Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines

Norman P. Jouppi
David W. Wall

Digital Equipment Corporation
Western Research Lab

Abstract

Superscalar machines can issue several instructions per cycle. Superpipelined machines can issue only one instruction per cycle, but they have cycle times shorter than the latency of any functional unit. In this paper these two techniques are shown to be roughly equivalent ways of exploiting instruction-level parallelism. A parameterizable code reorganization and simulation system was developed and used to measure instruction-level parallelism for a series of benchmarks. Results of these simulations in the presence of various compiler optimizations are presented. The *average degree of superpipelining* metric is introduced. Our simulations suggest that this metric is already high for many machines. These machines already exploit all of the instruction-level parallelism available in many non-numeric applications, even without parallel instruction issue or higher degrees of pipelining.

1. Introduction

Computer designers and computer architects have been striving to improve uniprocessor computer performance since the first computer was designed. The most significant advances in uniprocessor performance have come from exploiting advances in implementation technology. Architectural innovations have also played a part, and one of the most significant of these over the last decade has been the rediscovery of RISC architectures. Now that RISC architectures have gained acceptance both in scientific and marketing circles, computer architects have been thinking of new ways to improve uniprocessor performance. Many of these proposals such as VLIW [12], superscalar, and even relatively old ideas such as vector processing try to improve computer performance by exploiting instruction-level parallelism. They take advantage of this parallelism by issuing more than one instruction per cycle explicitly (as in VLIW or superscalar machines) or implicitly (as in vector machines). In this paper we will limit ourselves to improving uniprocessor performance, and will not discuss methods of improving application performance by using multiple processors in parallel.

As an example of instruction-level parallelism, consider the two code fragments in Figure 1-1. The three instructions in (a) are independent; there are no data dependencies between them, and in theory they could all be executed in parallel. In contrast, the three instructions in (b) cannot be executed in parallel, because the second instruction uses the result of the first, and the third instruction uses the result of the second.

```
Load   C1<-23(R2)        Add    R3<-R3+1
Add    R3<-R3+1          Add    R4<-R3+R2
FPAdd  C4<-C4+C3         Store  0[R4]<-R0

(a) parallelism=3        (b) parallelism=1
```

Figure 1-1: Instruction-level parallelism

The amount of instruction-level parallelism varies widely depending on the type of code being executed. When we consider uniprocessor performance improvements due to exploitation of instruction-level parallelism, it is important to keep in mind the type of application environment. If the applications are dominated by highly parallel code (e.g., weather forecasting), any of a number of different parallel computers (e.g., vector, MIMD) would improve application performance. However, if the dominant applications have little instruction-level parallelism (e.g., compilers, editors, event-driven simulators, lisp interpreters), the performance improvements will be much smaller.

In Section 2 we present a machine taxonomy helpful for understanding the duality of operation latency and parallel instruction issue. Section 3 describes the compilation and simulation environment we used to measure the parallelism in benchmarks and its exploitation by different architectures. Section 4 presents the results of these simulations. These results confirm the duality of superscalar and superpipelined machines, and show serious limits on the instruction-level parallelism avail-

able in most applications. They also show that most classical code optimizations do nothing to relieve these limits. The importance of cache miss latencies, design complexity, and technology constraints are considered in Section 5. Section 6 summarizes the results of the paper.

2. A Machine Taxonomy

There are several different ways to execute instructions in parallel. Before we examine these methods in detail, we need to start with some definitions:

operation latency
: The time (in cycles) until the result of an instruction is available for use as an operand in a subsequent instruction. For example, if the result of an Add instruction can be used as an operand of an instruction that is issued in the cycle after the Add is issued, we say that the Add has an operation latency of one.

simple operations
: The vast majority of operations executed by the machine. Operations such as integer add, logical ops, loads, stores, branches, and even floating-point addition and multiplication are simple operations. Not included as simple operations are instructions which take an order of magnitude more time and occur less frequently, such as divide and cache misses.

instruction class
: A group of instructions all issued to the same type of functional unit.

issue latency
: The time (in cycles) required between issuing two instructions. This can vary depending on the instruction classes of the two instructions.

2.1. The Base Machine

In order to properly compare increases in performance due to exploitation of instruction-level parallelism, we define a base machine that has an execution pipestage parallelism of exactly one. This base machine is defined as follows:

- Instructions issued per cycle = 1
- Simple operation latency measured in cycles = 1
- Instruction-level parallelism required to fully utilize = 1

The one-cycle latency specifies that if one instruction follows another, the result of the first is always available for the use of the second without delay. Thus, there are never any operation-latency interlocks, stalls, or NOP's in a base machine. A pipeline diagram for a machine satisfying the requirements of a base machine is shown in Figure 2-1. The execution pipestage is cross-hatched while the others are unfilled. Note that although several instructions are executing concurrently, only one instruction is in its execution stage at any one time. Other pipestages, such as instruction fetch, decode, or write back, do not contribute to operation latency if they are bypassed, and do not contribute to control latency assuming perfect branch slot filling and/or branch prediction.

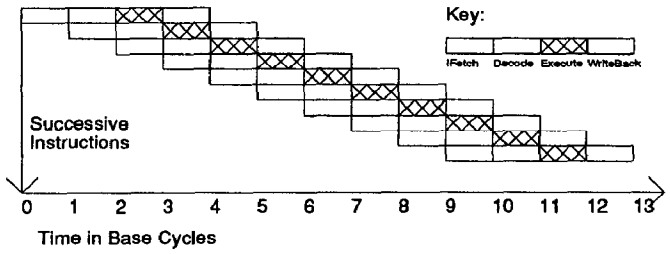

Figure 2-1: Execution in a base machine

2.2. Underpipelined Machines

The single-cycle latency of simple operations also sets the base machine cycle time. Although one could build a base machine where the cycle time was much larger than the time required for each simple operation, it would be a waste of execution time and resources. This would be an *underpipelined machine.* An underpipelined machine that executes an operation and writes back the result in the same pipestage is shown in Figure 2-2.

