Understanding some simple processor-performance limits

by P. G. Emma

To understand processor performance, it is essential to use metrics that are intuitive, and it is essential to be familiar with a few aspects of a simple scalar pipeline before attempting to understand more complex structures. This paper shows that cycles per instruction (CPI) is a simple dot product of event frequencies and event penalties, and that it is far more intuitive than its more popular cousin, instructions per cycle (IPC). CPI is separable into three components that account for the inherent work, the pipeline, and the memory hierarchy, respectively. Each of these components is a fixed upper limit, or "hard bound," for the superscalar equivalent components. In the last decade, the memory-hierarchy component has become the most dominant of the three components, and in the next decade, queueing at the memory data bus will become a very significant part of this. In a reaction to this trend, an evolution in bus protocols will ensue. This paper provides a general sketch of those protocols. An underlying theme in this paper is that power constraints have been a driving force in computer architecture since the first computers were built fifty years ago. In CMOS technology, power constraints will shape future microarchitecture in a positive and surprising way. Specifically, a resurgence of the RISC approach is expected in high-performance design which will cause the client and server microarchitectures to converge.

1. Introduction

As late as the 17th century in many European universities, only the very best students were told that they could someday hope to conquer long division if they applied themselves. At that time, Roman numerals were the preferred system of numerical representation. This demonstrates that if a bad system is chosen to represent aspects of a problem, relatively simple problems can be made quite difficult.

The popular instructions per cycle (IPC) is a poor metric to use in discussing processor performance because it does not lend itself to intuition about what the components of that performance are. That is, if a pipeline that runs at x IPC has improvements made to it so that it runs at $x + \Delta$ IPC, there is no way to intuit Δ (in units of IPC) on the basis of those improvements.

The first point made in this paper is that while IPC makes for good marketing, its inverse is what makes for good engineering intuition. This paper strongly advocates the use of cycles per instruction (CPI) as the metric of choice in processor microarchitecture discussions.

0018-8646/97/$5.00 © 1997 IBM

The second point made in this paper is a corollary to this first point. That is, scalar pipeline performance (in units of CPI) can be split into three relatively independent pieces: one involving the inherent execution component of the workload (cited incorrectly as "performance" in many studies), one involving pipeline effects, and one involving the memory hierarchy. Changes to the processor microarchitecture can be translated into changes in CPI almost intuitively by considering these three components independently.

Since this is the fiftieth anniversary of electronic digital computing, von Neumann's machine built at the Institute for Advanced Studies (henceforth called the IAS machine) is used to illustrate the point. This first-generation machine shares many similarities with modern processors, and it is used to demonstrate that it is crucial to understand instruction flow in a simple processor before hoping to understand flow in a complex (e.g., superscalar) processor. This notion is self-evident, but surprisingly many resist it.

The third point made in the paper builds on the first two points: The CPI components of scalar (single decode per cycle) flow imply bounds for the CPI components of superscalar (multiple decodes per cycle) flow. Therefore, if understanding (or undertaking the development of a detailed simulator for) the flow in a superscalar processor is an overwhelming task, a great deal can be gained by understanding the flow for a much simpler (scalar) case. Given a simple understanding of that simple (scalar) case, extrapolation to the more general case can be (at least) bounded.

The fourth point is a divergence from the first three points; it focuses on an aspect of performance that will become more dominant in the coming decade, hence, on an aspect of design that must evolve. This is the performance and design of the memory bus. The simpler designs and slower cycle times of past designs have not stressed bus utilization the way future designs will. Queueing effects grow nonlinearly once utilization exceeds certain thresholds. This must be mitigated with a new approach to bus arbitration.

The final point of the paper is that as an industry, we are devoting disproportionate resources to overly complex superscalar designs because we have not found a metric that helps us to avoid those features in the microarchitecture that hurt cycle time. That metric is power consumption. A clear focus on power will ultimately lead the industry through the GHz barrier, and will converge the microarchitectures of the client and server spaces in the process.

2. Separable components of CPI

A digital computer executes an instruction stream, which is a translated statement of a high-level problem. Computer performance is "benchmarked" using instruction streams that are thought to be representative of the typical work done by the computer.

