Strategies for Achieving Improved Processor Throughput

Matthew K. Farrens

Computer Science Division
University of California, Davis
Davis, CA 95616
(farrens@cs.ucdavis.edu)

Andrew R. Pleszkun

Department of Electrical and
Computer Engineering
University of Colorado-Boulder
Boulder, CO 80309-0425
(arp@tosca.colorado.edu)

ABSTRACT

Deeply pipelined processors have relatively low issue rates due to dependencies between instructions. In this paper we examine the possibility of interleaving a second stream of instructions into the pipeline, which would issue instructions during the cycles the first stream was unable to. Such an interleaving has the potential to significantly increase the throughput of a processor without seriously imparing the execution of either process. We propose a dynamic interleaving of at most 2 instructions streams, which share the the pipelined functional units of a machine. To support the interleaving of 2 instruction streams a number of interleaving policies are described and discused. Finally, the amount of improvement in processor throughput is evaluated by simulating the interleaving policies for several machine variants.

1. Introduction

An important metric for evaluating processor performance, especially in a multiprocessing context, is the throughput rate of a processor (defined as how quickly a processor can execute a set of independent tasks, or $\frac{\text{Number of Processes Executed}}{\text{Unit Time}}$). In order to increase the throughput of a processor, the number of processes that can complete in a fixed amount of time must be increased. There are at least two distinct methods to accomplish such a goal.

This work was supported by National Science Foundation Grants CCR-8706722 and CCR-90-11535.

One method is to decrease the time each process takes to complete. For example, given two processes P_A and P_B, that take time T_A and T_B to execute, the total amount of time necessary to execute the two programs sequentially is $T_T = T_A + T_B$. By employing performance-enhancing techniques like pipelining, T_T can be significantly reduced; and the deeper the pipeline, the higher the peak execution rate of a machine. Unfortunately, as pipeline depth increases, dependencies between instructions begin to impact performance. *Pipeline Hazards* such as Read After Write (RAW), Register Busy, and Result Bus Busy prevent a new instruction from beginning (*issuing*) on each clock. In the CRAY-1, for example, the instruction issue rate is below 45% for the Lawrence Livermore Loops [PlSo88]. This issue rate may be treated as an upper bound on the machine's performance since, due to their nature, one would expect excellent performance on the Lawrence Livermore Loops.

Another approach to decreasing T_T is to use a "superscalar" design (which supports the issue of more than one instruction per clock). However, as pointed out by Jouppi and Wall [JoWa89], the same dependencies between instructions that affect pipelined machines also affect superscalar machines. According to their study, machines with a high degree of pipelining provide virtually identical performance to machines that are superscalar to the same degree. Therefore, in this paper we will restrict ourselves to a deeply pipelined machine model, with the understanding that the same type of results should be expected from an equal-degree superscalar machine.

The low issue rate due to dependencies between instructions in a high-performance deeply-pipelined machine implies that it may be possible to significantly increase the throughput of a processor without affecting either T_A or T_B. This can be done by multiplexing the hardware between the two processors instead of executing the tasks sequentially. In this paper we investigate the improvement in throughput that can be achieved by having two independent processes, P_A and P_B, executing in parallel on the same processor framework. In this scheme, during each clock cycle, the instruction at the head of each instruction stream is decoded, and the issue logic determines which instruction can begin execution. The issue logic then allows one of the two instructions to begin execution and blocks the other. This scheme is attractive because most of the processor hardware is shared (all of the functional units, for example). The main expense is the replication of the register files and the issue/decode logic.

Even with two independent streams available to chose from, it may still not be possible to issue an instruction each clock cycle. This is due to the inherent data dependencies that exist within each stream, as well as hazards imposed by having to share hardware. The only way to determine the inter-related effects of these dependencies is to perform a detailed simulation of such a system. We will show in this paper that, for the benchmarks chosen, we can come quite close to reducing the total execution time of two processes P_A and P_B from T_T to the ideal of $\frac{T_T}{2}$.