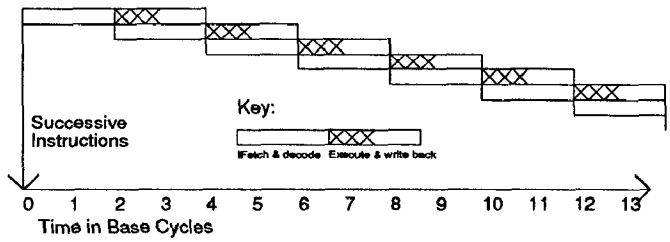

Figure 2-2: Underpipelined: cycle > operation latency

The assumption made in many paper architecture proposals is that the cycle time of a machine is many times larger than the add or load latency, and hence several adders can be stacked in series without affecting the cycle time. If this were really the case, then something would be wrong with the machine cycle time. When the add latency is given as one, for example, we assume that the time to read the operands has been piped into an earlier pipestage, and the time to write back the result has been pipelined into the next pipestage. Then the base cycle time is simply the minimum time required to do a fixed-point add and bypass the result to the next instruction. In this sense machines like the Stanford MIPS chip [8] are underpipelined, because they read operands out of the register file, do an ALU operation, and write back the result all in one cycle.

Another example of underpipelining would be a machine like the Berkeley RISC II chip [10], where loads can only be issued every other cycle. Obviously this reduces the instruction-level parallelism below one instruction per cycle. An underpipelined machine that

can only issue an instruction every other cycle is illustrated in Figure 2-3. Note that this machine's performance is the same as the machine in Figure 2-2, which is half of the performance attainable by the base machine.

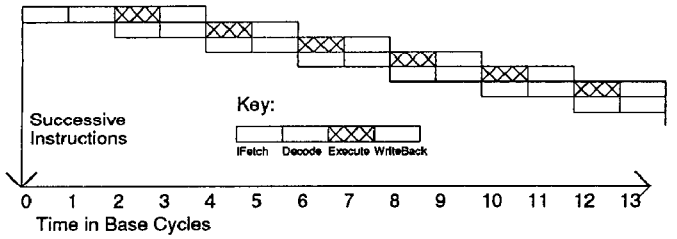

Figure 2-3: Underpipelined: issues < 1 instr. per cycle

In summary, an underpipelined machine has worse performance than the base machine because it either has:

- a cycle time greater than the latency of a simple operation, or
- it issues less than one instruction per cycle.

For this reason underpipelined machines will not be considered in the rest of this paper.

2.3. Superscalar Machines

As their name suggests, superscalar machines were originally developed as an alternative to vector machines. A superscalar machine of degree n can issue n instructions per cycle. A superscalar machine could issue all three parallel instructions in Figure 1-1(a) in the same cycle. Superscalar execution of instructions is illustrated in Figure 2-4.

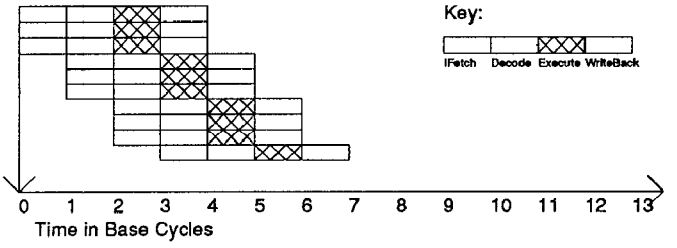

Figure 2-4: Execution in a superscalar machine (n=3)

In order to fully utilize a superscalar machine of degree n, there must be n instructions executable in parallel at all times. If an instruction-level parallelism of n is not available, stalls and dead time will result where instructions are forced to wait for the results of prior instructions.

Formalizing a superscalar machine according to our definitions:

- Instructions issued per cycle = n
- Simple operation latency measured in cycles = 1
- Instruction-level parallelism required to fully utilize = n

A superscalar machine can attain the same performance as a machine with vector hardware. Consider the operations performed when a vector machine executes a vector load chained into a vector add, with one element loaded and added per cycle. The vector machine performs four operations: load, floating-point add, a fixed-point add to generate the next load address, and a compare and branch to see if we have loaded and added the last vector element. A superscalar machine that can issue a fixed-point, floating-point, load, and a branch all in one cycle achieves the same effective parallelism.

2.3.1. VLIW Machines

VLIW, or *very long instruction word*, machines typically have instructions hundreds of bits long. Each instruction can specify many operations, so each instruction exploits instruction-level parallelism. Many performance studies have been performed on VLIW machines [12]. The execution of instructions by an ideal VLIW machine is shown in Figure 2-5. Each instruction specifies multiple operations, and this is denoted in the Figure by having multiple crosshatched execution stages in parallel for each instruction.

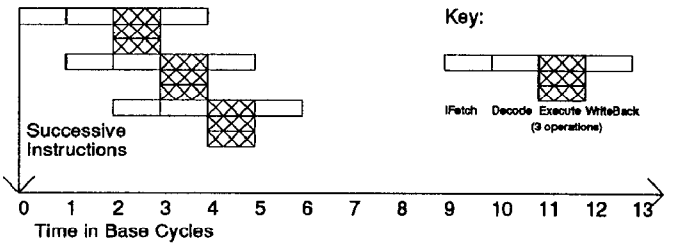

Figure 2-5: Execution in a VLIW machine

VLIW machines are much like superscalar machines, with three differences.

First, the decoding of VLIW instructions is easier than superscalar instructions. Since the VLIW instructions have a fixed format, the operations specifiable in one instruction do not exceed the resources of the machine. However in the superscalar case, the instruction decode unit must look at a sequence of instructions and base the issue of each instruction on the number of instructions already issued of each instruction class, as well as checking for data dependencies between results and operands of instructions. In effect, the selection of which operations to issue in a given cycle is performed at compile time in a VLIW machine, and at run time in a superscalar machine. Thus the instruction decode logic for the VLIW machine should be much simpler than the superscalar.

A second difference is that when the available instruction-level parallelism is less than that exploitable by the VLIW machine, the code density of the superscalar machine will be better. This is because the fixed VLIW format includes bits for unused operations while the superscalar machine only has instruction bits for useful operations.

A third difference is that a superscalar machine could be object-code compatible with a large family of non-parallel machines, but VLIW machines exploiting different amounts of parallelism would require different instruction sets. This is because the VLIW's that are able to exploit more parallelism would require larger instructions.

In spite of these differences, in terms of run time exploitation of instruction-level parallelism, the superscalar and VLIW will have similar characteristics. Because of the close relationship between these two machines, we will only discuss superscalar machines in general and not dwell further on distinctions between VLIW and superscalar machines.

