THE PERFORMANCE POTENTIAL OF MULTIPLE FUNCTIONAL UNIT PROCESSORS

by

A. R. Pleszkun and G. S. Sohi

Computer Sciences Technical Report #752

February 1988

The Performance Potential of Multiple Functional Unit Processors

A. R. Pleszkun and G. S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

Abstract

In this paper, we look at the interaction of pipelining and multiple functional units in single processor machines. As a basis for our studies we use a CRAY-like processor model and the issue rate (instructions per clock cycle) as the performance measure. We initially find that in non-vector machines, pipelining multiple function units does not provide significant performance improvements. Dataflow limits are then derived for our benchmark programs to determine the performance potential of each benchmark. In addition to a traditional dataflow limit computation, several other limits are computed which apply more realistic constraints on a computation. Based on these more realistic limits, we determine it is worthwhile to investigate the performance improvements that can be achieved from issuing multiple instructions each clock cycle. Several approaches are proposed and evaluated for issuing multiple instructions each clock cycle.

1. Introduction

To improve the performance of a single processor, designers look to approaches that permit parallelism, or overlap, in instruction execution. Historically, pipelining has been one of the most popular of these approaches. Another technique that can be used independently or to complement pipelining, is the use of multiple functional units. In either case, the application of such approaches can lead to substantial improvement in a machine's maximum performance because the number of resources that are simultaneously available to a running program is increased. As more resources are offered to a program, the program should execute faster and a higher performance should be achieved.

A well-known performance measure used to evaluate high performance machines is the instruction issue rate, i.e., the number of instructions that are issued per clock cycle. In conventional pipelined single processor machines, the issue rate is limited to 1 instruction per cycle since each instruction must pass through a single instruction decode and issue stage [1]. Experience and our results, indicate that the issue rate achieved when executing benchmark programs is always less than 1. Traditional dataflow analysis of the same programs indicates, however, that issue rates greater than 1 can be achieved. The difference between the actual issue rates and those based on dataflow analysis arise from overly optimistic assumptions made in the dataflow analysis. Dataflow analysis assumes an infinite number of resources; for example, all instructions that can issue in any clock cycle are indeed issued during that clock cycle. Although, dataflow analysis is very optimistic, the significant difference between it and actual issue rates indicates that an investigation of alternate machine organizations is worthwhile.

Historically, performance improvements have been sought by concentrating on methods that either increase the number of functional units (or their availability through pipelining) or increase the amount of buffer storage (registers). To approach the dataflow limit however, one must also investigate issuing multiple instruction each clock cycle. We therefore examine the performance that can be achieved in multiple function unit processors that permit multiple instructions to be issued each clock cycle. The studies reported here are an exploration of a design space in which we investigate the performance potential of machines that have reasonable, realistic limits on the performance of functional units and memory, but use different methods to issue more than 1 instruction per clock cycle. Some of the multiple instruction issue methods are clearly too difficult to implement, however, they are presented for the sake of completeness.

In this paper, we first describe our basic machine model. Then, we present performance improvements gained through pipelining for single issue unit. Next, we calculate dataflow limits for our benchmark programs to determine the potential for improving performance. Finally, we discuss several schemes that can issue multiple instructions per clock cycle in an attempt to improve performance.

2. The Basic Machine Model

To form a basis for comparing different instruction issue methods, we have selected a base architecture and organization to drive our studies. Each variant of an instruction issue method uses the same set of hardware functional units and memory. The instruction set of this base architecture is very similar to the CRAY-1S instruction set [2], consisting of 1-parcel (16 bits) and 2-parcel (32 bits) instructions. Unlike the CRAY-1S architecture, our base architecture can issue all instructions in 1 clock cycle if issue conditions are favorable. The hardware functional units in the base architecture have the same performance characteristics as the CRAY-1 functional units; the time taken for a scalar add is 2 clock cycles, etc. The register files are also the same as in the CRAY-1. All operations are measured in clock units and the clock speed is the same irrespective of the hardware organization.

In addition to varying the issue method, we also vary the memory access time and branch execution time since these 2 parameters can have a dramatic impact on performance in some cases. Using the CRAY-1 model, 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. Our experiments were performed for an 11 cycle memory access time (slow memory) and a 5 cycle memory access time (fast memory). A fast memory results 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 (for example a linear recurrence), 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, i.e., 5 clock cycles.

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. In our basic machine model, we have not incorporated 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. Since the base architecture uses a CRAY-like model, 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 and, therefore, requires 5 clock cycles to execute. We call the 5 cycle branch 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 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. These delays are partly artifacts of the CRAY-1S implementation and could possibly be eliminated. To evaluate the impact of eliminating these delays, simulations were performed in which a branch instruction took only 2 clock cycle to execute (it would still block on the availability of the A0 register). This type of branch is termed a *fast* branch.

For our studies we used a modified CRAY-1 simulator developed at the University of Wisconsin [3]. The benchmark programs were the original 14 Lawrence Livermore Loops [4]. Instruction traces were generated for each of the benchmark programs and then used to drive the simulations. The programs were divided into the 5 scalar loops, loops 5, 6, 11, 13 and 14 and the 9 vectorizable loops, loops 1, 2, 3, 4, 7, 8, 9, 10 and 12. Separate results are presented for the scalar loops and the vector loops. We make this separation because we expect the vectorizable loops to exhibit a reasonably high degree of parallelism while we expect the scalar loops to exhibit a comparatively low degree of parallelism. Unless noted otherwise, the results for each class of loops are given as the harmonic mean[5] of the individual loop issue rates (number of instructions issued per clock cycle). Since the memory access time and the branch execution time are orthogonal parameters, for each issue method, four machine variations were studied: (1) M11BR5, (2) M11BR2, (3) M5BR5, and (4) M5BR2. These variations correspond to the fast memory-slow memory and fast branch-slow branch combinations. Throughout our studies, we did not make any modifications to the code. Of course, with software optimizations, the results of each individual study would be different.