2. Background

The basis for this idea can be traced back to the I/O unit of the CDC 6600 [Thor70]. Because of the slow speed of I/O devices, the CDC 6600 I/O unit was configured as a set of 10 independent peripheral processors sharing a 10-stage pipelined central hardware core. The clock cycle for each of the peripheral processors was 10 times that of the central core, and in a round-robin fashion on each core clock cycle a different peripheral processor was allowed to start an instruction. This allowed 10 different I/O processes to be in execution simultaneously.

The concept of multiple processes sharing a pipelined functional unit appears again, in a slightly different form, in the Denelcor HEP architecture[1] [Smit81, AlGo90, HwBr84]. Here, an 8-stage execution pipeline is multiplexed across a set of *user* processes, instead of I/O processes. As in the CDC 6600, on each clock cycle an instruction from a different instruction stream is permitted to begin execution, and no two instructions from the same process can be in the pipeline concurrently. However, unlike the CDC6600, the instruction can be taken from the head of the queue of *any* available process. The underlying philosophy for the use of this approach arises from the desire to build a multi-processor/mult-processing system that will tolerate relatively long memory access times.

In the HEP, the delay to memory is relatively large and is not incorporated as part of the execution pipeline, so more than 8 processes are needed to keep the pipelined functional units full. In order to support a sufficient number of processes, hardware queues for up to 64 processes are provided per Process Execution Module (PEM). Thus when one process makes a memory request, it is pulled out of the pool of processes waiting on the execution pipeline and placed in a pool of processes waiting on the memory. Once the a memory request is satisfied, the process is placed back into the pool of processes waiting on the execution pipeline.

While the HEP multiplexed the execution pipeline in the context of a supercomputer, a similar approach was suggested by Kaminsky and Davidson [KaDa79] in the context of a single-chip processor. They argued that the main cost of supporting multiple processes is the need to duplicate the registers that store the state of a process. Due to their regular and dense structure, implementation of registers in a single-chip processor is a more efficient utilization of chip area than implementation of more complex control functions; therefore, the use of a shared pipeline in a single-chip processor provides a highly efficient use of chip real estate. However, they did not address the question of the ideal number of processes to support.

In this paper, we present an evaluation of the effectiveness of sharing pipelined functional units between two processes. The motivation behind selecting only two processes to share processor resources was the measured issue rate of the CRAY-1 - with an issue rate slightly less than 50% [PlSo88], on average an instruction enters the CRAY-1 functional units on every other clock. Therefore, an instruction from only one other stream can be expected to slip in and execute without affecting the original instruction stream. This second instruction stream is, in effect, **slot-stealing** from the first instruction stream. Unlike the other approaches described, the stages of the functional units are not necessarily shared every clock cycle in a round-robin fashion. Instead, full hazard checking is performed on both potential streams, and the original stream may be permitted to run as normal (depending on strategies to be outlined in detail later in this paper.)

3. Basic Machine Model

For our studies we used a modified CRAY-1 simulator developed at the University of Wisconsin [PaSm83]. The benchmark programs were the original 14 Lawrence Livermore Loops [McMa84]. Instruction traces were generated for each of the benchmark programs and then used to drive the simulations. No modifications to the code were performed; the code used is that produced by the Cray Fortran Compiler.

The simulator uses a processor model that is a slight variant of the CRAY-1S processor [CRAY82,Russ78].

---

[1] The HEP had several other interesting aspects, such as a system whereby a set of processes could share of the general-purpose registers. It also support a processor to memory interconnection scheme that used a *hot potato* routing strategy. With such a strategy, memory conflicts were, in effect, taken care of in the interconnection network. Finally, the shared data memory had a full/empty bit associated with each word. Between this bit and equivalent bits associated with the general-purpose registers, synchronization between processes was accomplished.

The hardware functional units in the simulator have the same performance characteristics as the CRAY-1 functional units; the time taken for a scalar add is 2 clock cycles, floating point multiply take 7 clock cycles, etc. The register files are also configured as in the CRAY-1. The only notable difference is that in the simulator, all instructions can issue in 1 clock cycle if issue conditions are favorable (unlike the CRAY-1S architecture, which requires an extra cycle for 32-bit instructions).