2.3.2. Class Conflicts

There are two ways to develop a superscalar machine of degree n from a base machine.

1. Duplicate all functional units n times, including register ports, bypasses, busses, and instruction decode logic.
2. Duplicate only the register ports, bypasses, busses, and instruction decode logic.

Of course these two methods are extreme cases, and one could duplicate some units and not others. But if all the functional units are not duplicated, then potential class conflicts will be created. A class conflict occurs when some instruction is followed by another instruction for the same functional unit. If the busy functional unit has not been duplicated, the superscalar machine must stop issuing instructions and wait until the next cycle to issue the second instruction. Thus class conflicts can substantially reduce the parallelism exploitable by a superscalar machine. (We will not consider superscalar machines or any other machines that issue instructions out of order. Techniques to reorder instructions at compile time instead of at run time are almost as good [6, 7, 17], and are dramatically simpler than doing it in hardware.)

2.4. Superpipelined Machines

Superpipelined machines exploit instruction-level parallelism in another way. In a superpipelined machine of degree m, the cycle time is $1/m$ the cycle time of the base machine. Since a fixed-point add took a whole cycle in the base machine, given the same implementation technology it must take m cycles in the superpipelined machine. The three parallel instructions in Figure 1-1(a) would be issued in three successive cycles, and by the time the third has been issued, there are three operations in progress at the same time. Figure 2-6 shows the execution of instructions by a superpipelined machine.

Formalizing a superpipelined machine according to our definitions:

- Instructions issued per cycle = 1, but the cycle time is $1/m$ of the base machine
- Simple operation latency measured in cycles = m
- Instruction-level parallelism required to fully utilize = m

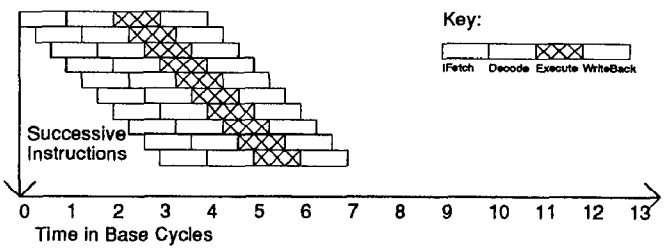

Figure 2-6: Superpipelined execution (m=3)

Superpipelined machines have been around a long time. Seymour Cray has a long history of building superpipelined machines: for example, the latency of a fixed-point add in both the CDC 6600 and the Cray-1 is 3 cycles. Note that since the functional units of the 6600 are not pipelined (two are duplicated), the 6600 is an example of a superpipelined machine with class conflicts. The CDC 7600 is probably the purest example of an existing superpipelined machine since its functional units are pipelined.

2.5. Superpipelined Superscalar Machines

Since the number of instructions issued per cycle and the cycle time are theoretically orthogonal, we could have a superpipelined superscalar machine. A superpipelined superscalar machine of degree (m,n) has a cycle time $1/m$ that of the base machine, and it can execute n instructions every cycle. This is illustrated in Figure 2-7.

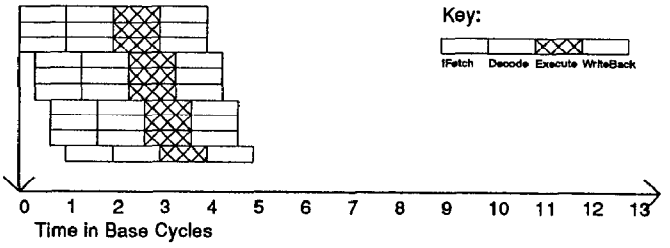

Figure 2-7: A superpipelined superscalar (n=3,m=3)

Formalizing a superpipelined superscalar machine according to our definitions:

- Instructions issued per cycle = n, and the cycle time is $1/m$ that of the base machine
- Simple operation latency measured in cycles = m
- Instruction-level parallelism required to fully utilize = $n*m$

2.6. Vector Machines

Although vector machines also take advantage of (unrolled-loop) instruction-level parallelism, whether a machine supports vectors is really independent of

whether it is a superpipelined, superscalar, or base machine. Each of these machines could have an attached vector unit. However, to the extent that the highly parallel code was run in vector mode, it would reduce the use of superpipelined or superscalar aspects of the machine to the code that had only moderate instruction-level parallelism. Figure 2-8 shows serial issue (for diagram readability only) and parallel execution of vector instructions. Each vector instruction results in a string of operations, one for each element in the vector.

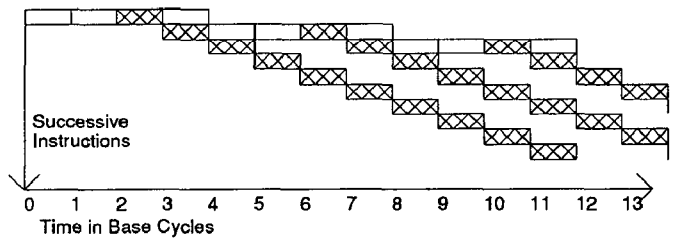

Figure 2-8: Execution in a vector machine

2.7. Supersymmetry

The most important thing to keep in mind when comparing superscalar and superpipelined machines of equal degree is that they have basically the same performance.

A superscalar machine of degree three can have three instructions executing at the same time by issuing three at the same time. The superpipelined machine can have three instructions executing at the same time by having a cycle time 1/3 that of the superscalar machine, and issuing three instructions in successive cycles. Each of these machines issues instructions at the same rate, so superscalar and superpipelined machines of equal degree have basically the same performance.

So far our assumption has been that the latency of all operations, or at least the simple operations, is one base machine cycle. As we discussed previously, no known machines have this characteristic. For example, few machines have one cycle loads without a possible data interlock either before or after the load. Similarly, few machines can execute floating-point operations in one cycle. What are the effects of longer latencies? Consider the MultiTitan [9], where ALU operations are one cycle, but loads, stores, and branches are two cycles, and all floating-point operations are three cycles. The MultiTitan is therefore a slightly superpipelined machine. If we multiply the latency of each instruction class by the frequency we observe for that instruction class when we perform our benchmark set, we get the *average degree of superpipelining*. The average degree of superpipelining is computed in Table 2-1 for the MultiTitan and the CRAY-1. To the extent that some operation latencies are greater than one base machine cycle, the remaining amount of exploitable instruction-level parallelism will be reduced. In this example, if the average degree of instruction-level parallelism in slightly parallel code is around two, the MultiTitan should not stall often because of data-dependency interlocks, but data-dependency interlocks should occur frequently on the CRAY-1.