3. A Single Issue Unit

Let us first investigate how different degrees of pipelining and instruction overlap effect the issue rate of a machine with a single instruction issue unit. Presumably, the greater the degree of pipelining, the greater the issue rate that can be achieved. Given a machine with multiple functional units, pipelining techniques may be applied to the functional units, the memory system, or both. In this section, we discuss 4 possible machine organizations, all with a single issue unit, and evaluate the performance improvement achieved as progressively more pipelining is used.

3.1. A Serial Machine

As a starting point and a lower-bound on the achievable issue rate, consider a simple, serial machine organization. In this *Simple Machine*, there are two distinct phases in processing an instruction: (i) an instruction fetch, decode and issue phase in which an instruction is fetched from memory, decoded and is dispatched to a functional unit for execution and (ii) an instruction execution phase in which the instruction is executed in a functional unit. At any time, at most one instruction can be in each phase of execution. Thus, the Simple Machine has a two stage pipeline. An instruction enters the decode and issue stage and proceeds to the execution stage when the previous instruction has left the execution stage. For example, the execute stage is busy for one cycle if the instruction is a register to register transfer instruction and is blocked for 11 cycles if the instruction is a memory load instruction (which takes 11 cycles to complete execution). Since all previous instructions are guaranteed to have finished when an instruction enters the execution stage and since no instruction can enter the execution stage until the previous instruction has left, no checks are needed for data dependencies. With only a single instruction in the execution stage, there is no overlap of instructions in the execution stage.

Since the Simple Machine makes no attempt to overlap the usage of its independent functional units, there is clearly some potential for improving the issue rate of the machine. The next step is, therefore, to organize the hardware so overlap of instructions in the execution stage is possible.

3.2. Overlapping the Execution of Instructions

To utilize the multiple functional units found in the execution unit, several design choices are possible. First, the functional units, including memory, do not have to be fully segmented, i.e., pipelined.

Therefore, the first and simplest choice permits all functional units, including memory, to simultaneously be active, however each unit is restricted to executing only one instruction at a time. Instructions that use distinct functional units can overlap in the execution stage but instructions that use the same functional unit cannot. We call this design the *SerialMemory Machine* since the memory (as well as the functional units) can handle at most a single request at any instant.

Since the memory is likely to be a heavily-used "functional unit" with a comparatively long latency (11 cycles), the next step in this progression of designs is to interleave the memory. An interleaved memory can service several requests simultaneously in a pipelined fashion. The functional units in the execution stage remain unchanged, i.e., each functional unit can service only one request at a time. This machine is called the *NonSegmented Machine* since the functional units are still nonsegmented. Such an organization of functional units is found in the CDC 6600 [6].

Finally, all the functional units in the execution unit can be segmented, simultaneously handling several independent requests at a maximum rate of one request per clock cycle. Since this organization corresponds to the organization found in the CRAY machines, we shall call this organization the *CRAY-like Machine*. In a such a machine, instructions that use distinct functional units as well as instructions that use the same functional unit can overlap in the execution stage.

The results of executing the benchmark programs on the 4 machines described above are shown in Table 1. As seen in the top-half of Table 1, scalar code achieves the greatest benefit if we allow the overlap of instructions that use distinct functional units. If the memory is slow, interleaving the memory is

Table 1. Instruction Issue Rates for Different Basic Machine Organizations.

Code	Machine	M11BR5	M11BR2	M5BR5	M5BR2
Scalar	Simple	0.24	0.25	0.32	0.33
	SerialMemory	0.35	0.36	0.48	0.50
	NonSegmented	0.43	0.45	0.50	0.53
	CRAY-like	0.44	0.47	0.51	0.55
Vector-izable	Simple	0.21	0.21	0.29	0.30
	SerialMemory	0.29	0.30	0.42	0.45
	NonSegmented	0.42	0.45	0.49	0.53
	CRAY-like	0.45	0.49	0.54	0.59

helpful. Pipelining the functional units, however, does not have a significant impact on performance. This is seen by comparing the results for the NonSegmented and CRAY-like machines. In all the machine organizations, a relatively large performance gain is made by interleaving the memory alone than by pipelining the functional units. This result is not counter-intuitive. For scalar code, it is rare that several instructions need to use the same functional unit in a time window that is determined by the latency of the functional unit. An exception is the memory since a large number of instructions reference the memory. If the latency of the memory is large, the performance can be improved significantly by interleaving the memory alone (about 23-25% over a serial memory for scalar code). If the latency of the memory is smaller, the performance improvement is not so significant (about 4-6% for scalar code). Vectorizable code benefits not only from overlap amongst instructions that use distinct functional units, it also benefits from the interleaving of memory and from the overlap of instructions that use the same functional unit.

For *vector operations* executed in the *vector unit*, clearly the functional units clearly should be highly pipelined to allow for maximum overlap in the processing of successive elements of a vector. If the same functional units are to be used both by scalar and vector operations (as in the CRAY machines), pipelining the functional units also makes sense. If, however, there are independent functional units for the scalar unit and for the vector unit, the computer designer may choose to pipeline the vector functional units only, thereby increasing the overlap of operations whereas the scalar functional units may not be pipelined at all, thereby reducing the skew factors in the clock [7].