To enlarge the range of applicability of our simulation results, we varied the memory access time and branch execution time since these two parameters can, in some cases, have a significant impact on performance. The memory access times were divided into two categories; an 11 cycle memory access time (slow memory) and a 5 cycle memory access time (fast memory). The slow memory time was derived from the fact that in the CRAY-1, a memory access requires 11 clock cycles from the time the load instruction is issued until the time the destination register is available for use.

A memory access will not require 11 cycles if some form of fast intermediate storage, i.e., some form of cache is provided. The CRAY-1S has no cache; however, it has 8 64-element vector registers that can be used as a software-controlled cache in some cases. For instance, if a piece of scalar code accesses arrays in a regular fashion (a linear recurrence, for example), elements of an array can be vector loaded from the memory into one of the vector registers. These elements can then be moved one by one into the scalar registers as needed. In such a case, the effective "memory access" time is simply the time taken to move an element from a vector register to a scalar register. Since the time perform this move is 5 clock cycles, we use 5 cycles as the fast memory access time.

The branch execution time was also varied, since control dependencies are a significant source of instruction blockage (especially in some of the more sophisticated issue methods). The simulator model does not incorporate any type of guessing or branch prediction to get an early start on the execution of a likely branch target path. Execution of the branch target is not started until the branch outcome is known. As in the CRAY-1, each branch that is encountered blocks at the issue stage for a period of 4 clock cycles, even if the contents of the A0 (the register upon which the branch decision is made) are available. Such a branch requires 5 clock cycles to execute, and is referred to as a *slow* branch.

The block time associated with a slow branch is due to 2 factors. The first is that a branch is a 2-parcel (32-bit) instruction and requires an extra clock to get the 2nd parcel from the instruction buffer. The other delay associated with a branch is the time it takes to fetch a new target instruction stream from the instruction buffers. Since these delays are partly artifacts of the CRAY-1S implementation and could possibly be eliminated, simulations were performed in which a branch instruction took only two clock cycles to execute (it would still block on the availability of the A0 register). This type of branch is called a *fast* branch.

For each of the interleaving strategies described in the following section, four different machine configurations were simulated (corresponding to the fast memory-slow memory and fast branch-slow branch combinations). Since the memory request rate of the Livermore Loops is very low, and the two processes sharing the pipeline are by definition independent, we assume no memory conflicts occur over data. Furthermore, since the Lawrence Livermore Loops are all relatively tight loops, we assume that they fit in the instruction buffers (one can think of these are as an instruction cache), meaning there are no memory conflicts over instructions either.

It was also assumed that the act of choosing between instruction streams for the next instruction has no affect on the machine cycle time. We felt this was a reasonable assumption to make since all that is required of the Stream Selection Logic is to combine 3 bits (the Stream A instruction valid bit, Stream B instruction valid bit, and Stream Selection bit).

4. Interleaving Strategies

Having two instruction streams (A and B) vying for use of the functional units requires the specification of a policy that selects an instruction from one or the other stream. In this study 3 different policies were considered:

1. Every-Cycle
2. Blocked
3. Prioritized

The **Every-Cycle** selection policy is the most conservative. In this scheme, during each clock cycle, the Stream Selection bit switches state, pointing to the stream that was not selected during the previous cycle. This toggling occurs on every clock, and is independent of the issueability of the instruction at the head of the stream. For example, if the instruction at the head of stream A is ready to issue but the next instruction in stream B is blocked due to some hazard, and the toggle is pointing at B, then no instruction will be issued on this clock cycle.

In the **Blocked** selection policy, the Stream Selection bit only changes state when a stream has no instruction available to issue. In other words, instruction stream A executes until it is blocked for some reason, at which time instruction stream B begins execution. This policy turns out to be very effective at slot-stealing, but the instruction streams do have the potential to interfere with each other. For example, the instructions at the head of both streams may be ready to execute, but only the stream selected by the Stream Selection bit is allowed to proceed. The other instruction stream is forced to wait.