Instr. class	Frequency	MultiTitan latency	CRAY-1 latency
logical	10%	x 1 = 0.1	x 1 = 0.1
shift	10%	x 1 = 0.1	x 2 = 0.2
add/sub	20%	x 1 = 0.2	x 3 = 0.6
load	20%	x 2 = 0.4	x11 = 2.2
store	15%	x 2 = 0.3	x 1 = 0.15
branch	15%	x 2 = 0.3	x 3 = 0.45
FP	10%	x 3 = 0.3	x 7 = 0.7
Average Degree of Superpipelining		1.7	4.4

Table 2-1: Average degree of superpipelining

3. Machine Evaluation Environment

The language system for the MultiTitan consists of an optimizing compiler (which includes the linker) and a fast instruction-level simulator. The compiler includes an intermodule register allocator and a pipeline instruction scheduler [16, 17]. For this study, we gave the system an interface that allowed us to alter the characteristics of the target machine. This interface allows us to specify details about the pipeline, functional units, cache, and register set. The language system then optimizes the code, allocates registers, and schedules the instructions for the pipeline, all according to this specification. The simulator executes the program according to the same specification.

To specify the pipeline structure and functional units, we need to be able to talk about specific instructions. We therefore group the MultiTitan operations into fourteen classes, selected so that operations in a given class are likely to have identical pipeline behavior in any machine. For example, integer add and subtract form one class, integer multiply forms another class, and single-word load forms a third class.

For each of these classes we can specify an operation latency. If an instruction requires the result of a previous instruction, the machine will stall unless the operation latency of the previous instruction has elapsed. The compile-time pipeline instruction scheduler knows this and schedules the instructions in a basic block so that the resulting stall time will be minimized.

We can also group the operations into functional units, and specify an issue latency and multiplicity for each. For instance, suppose we want to issue an instruction associated with a functional unit with issue latency 3 and multiplicity 2. This means that there are two units we might use to issue the instruction. If both are busy then the machine will stall until one is idle. It then issues the instruction on the idle unit, and that unit is unable to issue another instruction until three cycles later. The issue latency is independent of the operation latency; the former affects later operations using the same functional

unit, and the latter affects later instructions using the result of this one. In either case, the pipeline instruction scheduler tries to minimize the resulting stall time.

Superscalar machines may have an upper limit on the number of instructions that may be issued in the same cycle, independent of the availability of functional units. We can specify this upper limit. If no upper limit is desired, we can set it to the total number of functional units.

Our compiler divides the register set into two disjoint parts. It uses one part as temporaries for short-term expressions, including values loaded from variables residing in memory. It uses the other part as home locations for local and global variables that are used enough to warrant keeping them in registers rather than in memory. When number of operations executing in parallel is large, it becomes important to increase the number of registers used as temporaries. This is because using the same temporary register for two different values in the same basic block introduces an artificial dependency that can interfere with pipeline scheduling. Our interface lets us specify how the compiler should divide the registers between these two uses.

4. Results

We used our programmable reorganization and simulation system to investigate the performance of various superpipelined and superscalar machine organizations. We ran eight different benchmarks on each different configuration. All of the benchmarks are written in Modula-2 except for yacc.

ccom	Our own C compiler.
grr	A PC board router.
linpack	Linpack, double precision, unrolled 4x unless noted otherwise.
livermore	The first 14 Livermore Loops, double precision, not unrolled unless noted otherwise.
met	Metronome, a board-level timing verifier.
stan	The collection of Hennessy benchmarks from Stanford (including puzzle, tower, queens, etc.).
whet	Whetsones.
yacc	The Unix parser generator.

Unless noted otherwise, the effects of cache misses and systems effects such as interrupts and TLB misses are ignored in the simulations. Moreover, when available instruction-level parallelism is discussed, it is assumed that all operations execute in one cycle. To determine the actual number of instructions issuable per cycle in a specific machine, the available parallelism must be divided by the average operation latency.

4.1. The Duality of Latency and Parallel Issue

In section 2.7 we stated that a superpipelined machine and an ideal superscalar machine (i.e., without class conflicts) should have the same performance, since they both have the same number of instructions executing in parallel. To confirm this we simulated the eight benchmarks on an ideal base machine, and on superpipelined and ideal superscalar machines of degrees 2 through 8. Figure 4-1 shows the results of this simulation. The superpipelined machine actually has less performance than the superscalar machine, but the performance difference decreases with increasing degree.

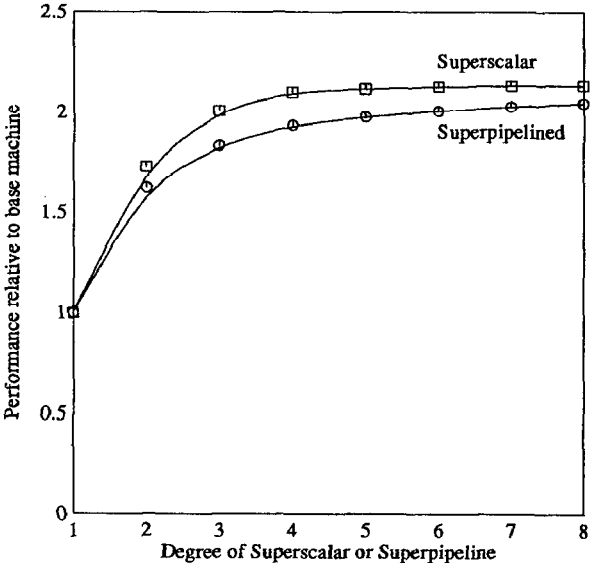

Figure 4-1: Supersymmetry

Consider a superscalar and superpipelined machine, both of degree three, issuing a basic block of six independent instructions (see Figure 4-2). The superscalar machine will issue the last instruction at time t_1 (assuming execution starts at t_0). In contrast, the superpipelined machine will take 1/3 cycle to issue each instruction, so it will not issue the last instruction until time $t_{5/3}$. Thus although the superscalar and superpipelined machines have the same number of instructions executing at the same time in the steady state, the superpipelined machine has a larger startup transient and it gets behind the superscalar machine at the start of the program and at each branch target. This effect diminishes as the degree of the superpipelined machine increases and all of the issuable instructions are issued closer and closer together. This effect is seen in Figure 4-1 as the superpipelined performance approaches that of the ideal superscalar machine with increasing degree.