3.3. Other Issue Schemes with a Single Issue Unit

Given the functional units of a *CRAY-like* machine, the instruction issue rate can be further improved by making the issue unit more elaborate. The instruction issue schemes of Section 3.2 block instruction issue when dependencies or hazards are encountered. The 2 types hazards that cause blockage are *read after write (RAW)* and *write after write (WAW)* hazards [8]. (Note that *write after read (WAR)* hazards are not important in a single processor situation.) Several issue schemes that permit instruction issue in spite of these hazards exist for single issue units. For example, the instruction issue scheme used in the CDC 6600 handles RAW hazards but blocks instruction issue when a WAW hazard is encountered [6]. The instruction issuing scheme used in the IBM 360/91 floating point unit issues instructions in spite of RAW and WAW hazards [9]. Another instruction issuing scheme presented in[10] not only issues instructions

despite existing hazards, it also implements precise interrupts. For such schemes, the issue stage is not concerned with dependencies - they are enforced by other units in the machine. Such issue schemes are, therefore, also referred to as *dependency resolution schemes*. Since such issue schemes have been studied elsewhere for single issue units [10-12], we shall not discuss them any further.

By issuing instructions in spite of dependencies it is possible to achieve the optimum instruction issue rate possible with a single issue unit. For example, using the dependency resolution scheme described in [10], the issue rate of an M11BR5 machine with a single issue unit can be improved to about 0.72 instructions per cycle for scalar code and 0.81 instructions for vectorizable code [13]. To achieve further improvements, one must issue more than one instruction per clock cycle, i.e., more than one issue unit must be used. In the following sections, we look at improving performance by modifying the issue stage to permit multiple instruction issue. In order to minimize the number of cases (and to get an upper-bound on performance), we restrict further experiments to machines with fully segmented functional units and an interleaved memory system, i.e., a machine with CRAY-like functional units.

4. Limits on Performance

With a single issue unit one can, at best, achieve an issue rate of 1 instruction per clock cycle. As was seen in the previous section, with simple issue strategies this maximum performance is not easily reached. When multiple issue units are used, the maximum issue rate is related to the number of instructions than can be simultaneously issued. With N issue units, the maximum issue rate is N instructions per clock cycle. However, as the number of issue units is increased, at some point, this number may exceed the width of the dataflow graph associated with a program. In such cases, the resulting issue rate may look modest compared to the maximum allowed by the hardware. In order to determine the effectiveness of multiple instruction issue, we found it necessary to compute an upper-bound on the issue rate. This upper-bound is related to the *dataflow limit* of the program and is generally unachievable for most machine organizations. The dataflow limit is calculated in two steps; first, a *pseudo-dataflow limit* is calculated and then a *resource limit* calculated. The actual dataflow limit for each program is the greater of these 2 limits.

The pseudo-dataflow limit is found by taking the number of instructions executed by a program and dividing by the amount of time taken to traverse a dataflow graph of the program. This computation

assumes that the program resides in memory in the form of a dataflow graph and that an instruction can execute as soon as its operands are available. There are no resource limits (such as limits on the bus structure, number of functional units, etc.), however, different portions of the dynamic program graph, i.e., different loop iterations, cannot start until the appropriate branch conditions have been resolved. In such an idealized machine, the best-case time taken to execute a program, i.e., the pseudo-dataflow limit, is the length of the critical path in the program. Since this limit is different for different encodings of an algorithm, this limit is a property of the encoding of the benchmark program rather than a property of the algorithm itself. Note that, the pseudo-dataflow limit is also dependent on compiler optimizations. For example, loop unrolling will in some cases shorten the critical path because some of the program's branches are removed.

The pseudo-dataflow limit can be an overly optimistic upper-bound because it assumes an unlimited number of resources. If there are a dozen independent floating point multiplies in the basic block of a loop, under the assumptions of the pseudo-dataflow limit, these can all proceed concurrently. For the basic machine we are studying, resources are limited. For example, there is only 1 floating point multiply unit and this unit can only accept 1 new floating point operations every clock cycle. From such realistic hardware limitations we calculated a resource limit for a program. When calculating the resource limit, we assumed that the hardware functional units were limited to those of our base machine. Therefore, for a given program, the the best-case execution time is bounded by the maximum number of instructions that use the same functional unit. For example, the best-case execution time for a basic block with a dozen independent floating point multiply operations is equal to 12 clock cycles plus the latency of the multiply unit on our hardware as opposed to simply the latency of the multiply unit in a pure dataflow machine with unlimited resources.

The performance limits for various machine organizations are given in Table 2. The top half of Table 2 (those entries with a "Pure" prefix) shows the issue rates for the pseudo-dataflow and the resource limits for vectorizable and scalar loops for each machine type. The actual limit is found by taking the smaller of the 2 limits for each program and calculating the harmonic mean. Because the actual limit for individual loops is sometimes the pseudo-dataflow limit and other times the resource limit, the actual limit for a set of loops is not simply the smallest of either the overall pseudo-dataflow or resource limits. The

Table 2. The Pseudo-Dataflow and Resource Limits for Vector and Scalar Loops.

