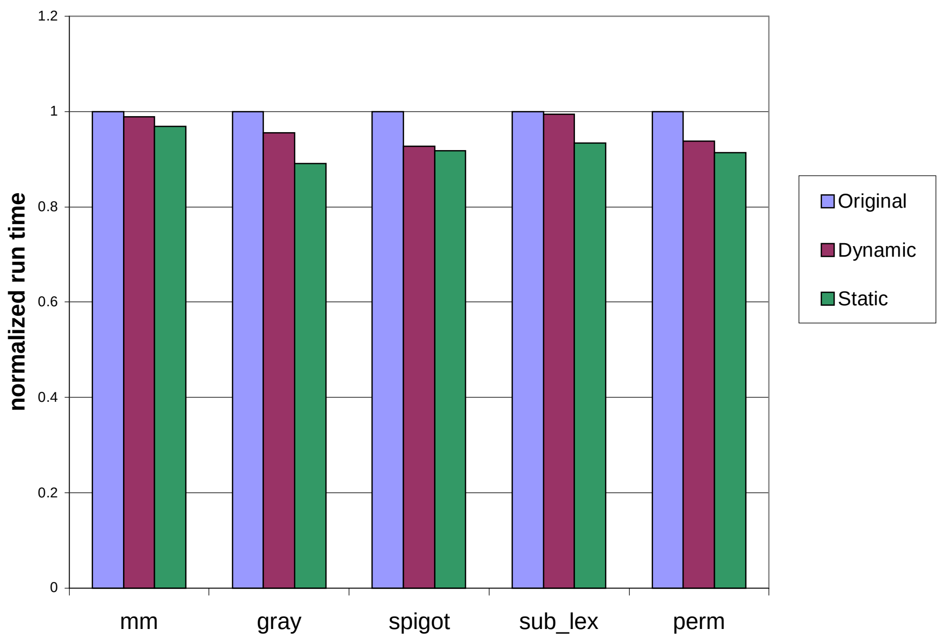

Figure 4.
Relative execution times for micro-benchmarks

The first bar for each benchmark represents the original execution time of the benchmark as compiled and assembled using SimpleScalar' s version of gcc. For the second bar of each benchmark, each assembly instruction was annotated with its corresponding delay using the rules that we have described above. No reordering or re-banking was done to the assembly to obtain the second result. We believe that this is a fair representation of the performance increase that can be obtained by dynamically annotating the assembly instructions as they arrive at the processor, since it would be fairly easy to implement a register locality annotation scheme in hardware. The third bar for each benchmark represents what a complier should be able to achieve by statically renaming registers so as to minimize the distance between an ALU and its source and destination registers. Since a compiler is able to look at a much longer series of instructions, it should be able to group values that it knows will be used together in the future into the same register bank. It is worth noting here that by long series of instructions does not mean thousands of instructions. Doing the re-banking by hand, we were only able to look at twenty to thirty instructions at a time, so we believe that a complier will be able to achieve at least comparable results.

Execution Cycles	mm	gray	spigot	sub_lex	perm	
Original	1234159	3103385	4314160	2372470	1923770	
Dynamic	1219670	2963033	4000905	2359729	1804523	
Static	1195202	2764572	3957655	2216297	1757949	

Normalized Run Time(percent)	mm	gray	spigot	sub_lex	perm	Average
Dynamic	98.8	95.5	92.7	99.4	93.8	96.1
Static	96.8	89.1	91.7	93.4	91.3	92.5

Percent Speedup	mm	gray	spigot	sub_lex	perm	Average
Dynamic	1.17	4.52	7.26	0.54	6.20	3.94
Static	3.16	10.92	8.26	6.58	8.62	7.51

Table 2.
Performance results

Our results tell a multi-level story. In Table 2, we can see that with a 7.5% average performance increase, the static re-banking technique provides good improvement for relatively little compiler effort and basically no effort by the hardware. This improvement is achieved not through some overly complex technique, but simply by exposing a developing problem and having the compiler recognize and address the problem. The second level to our results is the average 4% speedup seen by simply dynamically annotating the instructions as they are produced by gcc. This is a surprising good result considering that it does not require a change in the ISA and can be done with very little hardware.