Another difference between superscalar and superpipelined machines involves operation latencies that are non-integer multiples of a base machine cycle time. In particular, consider operations which can be performed in less time than a base machine cycle set by the integer add latency, such as logical operations or register-to-register moves. In a base or superscalar machine these operations would require an entire clock because that is

by definition the smallest time unit. In a superpipelined machine these instructions might be executed in one superpipelined cycle. Then in a superscalar machine of degree 3 the latency of a logical or move operation might be 2/3 longer than in a superpipelined machine of degree 3. Since the latency is longer for the superscalar machine, the superpipelined machine will perform better than a superscalar machine of equal degree. In general, when the inherent operation latency is divided by the clock period, the remainder is less on average for machines with shorter clock periods. We have not quantified the effect of this difference to date.

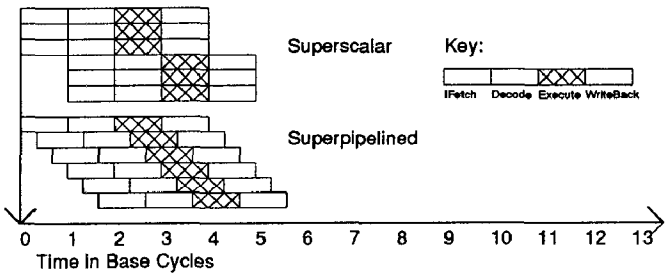

Figure 4-2: Start-up in superscalar vs. superpipelined

4.2. Limits to Instruction-Level Parallelism

Studies dating from the late 1960's and early 1970's [14, 15] and continuing today have observed average instruction-level parallelism of around 2 for code without loop unrolling. Thus, for these codes there is not much benefit gained from building a machine with superpipelining greater than degree 3 or a superscalar machine of degree greater than 3. The instruction-level parallelism required to fully utilize machines is plotted in Figure 4-3. On this graph, the *X* dimension is the degree of superscalar machine, and the *Y* dimension is the degree of superpipelining. Since a superpipelined superscalar machine of only degree (2,2) would require an instruction-level parallelism of 4, it seems unlikely that it would ever be worth building a superpipelined superscalar machine for moderately or slightly parallel code. The superpipelining axis is marked with the average degree of superpipelining in the CRAY-1 that was computed in Section 2.7. From this it is clear that vast amounts of instruction-level parallelism would be required before the issuing of multiple instructions per cycle would be warranted in the CRAY-1.

Unfortunately, latency is often ignored. For example, every time peak performance is quoted, maximum bandwidth independent of latency is given. Similarly, latency is often ignored in simulation studies. For example, instruction issue methods have been compared for the CRAY-1 assuming all functional units have 1 cycle latency [1]. This results in speedups of up to 2.7 from parallel issue of instructions, and leads to the mistaken conclusion that the CRAY-1 would benefit substantially from concurrent instruction issuing. In reality, based on Figure 4-3, we would expect the performance of the CRAY-1 to benefit very little from parallel instruction issue. We simulated the performance of the CRAY-1 assuming single cycle functional unit latency and actual functional unit latencies, and the results are given in Figure 4-4.

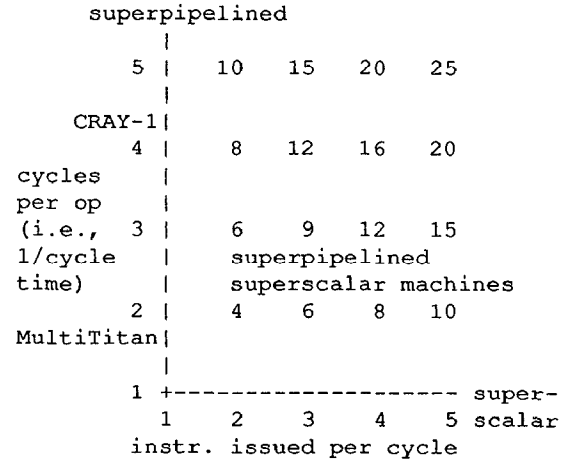

Figure 4-3: Parallelism required for full utilization

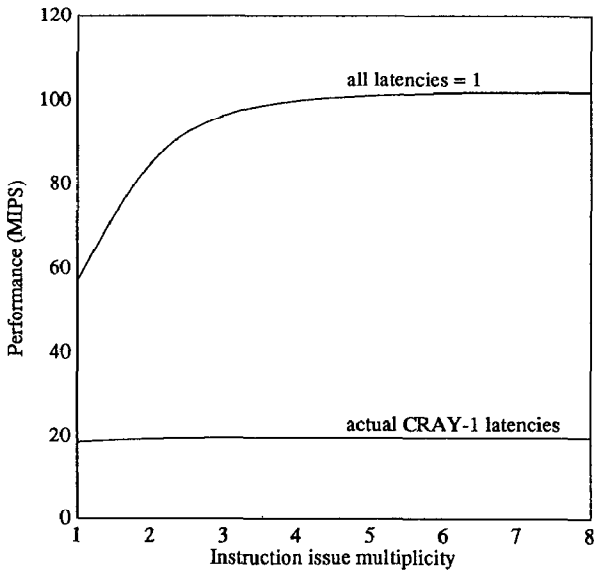

Figure 4-4: Parallel issue with unit and real latencies

As expected, since the CRAY-1 already executes several instructions concurrently due to its average degree of superpipelining of 4.4, there is almost no benefit from issuing multiple instructions per cycle when the actual functional unit latencies are taken into account.

4.3. Variations in Instruction-Level Parallelism

So far we have been plotting a single curve for the harmonic mean of all eight benchmarks. The different benchmarks actually have different amounts of instruction-level parallelism. The performance improvement in each benchmark when executed on an ideal superscalar machine of varying degree is given in Figure 4-5. Yacc has the least amount of instruction-level parallelism. Many programs have approximately two instructions executable in parallel on the average, including the C compiler, PC board router, the Stanford collection, metronome, and whetstones. The Livermore loops ap-

proaches an instruction-level parallelism of 2.5. The official version of Linpack has its inner loops unrolled four times, and has an instruction-level parallelism of 3.2. We can see that there is a factor of two difference in the amount of instruction-level parallelism available in the different benchmarks, but the ceiling is still quite low.