Code	Machine	Pseudo-Dataflow Limit	Resource Limit	Actual Limit
Scalar	Pure M11BR5	1.34	4.66	1.29
	Pure M11BR2	1.37	4.66	1.29
	Pure M5BR5	1.34	4.66	1.29
	Pure M5BR2	1.37	4.66	1.29
Vector-izable	Pure M11BR5	3.35	3.43	2.78
	Pure M11BR2	4.40	3.43	3.15
	Pure M5BR5	3.35	3.43	2.78
	Pure M5BR2	4.40	3.43	3.15
Scalar	Serial M11BR5	0.79	4.66	0.79
	Serial M11BR2	0.79	4.66	0.79
	Serial M5BR5	0.85	4.66	0.85
	Serial M5BR2	0.85	4.66	0.85
Vector-izable	Serial M11BR5	0.93	3.43	0.93
	Serial M11BR2	0.96	3.43	0.96
	Serial M5BR5	1.05	3.43	1.05
	Serial M5BR2	1.09	3.43	1.09

results in Table 2 indicate that with realistic limitations on the amount of available hardware and inherent program dependencies, very larger issue rates cannot be achieved. However, it is possible to achieve an issue rate of greater than 1 instruction per cycle and, therefore, the investigation of multiple issue units may be worthwhile.

The lower half of Table 2 (those entries with a "Serial" prefix) shows results that establish even tighter bounds on the issue rate for a certain class of machines. The actual limit for the "Pure" results represents the situation where there are an unlimited number of issue units and a limited number of functional unit resources. Implicit in the calculation of this limit is the assumption that an unlimited amount of buffer storage is available to store temporary or intermediate results. This assumption is significant when a WAW hazard is encountered. Consider 3 instructions, I_i, I_j, and I_k. Assume that both instructions I_i and I_j write to register X and that instruction I_k uses register X as a source operand. Furthermore, assume that instruction I_j finishes execution before instruction I_i. The "Pure" limits assume that instruction I_k can start execution as soon as the result of instruction I_j is available. In a machine that has no mechanism to buffer

the result of instruction I_j, issue of instruction I_j must be blocked, forcing it to finish, at best, at the same time as I_i. Forcing I_j to be delayed also has the unfortunate effect of delaying the issue I_k. The "Serial" results in Table 2 calculate the issue limit if WAW hazards are handled by forcing instructions that write into the same register to finish execution in order. This "serial" limit also gives us a handle on the best-case issue rate that could be achieved for a CRAY-like machine assuming that the memory is infinitely fast.

Forcing instructions that write to the same register to finish in order has a dramatic effect on the actual performance limit of a program. As is seen in Table 2, except for 2 machines organizations executing vectorizable loops (M5BR5 and M5BR2), it is not possible to achieve an issue rate greater than 1 instruction per cycle. In the light of these limits, the issue rates found in Table 1 are not as low as they may seem. Nonetheless, there is still room for improvement, specially for the "Pure" case.

5. The Use of Multiple Issue Units

We have seen that starting from a simple machine (Table 1) the issue rate was substantially improved in going to a CRAY-like machine. The issue rate could be further improved if more sophisticated issuing schemes such as those mentioned in Section 3.3 were used. However, with a single issue unit we can achieve, at best, an issue rate of 1 instruction per cycle [1]. Since the dataflow limits indicate the potential for a higher issue rate, we shall attempt to improve the issue rate even further. All the methods that we consider attempt to improve the issue rate by trying to issue multiple instructions per cycle, i.e., by making use of multiple issue units. Some of the approaches discussed are clearly not feasible from the point of view of implementation, however, they are presented for the sake of completeness.

5.1. Multiple Issue Units with Sequential Instruction Issue

The simplest version of multiple instruction issue has several issue units, each of which can issue 1 instruction per cycle. The one restriction in this scheme, however, is that instructions must issue in program order. In this model architecture, the hardware fetches a block of instructions into an instruction buffer. This buffer is very similar to the line of an instruction cache. The number of instructions in the buffer is equal to the number of issue units. Once fetched, the instructions in the instruction buffer are examined in parallel by the issue units. If any instruction cannot issue, succeeding instructions cannot be issued even if their resources are available. The instruction buffer is refilled only after all of its instructions

have issued. (Of course, on a branch the buffer may be refilled before all the instructions have issued.) Clearly, the logic to maintain the dependencies and perform such simultaneous issue can be prohibitively large (expensive); this is of little concern since we are mainly concerned with the *potential* performance that can be achieved, irrespective of hardware costs.

Since several instructions could possibly issue in a single cycle, several results could also be produced in a single cycle. Since these results must be stored into the register file, the register file must have an ample number of ports. Therefore, another parameter in our machine organization is the maximum number of results that can be produced in any given cycle, i.e., the number of result busses that form the interconnection between the outputs of the functional units and the register file.

Several designs for the interconnection between the functional units and the register file are possible. First, we could have N result busses where N is the number of issue units. These busses could be organized in a crossbar fashion, i.e., the result of an instruction issued from an any issue unit to a functional unit could appear on an any result bus. Thus, an issue unit selects any available result bus to route a result to the register file. The only blockage due to a result bus conflicts occurs when all N busses have already been scheduled for use in the same clock cycle. We call this the full crossbar or *X-Bar* organization. Clearly the logic needed to implement a scheduling algorithm for such a bus is extremely complex; furthermore, the register file needs N write ports.