This potential for interference led to the development of the **Prioritized** policy, which was modeled because there may be situations in which one process is more important than another, and what is desired is to execute a second process *only* in the dead cycles of the first process. In this policy, the Stream Selection bit will point to the higher priority instruction stream, A, until A blocks. At this point stream B will be allowed to resume execution. However, as soon as the conditions that caused A to block have been resolved, stream A is allowed to resume execution. With the Prioritized policy, the low-priority stream B should have a minimal impact on the execution time of stream A.

5. Configuration of the Result Bus

In a conventional system that supports a single process, there is only one result bus. In machines with fixed length functional units, like the HEP, no special result bus arbitration circuitry is required since no two instructions can complete at the same time. However, in machines like the CRAY that feature several multiple-length functional units, it is possible for more than one functional unit to finish simultaneously and require the use of the result bus. In order to prevent this from happening, the processor must *schedule* the use of the result bus, keeping track of when in the future each functional unit will finish, and reserving the result bus for that functional unit during that corresponding clock cycle. An instruction is not allowed to issue if the result bus is scheduled to be busy during the clock cycle that the requested functional unit would complete. This guarantees that the result bus will be available when the functional unit produces the result of a computation.

Since we are supporting two instruction streams specifically in order to increase the utilization of the functional units, it is apparent that an increase in the number of result bus conflicts is inevitable. While the *cheapest* approach is to have both processes continue to share a single result bus, the result bus may become a bottleneck in the system; one process may be unable to issue during what would normally be dead slots because the result bus has been scheduled by the other process.

This problem can be circumvented by supporting two separate result buses. By providing each process its own private bus to its register file, all internal conflicts between the two processes are removed. The only way one process can interfere with the other is through the interleaving policy or through the memory system.

A logical extension of this idea is to provide two *shared* buses, instead of two private ones. Such a configuration can actually allow a single process to execute faster than it would with only a single bus, since result bus conflicts internal to the process will be reduced. The justification for this approach is that as long as the hardware exists to support two buses, each process should be permitted to use both buses to the fullest extent. One possible drawback of this two-shared-bus option is that it requires that the register file be capable of supporting two simultaneous writes. It should be noted, however, that it is never possible for both buses to attempt to write into the same register at the same time, because of the data dependency checking that is performed at the issue stage.

6. Simulation Results

To evaluate the relative effectiveness of the interleaving strategies described above, a large number of simulations were performed. In these simulations, for each interleaving policy under study, the configuration of the result bus, the speed of external memory, and the branch times were varied. Figures 1 through 3 present the results of these simulation runs.

In each of these simulations, the first 14 Livermore Loops were concatenated to form process P_A. A random reordering of the 14 Loops was concatenated and used as the second stream (P_B). Dozens of random reorderings of the loops were used as the pool of tasks to run against the initial ordering, in order to ascertain if the ordering of the loops caused a significant difference in the results. Since no such ordering effects were observed, the results presented are an average of the random ordering results.

In each of the following figures, on the horizontal axis the various memory speed/branch speed combinations are displayed, while the vertical axis represents the percentage of time taken to run both instruction streams P_A and P_B to completion using one of the slot-stealing strategies vs. running both instructions streams serially (P_A *followed* by P_B). Depending on the interleaving policy, the utilization of available slots varies substantially. It should also be stressed that these are percentages, and not absolute times - the total amount of time necessary to execute the benchmark programs on the fast memory/fast branch machine is going to be less than on the slow memory/slow branch machine, but the relative performance of the various strategies may be constant.

As can be seen the three figures, regardless of the configuration of the machine in general the Blocked policy performs the best, while the Every-Cycle policy performs the worst. The effectiveness of a given strategy can be calculated by observing how close to the 50% line the strategy is (based again on the fact that the issue rate for the Lawrence Livermore Loops is approximately 45%). For example, in Figure 1 for the Slow Memory/Slow Branch configuration using the Blocked strategy, the total time required to execute the two processes was 11% more than needed to execute a single process, and 39% less than the time necessary to execute the two processes sequentially. Perfect slot-stealing is not achieved largely due to the increased number of result bus conflicts generated by