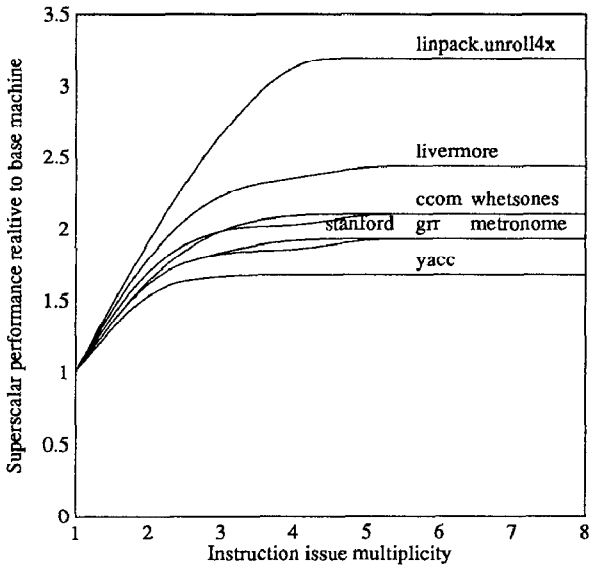

Figure 4-5: Instruction-level parallelism by benchmark

4.4. Effects of Optimizing Compilers

Compilers have been useful in detecting and exploiting instruction-level parallelism. Highly parallel loops can be vectorized [3]. Somewhat less parallel loops can be unrolled and then trace-scheduled [5] or software-pipelined [4, 11]. Even code that is only slightly parallel can be scheduled [6, 7, 17] to exploit a superscalar or superpipelined machine.

The effect of loop-unrolling on instruction-level parallelism is shown in Figure 4-6. The Linpack and Livermore benchmarks were simulated without loop unrolling and also unrolled two, four, and ten times. In either case we did the unrolling in two ways: *naively* and *carefully*. Naive unrolling consists simply of duplicating the loop body inside the loop, and allowing the normal code optimizer and scheduler to remove redundant computations and to re-order the instructions to maximize parallelism. Careful unrolling goes farther. In careful unrolling, we reassociate long strings of additions or multiplications to maximize the parallelism, and we analyze the stores in the unrolled loop so that stores from early copies of the loop do not interfere with loads in later copies. Both the naive and the careful unrolling were done by hand.

The parallelism improvement from naive unrolling is mostly flat after unrolling by four. This is largely because of false conflicts between the different copies of an unrolled loop body, imposing a sequential framework on some or all of the computation. Careful unrolling gives us a more dramatic improvement, but the parallelism available is still limited even for tenfold unrolling. One reason for this is that we have only forty temporary registers available, which limits the amount of parallelism we can exploit.

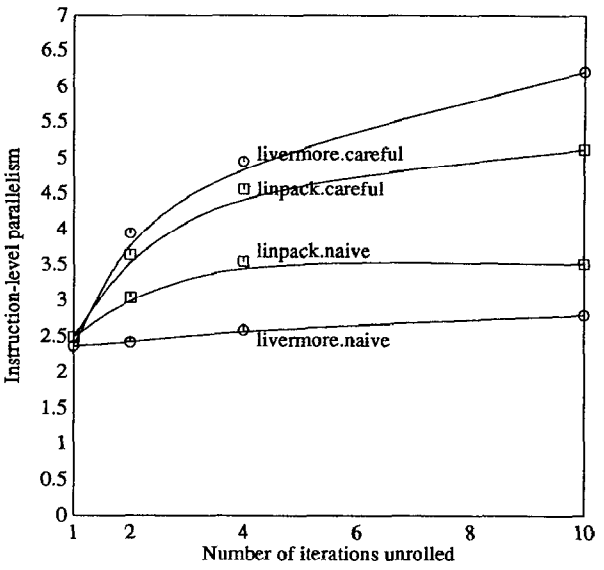

Figure 4-6: Parallelism vs. loop unrolling

In practice, the peak parallelism was quite high. The parallelism was 11 for the carefully unrolled inner loop of Linpack, and 22 for one of the carefully unrolled Livermore loops. However, in either case there is still a lot of inherently sequential computation, even in important places. Three of the Livermore loops, for example, implement recurrences that benefit little from unrolling. If we spend half the time in a very parallel inner loop, and we manage to make this inner loop take nearly zero time by executing its code in parallel, we only double the speed of the program.

In all cases, cache effects were ignored. If limited instruction caches were present, the actual performance would decline for large degrees of unrolling.

Although we see that moderate loop-unrolling can increase the instruction-level parallelism, it is dangerous to generalize this claim. Most classical optimizations [2] have little effect on the amount of parallelism available, and often actually decrease it. This makes sense; unoptimized code often contains useless or redundant computations that are removed by optimization. These useless computations give us an artificially high degree of parallelism, but we are filling the parallelism with makework.

In general, however, classical optimizations can either add to or subtract from parallelism. This is illustrated by the expression graph in Figure 4-7. If our computation consists of two branches of comparable complexity that can be executed in parallel, then optimizing one branch reduces the parallelism. On the other hand, if the computation contains a bottleneck on which other operations wait, then optimizing the bottleneck increases the parallelism. This argument holds equally well for most global optimizations, which are usually just combinations of local optimizations that require global

information to detect. For example, to move invariant code out of a loop, we just remove a large computation and replace it with a reference to a single temporary. We also insert a large computation before the loop, but if the loop is executed many times then changing the parallelism of code outside the loop won't make much difference.

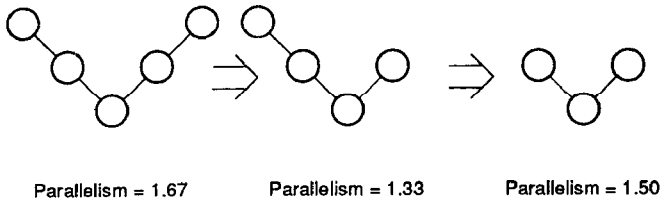

Figure 4-7: Parallelism vs. compiler optimizations

Global allocation of registers to local and global variables [16] is not usually considered a classical optimization, because it has been widespread only since the advent of machines with large register sets. However, it too can either increase or decrease parallelism. A basic block in which all variables reside in memory must load those variables into registers before it can operate on them. Since these loads can be done in parallel, we would expect to reduce the overall parallelism by globally allocating the variables to registers and removing these loads. On the other hand, assignments of new values to these variables may be easier for the pipeline scheduler to re-order if they are assignments to registers rather than stores to memory.