A simpler way of using N result busses restricts the result of an instruction issued by issue unit i to result bus i. This approach simplifies the result bus scheduling algorithm since each issue unit only needs to look at the reservations for a single bus. However, the register file still needs N write ports. We call this the *N-Bus* interconnection. Finally, we could choose to have fewer than N busses. In the limit, we consider the use of a single result bus, i.e., a *1-Bus* interconnection. In the 1-Bus interconnection, all results appear on a single result bus and the register file only needs a single write port. The simulation results for a machine with the N-bus and 1-Bus organization are found in Table 3. Since the results for the X-bar case are essentially the same as those for the N-bus case, we only present the results for the N-bus case.

Table 3 shows some interesting performance results for the scalar loops. First, having the capability of issuing up to 8 instructions per cycle is almost equivalent to having the capability of issuing 3 or 4

Table 3. Multiple Issue Units, Sequential Issue of Scalar Code.

Number of Issue Stations	Machine							
	M11BR5		M11BR2		M5BR5		M5BR2	
	N-Bus	1-Bus	N-Bus	1-Bus	N-Bus	1-Bus	N-Bus	1-Bus
1	0.44	0.44	0.47	0.47	0.51	0.51	0.55	0.55
2	0.45	0.45	0.49	0.49	0.54	0.53	0.58	0.58
3	0.46	0.46	0.50	0.50	0.55	0.55	0.60	0.60
4	0.46	0.46	0.50	0.50	0.55	0.55	0.60	0.60
5	0.47	0.46	0.50	0.50	0.56	0.55	0.61	0.60
6	0.47	0.46	0.50	0.50	0.56	0.55	0.61	0.60
7	0.47	0.47	0.51	0.51	0.56	0.56	0.61	0.61
8	0.47	0.47	0.51	0.51	0.56	0.56	0.61	0.61

instructions per cycle. This is found by reading down the columns of the table. In other words, most of the performance improvement one experiences from multiple issue occurs for a small number of issue stations. This fact simply highlights the number of dependencies that exist in the code. It is rare that 2 consecutive instructions are independent and can issue simultaneously without blocking. Furthermore, remember that for these simulations, instructions are still forced to issue in order. Comparing N-bus and 1-Bus columns, we see that restricting the size or use of result bus does not significantly impact performance. The issue rates for these machines simply are not high enough in any of the cases to cause a large amount of contention for the result bus.

A second interesting point is that the performance improvements made from multiple issue and the associated hardware complexity can be equally achieved by reducing the memory access time or the branch block time. Looking at columns M11BR5 and M11BR2, the issue rate for a machine with a slow branch and 5 issue units is identical to a machine with a single issue unit and a fast branch. A similar result is seen when comparing M5BR5 with M5BR2. For a given branch execution time, a faster memory access time also has a significant performance impact (consider the last row of the M11BR5 column and the first row of the M5BR5 column.)

The results for simulations using the vectorizable loops are shown in Table 4. As might be expected the overall issue rates for the vectorizable loops are higher than those for the scalar loops. There is a slight

Table 4. Multiple Issue Units, Sequential Issue for Vectorizable Code.

Number of Issue Stations	Machine							
	M11BR5		M11BR2		M5BR5		M5BR2	
	N-Bus	1-Bus	N-Bus	1-Bus	N-Bus	1-Bus	N-Bus	1-Bus
1	0.45	0.45	0.49	0.49	0.54	0.54	0.59	0.59
2	0.48	0.48	0.53	0.52	0.58	0.57	0.64	0.63
3	0.49	0.48	0.53	0.52	0.58	0.57	0.64	0.64
4	0.49	0.48	0.54	0.53	0.59	0.59	0.66	0.65
5	0.49	0.48	0.54	0.53	0.59	0.59	0.66	0.65
6	0.50	0.49	0.54	0.53	0.59	0.59	0.66	0.65
7	0.50	0.49	0.54	0.53	0.59	0.59	0.66	0.65
8	0.50	0.49	0.54	0.53	0.60	0.59	0.66	0.66

difference between the results for the N-Bus and 1-Bus organizations, but clearly the single result bus still is not saturated. The tradeoffs between multiple issue units and a faster memory or branch execution time are much the same as for the scalar code.

5.2. Multiple Issue Units with Out-of-Order Instruction Issue

In the multiple issue machine of the previous section, a blocked instruction prevents the issue of all succeeding instructions. An elaboration of this simple multiple issue scheme permits succeeding instructions to issue if they can, i.e., out-of-order issue of instructions currently in the instruction buffer is permitted. Dependencies still block instruction issue, i.e, an instruction cannot issue if it encounters a RAW or WAW hazard due to the instruction that precedes it in the instruction buffer. This out-of-order issue of multiple instructions clearly requires a significant amount of hardware to carry out the register interlock operations that must be performed (unless each instruction is expanded as in[12]). The results of simulations for this issue mechanism are presented in Table 5 and Table 6.

As one looks down the columns of these tables, the issue rate increases, but in a non-monotone manner. This non-monotone increase as the number of issue stations increases is due to the fact that only the instructions currently in the instruction buffer are candidates for issue. As with the in-order issue case, the instruction buffer is not filled until all the instructions currently in the issue buffer have been issued. As the number of issue stations increases, meaning the size of the instruction buffer increases, more

Table 5. Multiple Issue Units, Out-of-Order Issue for Scalar Code.