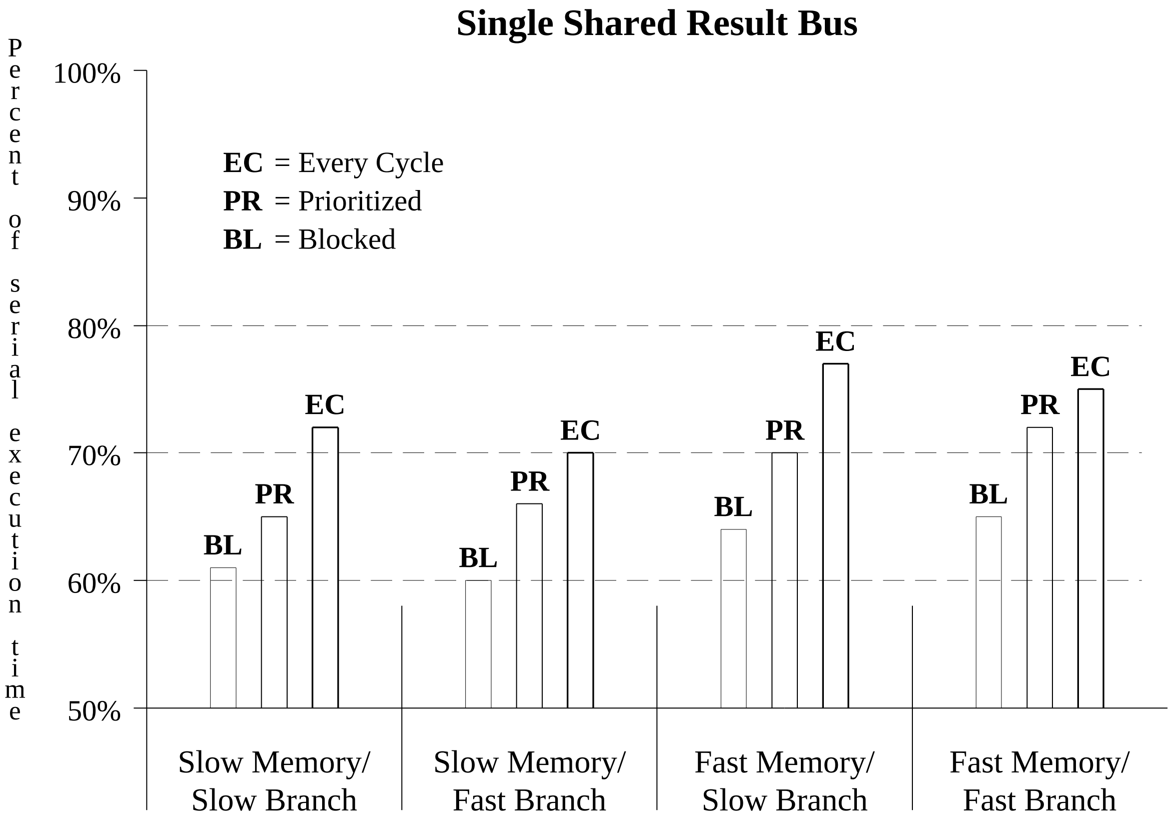

Figure 1.

forcing two streams to share the single result bus.

The reason the Prioritized scheme always finishes between the Blocked and Every-Cycle schemes is due to its inherent structure. While the higher priority process (P_A) is executing, the hardware is being fully utilized, with the second process (P_B) executing in the dead slots of P_A. However, process P_A is **guaranteed** to complete first, since it has priority, and process P_B is left to complete in an essentially sequential manner (since in our simulations there was no other process to execute in its dead slots).

A much better metric for the prioritized scheme is the impact of hardware sharing on the execution time of the primary process. We found that with a single shared bus structure, the high priority process takes 2% to 3% longer to execute than if it were running alone, due to the result bus conflicts generated by the low-priority process. However, for the other bus structures studied, the high-priority process finished in the same amount of time or in 1% less time than in the original single stream case. This indicates that the Prioritized scheme is extremely effective if one has high and low-priority processes.