We simulated our test suite with various levels of optimization. Figure 4-8 shows the results. The leftmost point is the parallelism with no optimization at all. Each time we move to the right, we add a new set of optimizations. In order, these are pipeline scheduling, intra-block optimizations, global optimizations, and global register allocation. In this comparison we used 16 registers for expression temporaries and 26 for global register allocation. The dotted and dashed lines allow the different benchmarks to be distinguished, and are not otherwise significant.

Doing pipeline scheduling can increase the available parallelism by 10% to 60%. Throughout the remainder of this paper we assume that pipeline scheduling is performed. For most programs, further optimization has little effect on the instruction-level parallelism (although of course it has a large effect on the performance). On the average across our test suite, optimization reduces the parallelism, but the average reduction is very close to zero.

The behavior of the Livermore benchmark is anomalous. A large decrease in parallelism occurs when we add optimization because the inner loops of these benchmarks contain redundant address calculations that are recognized as common subexpressions. For example, without common subexpression elimination the address of A[I] would be computed twice in the expression "A[I] = A[I] + 1". It happens that these redundant calculations are not bottlenecks, so removing them decreases the parallelism.

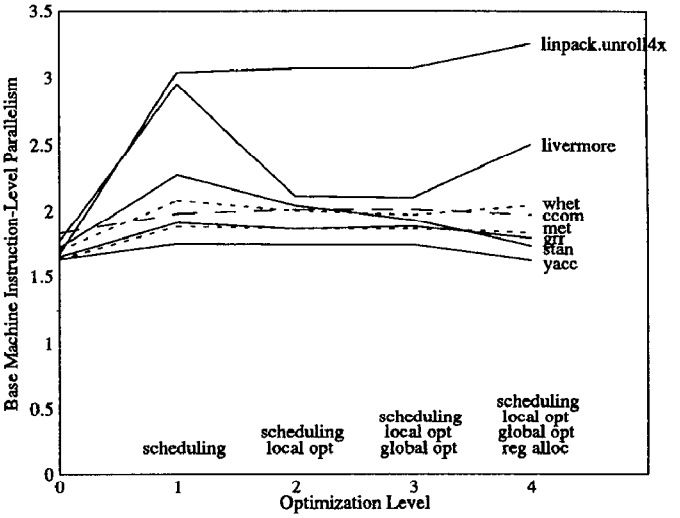

Figure 4-8: Effect of optimization on parallelism

Global register allocation causes a slight decrease in parallelism for most of the benchmarks. This is because operand loads can be done in parallel, and are removed by register allocation.

The numeric benchmarks Livermore, Linpack, and Whetstones are exceptions to this. Global register allocation increases the parallelism of these three. This is because key inner loops contain intermixed references to scalars and to array elements. Loads from the former may appear to depend on previous stores to the latter, because the scheduler must assume that two memory locations are the same unless it can prove otherwise. If global register allocation chooses to keep a scalar in a register instead of memory, this spurious dependency disappears.

In any event, it is clear that very few programs will derive an increase in the available parallelism from the application of code optimization. Programs that make heavy use of arrays may actually lose parallelism from common subexpression removal, though they may also gain parallelism from global register allocation. The net result seems hard to predict. The single optimization that does reliably increase parallelism is pipeline scheduling itself, which makes manifest the parallelism that is already present. Even the benefit from scheduling varies widely between programs.

5. Other Important Factors

The preceding simulations have concentrated on the duality of latency and parallel instruction issue under ideal circumstances. Unfortunately there are a number of other factors which will have a very important effect on machine performance in reality. In this section we will briefly discuss some of these factors.

5.1. Cache Performance

Cache performance is becoming increasingly important, and it can have a dramatic effect on speedups obtained from parallel instruction execution. Figure 5-1 lists some cache miss times and the effect of a miss on machine performance. Over the last decade, cycle time has been decreasing much faster than main memory access time. The average number of machine cycles per instruction has also been decreasing dramatically, especially when the transition from CISC machines to RISC machines is included. These two effects are multiplicative and result in tremendous increases in miss cost. For example, a cache miss on a VAX 11/780 only costs 60% of the average instruction execution. Thus even if every instruction had a cache miss, the machine performance would only slow down by 60%! However, if a RISC machine like the WRL Titan [13] has a miss, the cost is almost ten instruction times. Moreover, these trends seem to be continuing, especially the increasing ratio of memory access time to machine cycle time. In the future a cache miss on a superscalar machine executing two instructions per cycle could cost well over 100 instruction times!

Machine	cycles per instr	cycle time (ns)	mem time (ns)	miss cost cycles	miss cost instr
VAX11/780	10.0	200	1200	6	.6
WRL Titan	1.4	45	540	12	8.6
?	0.5	5	350	70	140.0

Table 5-1: The cost of cache misses

Cache miss effects decrease the benefit of parallel instruction issue. Consider a 2.0cpi (i.e., 2.0 cycles per instruction) machine, where 1.0cpi is from issuing one instruction per cycle, and 1.0 cpi is cache miss burden. Now assume the machine is given the capability to issue three instructions per cycle, to get a net decrease down to 0.5cpi for issuing instructions when data dependencies are taken into account. Performance is proportional to the inverse of the cpi change. Thus the overall performance improvement will be from 1/2.0cpi to 1/1.5cpi, or 33%. This is much less than the improvement of 1/1.0cpi to 1/0.5cpi, or 100%, as when cache misses are ignored.

5.2. Design Complexity and Technology Constraints

When machines are made more complicated in order to exploit instruction-level parallelism, care must be taken not to slow down the machine cycle time (as a result of adding the complexity) more than the speedup derived from the increased parallelism. This can happen in two ways, both of which are hard to quantify. First, the added complexity can slow down the machine by adding to the critical path, not only in terms of logic stages but in terms of greater distances to be traversed when crossing a more complicated and bigger machine. As we have seen from our analysis of the importance of latency, hiding additional complexity by adding extra pipeline stages will not make it go away. Also, the machine can be slowed down by having a fixed resource (e.g., good circuit designers) spread thinner because of a larger design. Finally, added complexity can negate performance improvements by increasing time to market. If the implementation technologies are fixed at the start of a design, and processor performance is quadrupling every three years, a one or two year slip because of extra complexity can easily negate any additional performance gained from the complexity.