Number of Issue Stations	Machine							
	M11BR5		M11BR2		M5BR5		M5BR2	
	N-Bus	1-Bus	N-Bus	1-Bus	N-Bus	1-Bus	N-Bus	1-Bus
1	0.44	0.44	0.47	0.47	0.51	0.51	0.55	0.55
2	0.46	0.46	0.49	0.49	0.55	0.54	0.60	0.60
3	0.48	0.47	0.51	0.50	0.56	0.56	0.61	0.61
4	0.50	0.50	0.52	0.51	0.62	0.61	0.64	0.64
5	0.49	0.48	0.51	0.51	0.59	0.59	0.63	0.63
6	0.50	0.49	0.52	0.51	0.60	0.60	0.63	0.63
7	0.51	0.51	0.52	0.52	0.63	0.62	0.65	0.65
8	0.51	0.50	0.52	0.52	0.62	0.61	0.64	0.64

Table 6. Multiple Issue Units, Out-of-Order Issue for Vectorizable Loops.

Number of Issue Stations	Machine							
	M11BR5		M11BR2		M5BR5		M5BR2	
	N-Bus	1-Bus	N-Bus	1-Bus	N-Bus	1-Bus	N-Bus	1-Bus
1	0.45	0.45	0.49	0.49	0.54	0.54	0.59	0.59
2	0.48	0.48	0.53	0.52	0.58	0.58	0.64	0.65
3	0.50	0.49	0.54	0.53	0.59	0.59	0.65	0.65
4	0.52	0.51	0.55	0.55	0.62	0.62	0.68	0.68
5	0.51	0.50	0.54	0.53	0.61	0.60	0.66	0.66
6	0.53	0.53	0.57	0.56	0.64	0.63	0.69	0.69
7	0.54	0.53	0.57	0.56	0.65	0.64	0.69	0.69
8	0.54	0.53	0.57	0.56	0.64	0.64	0.69	0.69

instructions become available for issue. However, the way in which branch instructions fall into the buffer changes. So, there are cases where previously a branch instruction was the last instruction in the buffer and now it resides alone in the instruction buffer. This leads to the "sawtooth" pattern of the issue rates.

In comparing pairs of N-Bus and 1-Bus columns, again we see little contention for the result bus. If we look at Table 5, and compare slow versus fast branch execution for a given memory speed (M11BR5 vs. M11BR2 and M5BR5 vs. M5BR2) for 4 or more issue units, we see that a fast branch execution time has a smaller impact on performance than in the previous case. Out-of-order issue with multiple issue units permits the branch instruction to get started a little earlier than in the sequential issue case. For the

vectorizable loops, Table 6, a fast branch still offers some measurable performance improvement.

If one considers the results in Tables 3 and 4 and compares them to those of Tables 5 and 6, we can see that overall, once 4 issue stations are being used, an out-of-order issue strategy improves the issue rate by approximately 0.03. While these results indicate that a 20% to 25% improvement in issue rate can be achieved with multiple issue units, the final issue rate is much less than the issue limits calculated in Table 2. This unsatisfactory result leads us to consider, slightly different multiple issue strategies.

5.3. Multiple Issue Units with Dependency Resolution

In the issue methods presented above, instruction issue is blocked when a hazard occurs. Hazards exist due to: (i) data dependencies inherent in the algorithm and (ii) dependencies caused by limited resources such as registers. Several dependency resolution schemes were mentioned in Section 3.3. In this section, we see how dependency resolution schemes would perform with multiple issue units. Rather than discuss the performance of multiple issue units with all known dependency resolution schemes, we choose one scheme, namely the *RUU scheme* presented in[10] and [13]. The RUU scheme was selected because it guarantees precise interrupts, a feature that is desirable in a machine with multiple issue units, and because we had access to detailed simulation tools for the RUU dependency resolution scheme. Before proceeding, we give a brief description of the scheme; details can be found in [10, 13].

In the RUU scheme, reservation stations used to hold instructions that are waiting for their operands are consolidated into a single unit called the *Register Update Unit (RUU)*. Each RUU entry has a reservation station and some extra fields [10]. An instruction proceeds to the functional units from the RUU. The results from the functional units are then written back to the RUU from where they are eventually written into the register file.

When instructions issue, they are placed in the RUU where they wait until their operands become available. With N issue units, N instructions can be placed in the RUU every clock cycle unless: (i) a branch instruction is encountered or (ii) the RUU is full. Each general purpose register has an associated counter that allows the RUU to support multiple *instances* of a register. These counters are used by the issue units to obtain the correct operand tags for their source and destination operands. From the RUU, up to N instructions can proceed to the functional units for execution in any clock cycle. Results from the

head of the RUU (up to N per clock cycle) are written back into the register file when they are available. When the results leave the RUU, the corresponding RUU slots are freed for reuse by other instructions.

In our studies, we simulated a restricted N-Bus organization, i.e., each issue unit had a preassigned set of RUU slots and a preassigned set of busses throughout the machine, and a 1-Bus organization. In the 1-Bus organization, there is a single bus from the RUU to the functional units, a single bus from the functional units back to the RUU and a single bus from the RUU to the register file. Since we are interested in potential performance improvement, we allowed bypass logic in the RUU even though such logic might be quite expensive to implement.

The results of simulations for scalar code are presented in Table 7 and for vectorizable code in Table 8. We present the results for up to 4 issue units since having more than 4 issue units did not make a significant difference. From Table 7 one can see that, for scalar code, the biggest improvement from a simple CRAY-like organization comes from using dependency resolution with a single issue unit, i.e., by allowing instructions to proceed from the issue stage without waiting for their operands to become available. For example, for the M11BR5 machine, the issue rate can be increased from 0.44 to 0.73 by using a single issue unit with this dependency resolution scheme. For this scheme we could, at best, achieve an issue rate of 0.83 with 4 issue units, a 4-Bus organization and an RUU of 40 entries. We could come quite close to this maximum in achieving an issue rate of 0.76 by using only 2 issue, 2 busses and a RUU of 20 entries. As is expected, an issuing scheme that uses dependency resolution can tolerate slower memory by increasing the amount of buffer storage available (the size of the RUU). Furthermore, by using 3 issue units, we can achieve a performance of about 64%-69% of the theoretical maximum performance of 1.29 instructions per cycle.