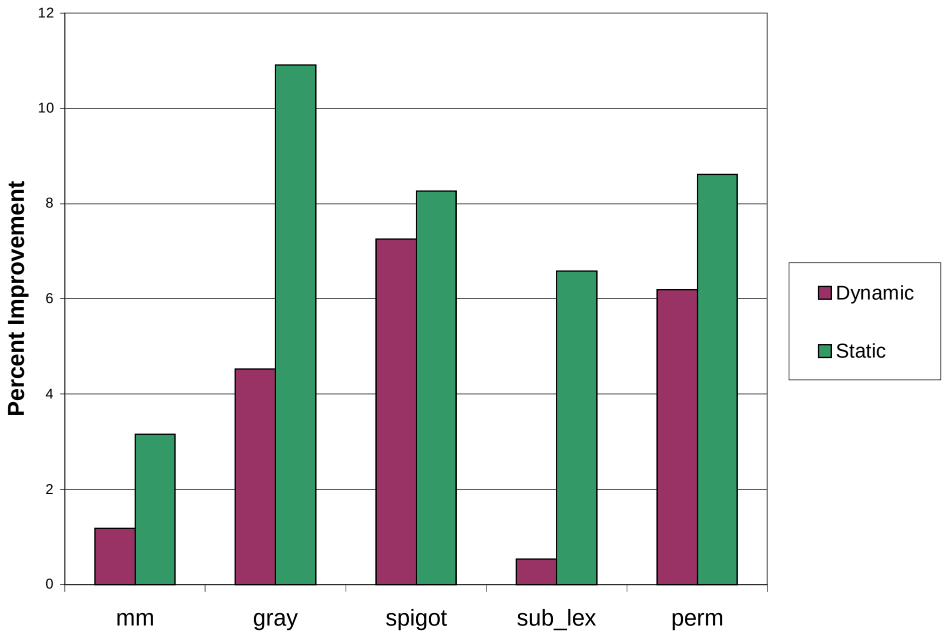

Figure 5.
Percent speedup performance results

We feel that the static method shows promise as a technique for addressing the issue of increasing interconnect delay, and that it is better overall than the dynamic method of optimizing for physical proximity between registers and ALUs. There are however some individual exceptions. In the performance results for spigot, we see in figure 5 that there are situations where dynamic banking achieves the majority of the performance benefit that can be obtained. Dynamic banking is able to obtain such large portion of the potential gain because of the way that gcc uses registers. Gcc assigns registers in increasing numerical order. If a program uses very few registers, then they will tend to all be in the first bank. If this was the only program that was used, then a dynamic banking method might be considered preferable since it would alleviate the need to change the ISA and thus the compiler. The dynamic method can also be incorporated into the existing register renaming techniques used by modern processors. However, in the case of sub_lex, we see an example where dynamic banking does very little. In this case, our static method is able to create more register locality by preventing values that will later be used locally from being stored into farther registers. By taking this relatively easy step, we can obtain 12 times more improvement over the dynamic method. Even with the constraints that static banking might pose, if sub_lex were an accurate representation of the code that is run on a machine, it would likely be worth the extra trouble for a 6.5% improvement.

Another interesting result to note is the relatively poor improvement seen in mm. This is due to the assembled code in mm having smaller amounts of code between each “function/jump”. We did not try to schedule register through a segment jump since there are compiler conventions as to the registers used to pass each value. Thus in general, there were fewer instructions that could be effectively banked and also fewer instructions with registers naturally occurring in the same bank. As a result, both the dynamic and static methods suffered. This is in contrast to gray which has a single large segment that it loops through. The large segment allowed for a larger window of instructions in which we could reorder. This property greatly benefited the static method, which was able to obtain an 11% improvement on gray. The dynamic method also benefits from large continuous code segments, but once the code segments become too large and the compiler starts to use more registers, the amount of naturally occurring register locality may actually decrease. This explains why the dynamic method actually did better on perm than gray whereas the static method achieved a larger performance increase on gray than it did on perm.