Since a superpipelined machine and a superscalar machine have approximately the same performance, the decision as to whether to implement a superscalar or a superpipelined machine should be based largely on their feasibility and cost in various technologies. For example, if a TTL machine was being built from off-the-shelf components, the designers would not have the freedom to insert pipeline stages wherever they desired. For example, they would be required to use several multiplier chips in parallel (i.e., superscalar), instead of pipelining one multiplier chip more heavily (i.e., superpipelined). Another factor is the shorter cycle times required by the superpipelined machine. For example, if short cycle times are possible though the use of fast interchip signalling (e.g., ECL with terminated transmission lines), a superpipelined machine would be feasible. However, relatively slow TTL off-chip signaling might require the use of a superscalar organization. In general, if it is feasible, a superpipelined machine would be preferred since it only pipelines existing logic more heavily by adding latches instead of duplicating functional units as in the superscalar machine.

6. Concluding Comments

In this paper we have shown superscalar and superpipelined machines to be roughly equivalent ways to exploit instruction-level parallelism. The duality of latency and parallel instruction issue was documented by simulations. Ignoring class conflicts and implementation complexity, a superscalar machine will have slightly better performance (by less than 10% on our benchmarks) than a superpipelined machine of the same degree due to the larger startup transient of the superpipelined machine. However, class conflicts and the extra complexity of parallel over pipelined instruction decode could easily negate this advantage. These tradeoffs merit investigation in future work.

The available parallelism after normal optimizations and global register allocation ranges from a low of 1.6 for Yacc to 3.2 for Linpack. In heavily parallel programs like the numeric benchmarks, we can improve the parallelism somewhat by loop unrolling. However, dramatic improvements are possible only when we carefully restructure the unrolled loops. This restructuring requires us to use knowledge of operator associativity, and to do interprocedural alias analysis to determine when memory references are independent. Even when we do

this, the performance improvements are limited by the non-parallel code in the application, and the improvements in parallelism are not as large as the degree of unrolling. In any case, loop unrolling is of little use in non-parallel applications like Yacc or the C compiler.

Pipeline scheduling is necessary in order to exploit the parallelism that is available; it improved performance by around 20%. However, classical code optimization had very little effect on the parallelism available in non-numeric applications, even when it had a large effect on the performance. Optimization had a larger effect on the parallelism of numeric benchmarks, but the size and even the direction of the the effect depended heavily on the code's context and the availability of temporary registers.

Finally, many machines already exploit most of the parallelism available in non-numeric code because they can issue an instruction every cycle but have operation latencies greater than one. Thus for many applications, significant performance improvements from parallel instruction issue or higher degrees of pipelining should not be expected.

7. Acknowledgements

Jeremy Dion, Mary Jo Doherty, John Ousterhout, Richard Swan, Neil Wilhelm, and the reviewers provided valuable comments on an early draft of this paper.

References

1. Acosta, R. D., Kjelstrup, J., and Torng, H. C. "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors." *IEEE Transactions on Computers C-35*, 9 (September 1986), 815-828.

2. Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D.. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

3. Allen, Randy, and Kennedy, Ken. "Automatic Translation of FORTRAN Programs to Vector Form." *ACM Transactions on Programming Languages and Systems 9*, 4 (October 1987), 491-542.

4. Charlesworth, Alan E. "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family." *Computer 14*, 9 (September 1981), 18-27.

5. Ellis, John R. *Bulldog: A Compiler for VLIW Architectures*. Ph.D. Th., Yale University, 1985.

6. Foster, Caxton C., and Riseman, Edward M. "Percolation of Code to Enhance Parallel Dispatching and Execution." *IEEE Transactions on Computers C-21*, 12 (December 1972), 1411-1415.

7. Gross, Thomas. Code Optimization of Pipeline Constraints. Tech. Rept. 83-255, Stanford University, Computer Systems Lab, December, 1983.

8. Hennessy, John L., Jouppi, Norman P., Przybylski, Steven, Rowen, Christopher, and Gross, Thomas. Design of a High Performance VLSI Processor. Third Caltech Conference on VLSI, Computer Science Press, March, 1983, pp. 33-54.

9. Jouppi, Norman P., Dion, Jeremy, Boggs, David, and Nielsen, Michael J. K. MultiTitan: Four Architecture Papers. Tech. Rept. 87/8, Digital Equipment Corporation Western Research Lab, April, 1988.

10. Katevenis, Manolis G. H. Reduced Instruction Set Architectures for VLSI. Tech. Rept. UCB/CSD 83/141, University of California, Berkeley, Computer Science Division of EECS, October, 1983.

11. Lam, Monica. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. SIGPLAN '88 Conference on Programming Language Design and Implementation, June, 1988, pp. 318-328.

12. Nicolau, Alexandru, and Fisher, Joseph A. "Measuring the Parallelism Available for Very Long Instruction Word Architectures." *IEEE Transactions on Computers C-33*, 11 (November 1984), 968-976.

13. Nielsen, Michael J. K. Titan System Manual. Tech. Rept. 86/1, Digital Equipment Corporation Western Research Lab, September, 1986.

14. Riseman, Edward M., and Foster, Caxton C. "The Inhibition of Potential Parallelism by Conditional Jumps." *IEEE Transactions on Computers C-21*, 12 (December 1972), 1405-1411.

15. Tjaden, Garold S., and Flynn, Michael J. "Detection and Parallel Execution of Independent Instructions." *IEEE Transactions on Computers C-19*, 10 (October 1970), 889-895.

16. Wall, David W. Global Register Allocation at Link-Time. SIGPLAN '86 Conference on Compiler Construction, June, 1986, pp. 264-275.

17. Wall, David W., and Powell, Michael L. The Mahler Experience: Using an Intermediate Language as the Machine Description. Second International Conference on Architectural Support for Programming Languages and Operating Systems, IEEE Computer Society Press, October, 1987, pp. 100-104.