For vectorizable code where there is more parallelism, using multiple issue units can improve the issue rate significantly. In most cases, 3-4 issue units can be used before performance starts leveling off. Beyond 4 issue units, the competition for the limited number of functional units becomes a bottleneck. When sufficient parallelism exists in the code, the use of a single result bus can be a bottleneck. Note that we can achieve an issue rate of slightly greater than 1 instruction with a single result bus since some instructions (for example branch instructions) do not use the result bus. With 4 issue units, it is possible to

Table 7: Multiple Issue Units with Dependency Resolution; Scalar Code.

Machine	RUU Size	Number of Issue Units							
		1		2		3		4	
		N-Bus	1-Bus	N-Bus	1-Bus	N-Bus	1-Bus	N-Bus	1-Bus
M11BR5	10	0.59	0.59	0.61	0.59	0.62	0.59	0.62	0.59
	20	0.67	0.67	0.76	0.69	0.79	0.69	0.79	0.69
	30	0.69	0.69	0.76	0.70	0.82	0.70	0.82	0.70
	40	0.72	0.72	0.76	0.74	0.83	0.74	0.83	0.74
	50	0.72	0.72	0.78	0.75	0.83	0.75	0.83	0.75
	100	0.72	0.72	0.78	0.75	0.83	0.75	0.83	0.75
M11BR2	10	0.60	0.60	0.61	0.60	0.62	0.60	0.62	0.60
	20	0.71	0.71	0.79	0.72	0.81	0.72	0.80	0.72
	30	0.73	0.73	0.80	0.75	0.82	0.75	0.83	0.75
	40	0.74	0.74	0.81	0.78	0.83	0.78	0.82	0.78
	50	0.74	0.74	0.83	0.78	0.83	0.78	0.83	0.78
	100	0.74	0.74	0.83	0.78	0.83	0.78	0.83	0.78
M5BR5	10	0.66	0.66	0.71	0.68	0.74	0.68	0.74	0.68
	20	0.70	0.70	0.81	0.74	0.82	0.74	0.84	0.74
	30	0.72	0.72	0.83	0.77	0.85	0.77	0.86	0.77
	40	0.75	0.75	0.84	0.80	0.86	0.80	0.87	0.80
	50	0.75	0.75	0.85	0.80	0.86	0.80	0.87	0.80
	100	0.75	0.75	0.85	0.81	0.86	0.81	0.87	0.81
M5BR2	10	0.70	0.70	0.73	0.71	0.74	0.71	0.74	0.71
	20	0.75	0.75	0.86	0.77	0.85	0.78	0.86	0.78
	30	0.78	0.78	0.87	0.80	0.88	0.81	0.87	0.81
	40	0.80	0.80	0.88	0.81	0.89	0.84	0.89	0.84
	50	0.80	0.80	0.88	0.81	0.89	0.84	0.89	0.84
	100	0.80	0.80	0.88	0.84	0.89	0.84	0.89	0.84

Table 8: Multiple Issue Units with Dependency Resolution; Vectorizable Code;

Machine	RUU Size	Number of Issue Units							
		1		2		3		4	
		N-Bus	1-Bus	N-Bus	1-Bus	N-Bus	1-Bus	N-Bus	1-Bus
M11BR5	10	0.62	0.62	0.64	0.63	0.65	0.63	0.65	0.62
	20	0.76	0.76	0.91	0.81	0.93	0.81	0.94	0.81
	30	0.80	0.80	1.04	0.86	1.10	0.86	1.13	0.86
	40	0.81	0.81	1.08	0.89	1.15	0.90	1.20	0.89
	50	0.81	0.81	1.15	0.90	1.22	0.93	1.29	0.93
	100	0.81	0.81	1.23	0.92	1.46	0.97	1.59	0.97
M11BR2	10	0.63	0.63	0.65	0.63	0.65	0.63	0.65	0.63
	20	0.81	0.81	0.96	0.85	0.97	0.85	0.98	0.85
	30	0.85	0.85	1.12	0.92	1.19	0.92	1.22	0.92
	40	0.88	0.88	1.21	0.97	1.29	0.97	1.32	0.97
	50	0.88	0.88	1.31	1.00	1.40	1.00	1.45	1.00
	100	0.88	0.88	1.44	1.03	1.73	1.03	1.87	1.03
M5BR5	10	0.73	0.73	0.78	0.74	0.78	0.74	0.79	0.74
	20	0.80	0.80	0.99	0.87	1.04	0.89	1.05	0.89
	30	0.82	0.82	1.08	0.91	1.18	0.93	1.22	0.94
	40	0.82	0.82	1.11	0.93	1.22	0.96	1.29	0.97
	50	0.82	0.82	1.16	0.94	1.29	0.97	1.35	0.97
	100	0.82	0.82	1.22	0.94	1.50	0.97	1.65	0.98
M5BR2	10	0.75	0.75	0.78	0.76	0.79	0.76	0.79	0.76
	20	0.89	0.89	1.08	0.95	1.12	0.95	1.13	0.95
	30	0.91	0.91	1.23	0.99	1.34	0.99	1.36	0.99
	40	0.91	0.91	1.29	1.02	1.40	1.02	1.47	1.02
	50	0.91	0.91	1.36	1.02	1.50	1.02	1.59	1.02
	100	0.91	0.91	1.45	1.03	1.78	1.03	2.01	1.03

achieve a performance of about 63% of the theoretical maximum performance for vectorizable code.