At the far left of Figure 1 is the machine variant that is the same as the CRAY-1S, a relatively long memory access time and branch delay (in clock cycles). For the Every-Cycle policy, the two instruction streams are executed in 72% of the time it would take to execute both in sequence. A Prioritized policy does somewhat better at 65%, while the blocked strategy performs the best at 61%.

In relative terms, each interleaving policy appears to demonstrate poorer performance on the fast memory/fast branch machine than on the slow memory/slow branch configuration. This is due primarily to the fact that as the memory and/or branch speeds increase, fewer dead slots exist in an instruction stream for the alternate process to steal. However, the difference is not dramatic, indicating that most dead slots are created by other than memory hazards.

It is also important to reiterate that one must keep in mind the effect of the use of a percentage to measure the effectiveness of an interleaving policy when looking at the figures. For example, the prioritized policy for the fast memory/fast branch configuration takes fewer cycles (less absolute time) than the prioritized policy for the slow

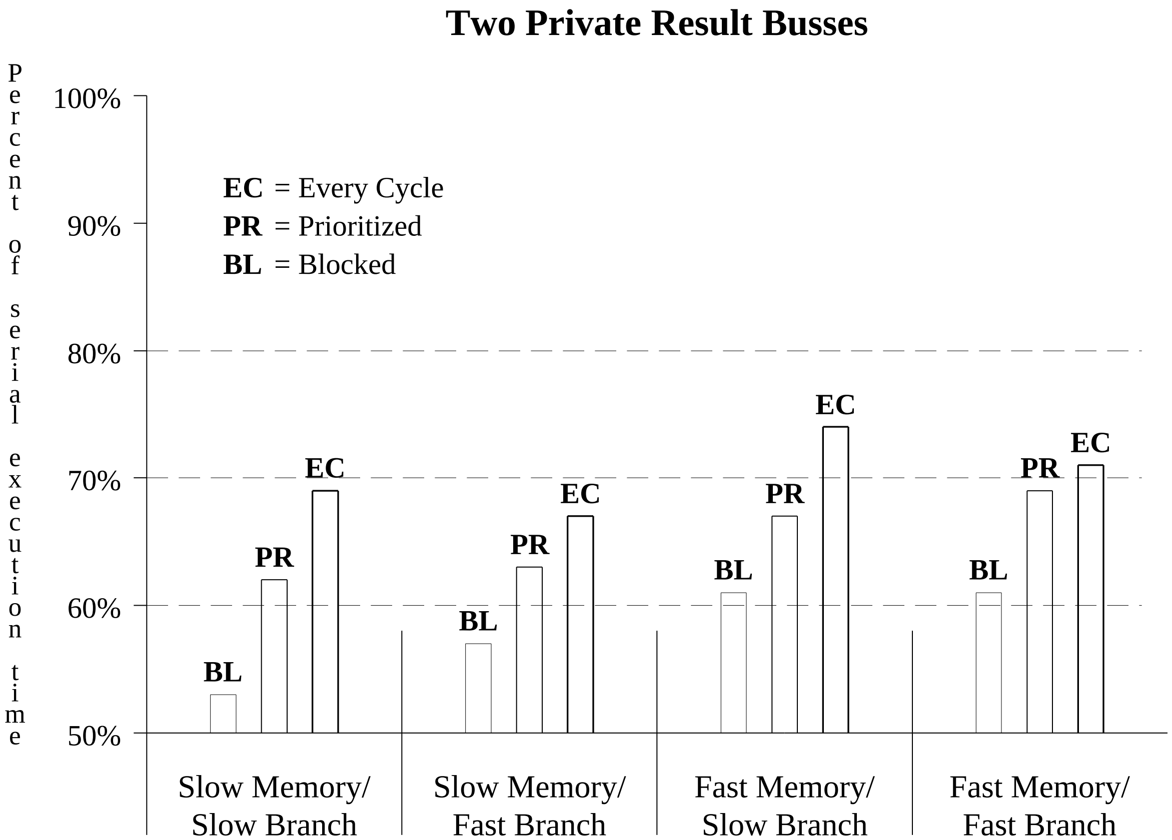

Figure 2.

memory/slow branch configuration. However, since a process does not have to block as long in a fast memory/fast branch scheme, the second process has less opportunity to steal slots from the first process; thus the larger *relative* improvement.