An instruction stream is a sequence of instructions that effects a sequence of events when executed on a computer. The particular events effected depend on the microarchitecture of the computer, but the frequency of those events is inherent in the instruction stream.

The basic unit of time within a computer is the cycle, and each event caused by an instruction in the program takes an integer number of cycles. Some events are concurrent, and some events take zero cycles when they occur in certain microarchitectures.

Cycles per instruction (CPI) is the most natural metric for expressing processor performance because it is the product of two measurable things:

$$CPI = (\text{cycles per event}) \cdot (\text{events per instruction}).$$

The number of cycles per event is determined by the event type for a particular microarchitecture (it can be zero for some event types), and the number of events per instruction is known for each workload (independent of the microarchitecture).

For example, consider branches in the program. If one out of five instructions in a program is a branch instruction, the event frequency is 0.2 branches per instruction for that program. If a branch causes a three-cycle delay in a processor, the CPI contribution of branches in that processor is 0.6 CPI for that program. That is, on the average, an instruction incurs 0.6 cycles of delay because of branches in the program. A performance enhancer such as a branch prediction mechanism can be thought of as a mechanism that either reduces the frequency of branches or reduces the average penalty for a branch.

Other events that can cause delay include data dependencies (instructions that are delayed because they need the results of previous instructions that have not yet executed), cache misses (referenced data that are not in the local cache and must be fetched from the memory hierarchy), and serialization events (explicit draining of the processor pipeline to ensure correct function of certain complex operations). For the most part, the frequencies of these events can be measured directly from a program (benchmark) without knowledge of the microarchitecture of the processor that will execute the program.

Therefore, a benchmark is really a specification of event frequencies, and those frequencies apply to any microarchitecture. A specific microarchitecture determines the delay (penalty) associated with each event type (which can be zero). CPI is the most natural expression of performance because it is the dot product of a benchmark (event frequencies) and a microarchitecture (event

penalties) for all event types. Further, a very small number of events account for most of the performance of a processor.

3. Some observations on von Neumann's IAS machine

Figure 1 shows a sketch of the organization of the IAS machine, which is considered to be the first stored-program computer. This sketch was taken directly from Prof. John Hayes's book [1] as he adapted it from von Neumann's verbal description of the machine [2], and it is reproduced here courtesy of Prof. Hayes and of McGraw-Hill. The machine comprises a central processing unit (CPU) that is coupled to a main memory and some simple input–output equipment (I/O). For the purposes of this discussion, we ignore the I/O portion of the figure because we are primarily concerned with the active processing part of the computer. In the case of the IAS machine, the I/O equipment is used to preload the main memory with a program and its data, and then to start the CPU. The program and its data live completely in main memory until completion of the program; i.e., the I/O does not interact with the program.

The CPU comprises a data processing unit and a program control unit. The data processing unit contains an arithmetic logic unit and some internal registers, and the program control unit contains buffer registers that hold instructions and circuitry for decoding those instructions and generating the requisite control signals that govern the operation of the aggregation in Figure 1. In modern parlance, the data processing unit is called an execution unit (EU), and the program control unit is called an instruction unit (IU).

In the IAS machine, main memory has the appearance of a large register file from the instruction-level semantic point of view, but some of the register space contains the program itself. (The instructions of the program are within the data space, and can be dynamically modified by the program.) By program construction, the program and its data fit completely in main memory. Roughly speaking, main memory has the appearance of a modern-day cache that never misses (both because the program is constrained to live in main memory and because it is semantically impossible to miss within a register file).

The operation of the IAS machine is as follows. Each instruction is fetched and executed in its entirety only after the previous instruction has completed in its entirety, and before the following instruction is fetched. Furthermore, each instruction is processed in two nonoverlapped phases. First, the instruction is fetched and decoded (I phase), and then the instruction is executed by the data processing unit (E phase). These phases are not pipelined; they are done sequentially for each instruction.