An interesting point to note is that, for a machine with N issue units and an N-Bus organization, the register file has N write ports and $2N$ read ports. Indeed, many of the read ports are wasted since the dependency resolution scheme allows several instructions to fetch their operands from the reservation stations in the RUU rather than from the register file itself. Likewise, the number of write ports itself could be reduced since the value of a register need not be updated if there is a succeeding instruction that will update the value of the register. To restrict the number of experiments, we did not study such cases.

6. Discussion and Conclusions

From the studies reported in this paper, we can make several observations about the design of the scalar units of multiple functional unit processors that have functional unit capabilities similar to our basic

architecture.

For code that is inherently scalar, i.e., does not have a large degree of parallelism, a simple, serial machine with a single issue unit achieves a performance of about 18%-26% of the theoretical maximum performance, depending upon the memory latency and branch time. By allowing the overlap of instructions in distinct functional units, performance could be improved to about 27%-39% of the theoretical maximum. The performance can further be improved to about 33%-41% of the theoretical maximum by interleaving the memory. With a simple instruction issuing strategy, i.e., one that enforces dependencies in the issue stage, pipelining the functional units to overlap the execution of instructions that use the same functional units will not be very beneficial - only an additional 2%-4% improvement in performance is achieved. Thus, if dependencies are enforced by the issue stage, one might choose not to pipeline a machine's functional units. This may actually increase performance by eliminating the uncertainities in propagation delays and clock skews that affect the clock speed, and thereby reduce the clock period and/or decrease the functional unit latencies.

If one reduces instruction blockage at the issue stage, the performance of a machine with a single issue unit could be improved to about 56%-62% of the theoretical maximum. Possibilities for reducing instruction blockage include software code scheduling techniques and/or hardware dependency resolution techniques. To further improve the performance of the benchmark programs, several issue units can be used, allowing several instructions to be issued in each clock cycle. By issuing multiple instructions, our base machine can achieve a performance of 60%-68% of the theoretical maximum using 2 issue units and 64%-69% of the theoretical maximum using 4 issue units. If multiple instructions are to be issued every cycle, it is crucial that steps be taken to prevent instruction blockage at the issue stage due to data dependencies.

For code that is vectorizable, i.e., has a comparatively large degree of parallelism, the situation is somewhat different. Since the inherent amount of parallelism in the code is large, theoretical limits indicate a potential for issuing, on the average, 2.78-3.15 instructions per cycle. A simple, serial machine can achieve a performance of only about 7%-9% of this theoretical maximum. By allowing the overlap of instructions in distinct functional units this performance can be improved to about 10%-14% of the theoretical maximum. If the memory is interleaved, this performance can be improved to about 15%-17% of the

theoretical maximum. Pipelining the functional units to allow overlap of instructions in the same functional unit improves performance by an additional 8%-10% (as opposed to only 2%-6% for scalar code). Eliminating instruction blockage in the single issue unit machine further improves performance to about 29% of the theoretical maximum. Additional performance improvements require the use of more than 1 issue unit. By using 2 issue units, performance can be improved to about 44%-46% of the theoretical maximum; with 3-4 issue units, performance can be improved to about 57%-64% of the theoretical maximum. As with scalar code, if multiple instructions are to be issued every cycle, it is crucial that steps be taken to prevent instruction blockage at the issue stage due to data dependencies

References

[1] M. J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, vol. 54, pp. 1901-1909, December, 1966.

[2] *CRAY-1 Computer Systems, Hardware Reference Manual.* Chippewa Falls, WI: Cray Research, Inc., 1982.

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

[4] F. H. McMahon, *FORTRAN CPU Performance Analysis.* Lawrence Livermore Laboratories, 1972.

[5] J. Worlton, "Understanding Supercomputer Benchmarks," *Datamation*, vol. 30, pp. 121-130, September, 1984.

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

[7] S. R. Kunkel and J. E. Smith, "Optimal Pipelining in Supercomputers," *Proc. 13th Annual Symposium on Computer Architecture*, pp. 404-413, June 1986.

[8] P. M. Kogge, *The Architecture of Pipelined Computers.* New York: McGraw-Hill, 1981.

[9] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, pp. 25-33, January 1967.

[10] G. S. Sohi and S. Vajapeyam, "Instruction Issue Logic for High-Performance, Interruptable Pipelined Processors," in *Proc. 14th Annual Symposium on Computer Architecture*, Pittsburgh, PA, pp. 27-36, June, 1987.

[11] S. Weiss and J. E. Smith, "Instruction Issue Logic for Pipelined Supercomputers," *Proc. 11th Annual Symposium on Computer Architecture*, pp. 110-118, June 1984.

[12] R. D. Acosta, J. Kjelstrup, and H. C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Trans. on Computers*, vol. C-35, pp. 815-828, September 1986.

[13] G. S. Sohi, "Instruction Issue Logic for High-Performance, Interruptable, Multiple Functional Unit, Pipelined Computers," Computer Sciences Technical Report #704, University of Wisconsin-Madison, Madison, WI 53706, July, 1987.