Figure 2 shows the results for a two result bus system in which each processor has its own private result bus. Comparing this figure to Figure 1, across the board performance improvements are evident. For all interleaving policies and machine configurations the two private bus system provides performance improvements over the single shared bus. Since the two private result bus structure eliminates any blocking due to result conflicts between the two processes, we see that for the slow memory/slow branch configuration using the Blocked strategy it only takes **3%** more time to completely execute two processes than to execute a single process. This is an increase in throughput of 1.89, very close to the ideal of 2. Even for a fast memory/fast branch system, the blocked policy permits two processes to finish within 11% of a single process. In other words, the throughput has improved by a factor of 1.64.

It is also important to note that for the Prioritized policy the higher priority process finishes at **exactly** the same time as it would have had the second process not been running. This is significant for environments with multiple-priority processes: For example, a workstation using this technique might make the user process process P_A, and could then execute the operating system "for free" by designating it as the secondary process P_B.

Although Figure 3 shows that the relative performance of the various strategies declines with a two shared bus result structure, it should be noted again that with two shared result buses the number of conflicts within an instruction stream decreases, providing fewer dead slots for the alternate instruction stream to steal. The individual processes are also actually completing in less time given this bus structure than they did with the single bus structure. Therefore, if the processor is going to be designed to support two buses, even though the relative improvement figures imply that the private bus structure should be chosen, in terms of absolute execution times, the shared bus structure is more beneficial.