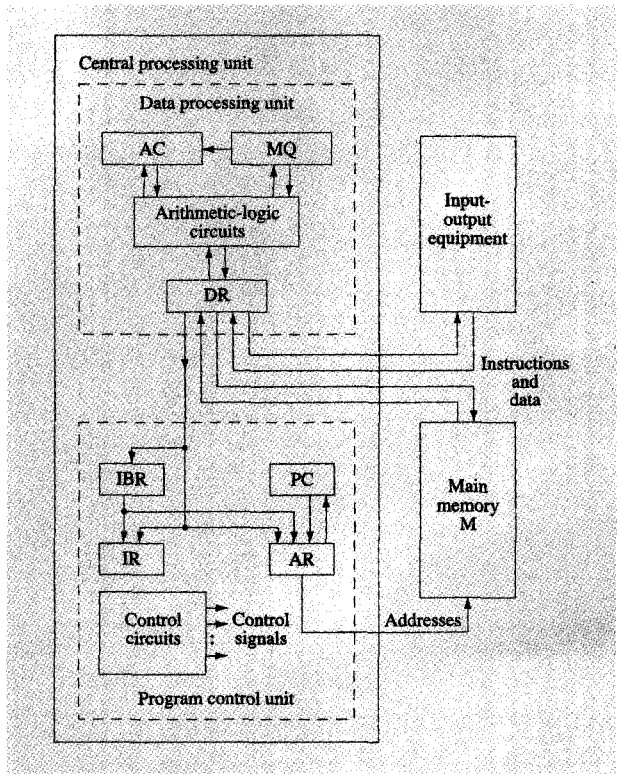

Figure 1

Structure of von Neumann's IAS processor. Reprinted with the permission of the McGraw-Hill Book Company from page 23 of the second edition of *Computer Architecture and Organization* by John P. Hayes, published by the McGraw-Hill Book Company, copyright 1988.

Note that there is a single bus between main memory and the CPU. This bus is used to transfer an instruction during the I phase, and an operand during the E phase. This is what is commonly referred to as the "von Neumann bottleneck," which is widely (and incorrectly) perceived to be the "flaw" of the von Neumann programming model. (Very interestingly, the von Neumann bottleneck has its genesis in the solution to a power-budget constraint, as is explained later.)

A programming model is the high-level part of a computer architecture. Computer architecture is the contractual interface between the hardware and the program; that is, a computer architecture is a set of rules that describe the logical function of a machine as observable by a program running on that machine.

Computer architecture does not specify what the hardware actually does; it specifies what the hardware appears to do. The entire point of defining an architecture is to isolate the programmer from those details. Architecture is a level of abstraction that allows a program

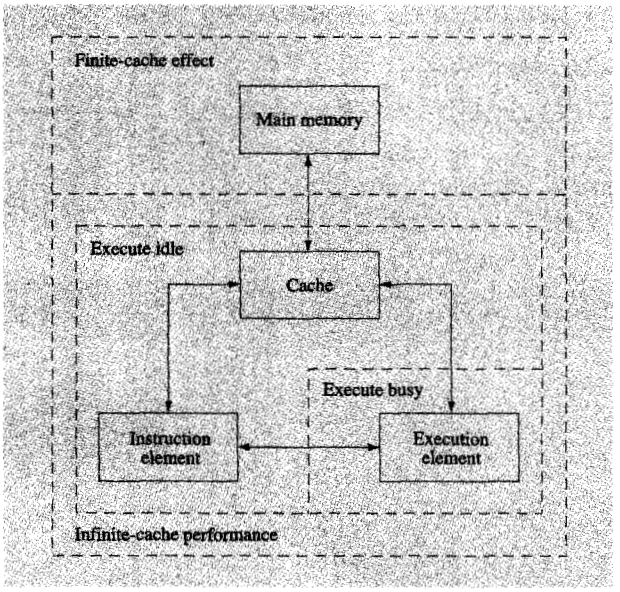

Figure 2

A modern processor: Conceptualization of performance.

to have consistent function when run on machines having different implementations (microarchitectures).

The von Neumann programming model is simply this: A program (which is a sequence of instructions) generates outputs that are consistent with the outputs generated by that program if each of the instructions in the program sequence is fetched and executed in its entirety only after all instructions preceding it have completed, and before any instructions following it are fetched. A simpler (albeit ambiguous) statement of the same thing is that "executing a sequence of instructions should have the effect of executing the instructions in the order in which they appear in the sequence," or more simply, "a sequence of instructions is executed in sequence."