For the latter three benchmarks (mm, gray, perm), we see a rather consistent and probably more representative trend. In every case there is good speedup in the dynamic case. With the static case, however, there is a substantial additional speedup. We believe that this justifies our claim that implementing the static method is worth the additional hardware constraints. This is especially true in context of the current trend toward exposing more of the hardware architecture to the ISA/compiler. We recognize that static banking has potential implementation problems if used in conjunction with register renaming hardware. We believe that since this is a forward looking paper, it is fair for us to assume that the current trend towards exposing more of the hardware architecture will continue and thus the requirement for the static banking method that the register naming be exposed is a reasonable one.

Although we limited the scope of our project to integer applications, there is no inherent reason that this technique would not also provide benefits for floating point operations. It might be interesting for a future group to explore the benefits and challenges of applying a register banking technique to floating-point instructions and ALUs.

V. Conclusion

We have demonstrated that in a future architecture where delay due to interconnect composes a significant portion of functional unit execution time, techniques to manage the use of interconnect are needed. By banking the register file, combined with static optimization, we have achieved an average of 7.51 percent speedup. Techniques like this can be useful in eliminating extra pipeline stages added simply for interconnect delay. These techniques will only become more important in the future as feature scaling increases the relative importance of interconnect delay.

References

[1] Y. Massoud, ELEC 521: Modeling and Design of High Speed Integrated Circuits, Rice University, Fall 2003.

[2] International Technology Roadmap for Semiconductors 2002 Update.

[3] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal, " The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs", *IEEE Micro*, March/April 2002.

[4] R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal, V 11, Jan 1967.

[5] *The SimpleScalar Tool Set, Version 3.0*

[6] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The Simplescalar Tool Set. Technical Report CS-TR-1996-1308, 1996.

[7] Todd Austin, SimpleScalar Hacker's Guide, SimpleScalar LLC.

Appendix

Instruction Fetch Queue Size	-fetch:ifqsize	4 Instructions
Branch Misprediction Latency	-fetch:mplat	3 Cycles
Front-End Speed Relative to Core	-fetch:speed	1
Branch Predictor Type	-brpred	Perfect/2lev
Bimodal Predictor BTB Size	-brpred:bimod	2048
2-Level Predictor Config	-brpred:2lev	1 1024 8 0
BTB config	-bpred:btb	512 4
Instruction Decode	-decode:width	4 per Cycle
Instruction Issue Width	-issue:width	4 per Cycle
In-order Issue	-issue:inorder	False
Issue Instructions Down Wrong Execution Paths	-issue:wrongpath	False
Instruction Commit	-commit:width	4
Register Update Unit Size	-ruu:size	32/128/256
Load/Store Queue Size	-lsq:size	16
L1 Data Cache Config	-cache:dl1	dl1:128 32 4 1
L1 Data Cache Hit Latency	-cache:dl1lat	1 Cycle
L2 Data Cache Config	-cache:dl2	ul2:1024 64 4 1
L2 Data Cache Hit Latency	-cache:dl2lat	6
Flush Caches on System Calls	-cache:flush	False
Convert 64 Bit Addresses to 32 Bit	-cache:icompress	False
Memory Access Latency <first> <rest>	-mem:lat	36 2
Memory Access Bus Width	-mem:width	8 Bytes
Instruction TLB Configuration	-tlb:itlb	itlb:16:4096:4:1
Data TLB Configuration	-tlb:dtlb	Dtlb:32:4096:4:1
Instruction/Data TLB Miss Latency	-tlb:lat	40 Cycles
Number of Int ALUs	-res:ialu	4
Number of Int Multipliers	-res:imult	1
Number o fMemory Ports	-res:memport	2
Number of Floating Point ALU	-res:fpalu	4
Number of Floating Point Multiplier/Dividers	-res:fpmult	1
Operate in Backward-Compatible Bugs Mode	-bugcompat	false

SimpleScalar Configuration Settings