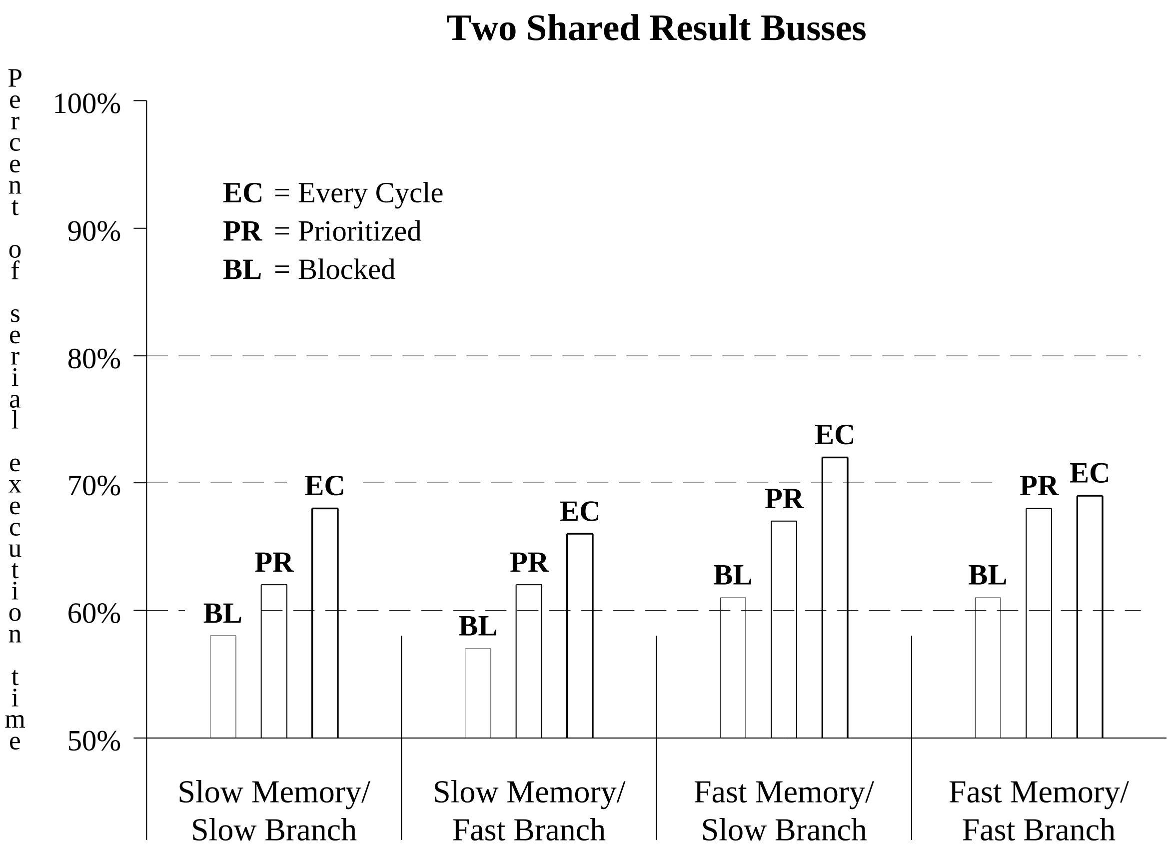

Figure 3.

7. Summary and Conclusions

In this paper, we examined the performance improvement that can be achieved by permitting two independent instruction streams to share a single set of functional units. The motivation for this study arose from the relatively low issue rates found in deeply pipelined processors. In such systems, roughly half the time an instruction is blocked at the issue stage. This implies that a second instruction stream can slip in and issue instructions during the cycles that the original instruction stream can not.

A number of policies that could be used in issuing and executing two instructions streams were investigated. The policies were evaluated with different result bus structures, memory access delays and branch execution times. For a single result bus structure, two processes could interleave their execution so as to executed in 61% to 78% of the time it would take to execute the processes sequentially. This corresponds to an improvement in system throughput ranging from 1.64 to 1.28. With two result buses, each allocated to a particular process, the processes could be interleaved to execute in 53% to 74% of the time would take to execute sequentially. This effectively improves the throughput of the system by a factor of from 1.89 to 1.35. And finally, with two shared results buses, the two processes could execute in 59% to 72% of the time they would execute sequentially (1.69 to 1.39 improvement factors). The variation in the execution times is due to different interleaving policies, memory accesses delays and branch delays. In the best case, at the cost of duplicating the register files and, possibly the decode and issue logic, it is possible to execute 2 processes in almost the same amount of time it takes execute one process.

8. Acknowledgements

We would like to extend a special thanks to Rob Fanfelle for running initial simulations of this idea on the PIPE simulator, and demonstrating the potential of the idea.

References

[AlGo90] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing,* Benjamin/Cummings Publishing Co. Inc., pp. 425-429, 1990.

[CRAY82] *CRAY-1 Computers, Hardware Reference Manual,* Chippewa Falls, WI, Cray Research Inc., 1982.

[HwBr84] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill Book Co., pp. 669-684, 1984.

[JoWa89] N. P. Jouppi and D. W. Wall ''Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines,'' *Third International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 272-282 April 1989.

[KaDa79] W. J. Kaminsky and E. S. Davidson, ''Developing a Multiple-Instruction-Stream Single-Chip Processor,'' *Computer,* pp. 66-76, December 1979.

[McMa84] F. H. McMahon, ''LLNL FORTRAN KERNELS: MFLOPS,'' Lawrence Livermore Laboratories, Livermore, CA, March 1984.

[PaSm83] N. Pang and J. E. Smith, *CRAY-1 Simulation Tools,* Tech. Report ECE-83-11, University of Wisconsin-Madison, Dec. 1983.

[PlSo88] ''The Performance Potential of Multiple Functional Unit Processors,'' *Proceedings of the 15th Annual Symposium on Computer Architecture,* pp. 37-44, June 1985.

[Russ78] R. M. Russel, ''The CRAY-1 Computer System,'' *Communications of the ACM,* vol. 21, no. 1, pp. 63-72, January 1978.

[Smit81] B. J. Smith, ''Architecture and Applications of the HEP Multiprocessor Computer System,'' *SPIE Real Time Signal Processing IV,* Vol. 298, pp. 241-248, Aug. 1981.

[Thor70] J. E. Thornton, *Design of a Computer - The Control Data 6600,* Scott, Foresman and Co,. 1970.