The von Neumann programming model is merely a formal statement that the execution of a program should not be nonsensical. It does not imply that the hardware cannot execute instructions out of order, and it does not imply that there is only one bus or an inherent bottleneck. All it implies is that the hardware must not generate program outputs that are inconsistent with the outputs that would be generated by hardware that is constrained in the fashion implied by the programming model.

In fact, the IAS machine has a single bus to memory, and the I and E phases are not overlapped (not pipelined). Each vacuum tube in the IAS machine runs with its cathode at 200 V, and turns on when its grid exceeds 20 V. Putting 20 V on the grid is most easily accomplished by using a resistive divider, but this results in an undesirable steady-state power dissipation. To contain their power budget, the designers of the IAS machine used a more complex scheme to propagate logic values.

Instead of having resistive dividers between the stages of the arithmetic-logic unit (i.e., the carry chain), the IAS machine uses bypass capacitors to couple the stages. Therefore, instead of propagating static states (high or low voltages), the IAS machine propagates pulses.[1] (This was the first dynamic logic.)

If logic pulses are passed repeatedly through a capacitor with no quiescent period, a dc bias builds up, creating unacceptable noise susceptibility. The nonpipelined nature of the IAS microarchitecture provides a natural solution to this problem: The execution unit quiesces during the I phase, and the instruction unit quiesces during the E phase.

The von Neumann bottleneck has little to do with architecture and a lot to do with power. As described later, power budgets will steer microarchitecture in the next decade in a fairly obvious way.

4. The modern processor and its performance

Figure 2 is a high-level diagram of a modern processor. The processor core comprises an instruction unit (IU), an execution unit (EU), and a cache (C). The processor core is coupled to a main memory which can be a hierarchical system.

At this level of abstraction, the processor core is very similar to the IAS machine in many ways. (The greatest similarity is that the IAS machine never experiences a cache miss, just like any modern processor when running ISPEC.)

The major difference in the microarchitecture is that the IU and the EU operate concurrently in modern processors. This is basic "pipelining," and in a well-designed processor, the cache can accommodate the bandwidth requirements of both units running concurrently. In some modern architectures, instruction data and operand data are distinct, and reside in separate caches that are (necessarily) visible to the instruction set architecture (ISA). In the IAS machine, the two forms of data are indistinguishable, and the interpretation of a datum is contextual.

The performance components of this processor correspond to its major hardware components, which are enclosed by the dotted lines in Figure 2. Therefore, partitioning performance into these three major components (as described below) is the most natural means for understanding processor performance.

The total CPI for a processor is the sum of an infinite-cache performance and a finite-cache effect (FCE). The

[1] Related in a conversation with J. Pomerene, engineer on the IAS project.

infinite-cache performance is the CPI for the core processor under the assumption that there are no cache misses. Roughly speaking, this is raw processor speed when the effects of the memory hierarchy are ignored. The FCE accounts for the effects of the memory hierarchy.

• *Finite-cache effect (FCE)*

The FCE is the CPI associated with cache misses:

FCE = (cycles per miss) · (misses per instruction).

The misses-per-instruction component is commonly called the miss rate, and the cycles-per-miss component is called the miss penalty. (In the literature, the terms "miss rate" and "miss ratio" are often used interchangeably. This is incorrect. The miss ratio is the ratio of misses to references, and this can only be related to the instruction frequency by knowing the average number of references per instruction.)

The beauty of separating the FCE in this way is that for a given cache geometry (size, line size, and set associativity), it allows a benchmark to be characterized by its miss rate quite independently of the details of the cache-to-memory interface. That miss rate applies to any other processor whose cache shares the same geometry.

Measuring the miss rate does not require complex simulation, but it requires simulating many misses (e.g., millions) for an accurate measurement. A miss rate simulation merely requires a data structure that reflects the cache geometry (the "simulated cache"), and a long reference stream to run through that data structure to effect replacements (misses). The simulation does not require any level of detail, or any emulation of actual hardware mechanisms.

Characterization of the miss penalty requires a detailed simulation of the processor and the memory system to which it is coupled, but an accurate characterization can be done with a relatively small number of misses (e.g., thousands). That is, to characterize miss penalty, the actual hardware mechanisms that process a miss must be simulated in detail, and they must be simulated within the context of the processor. This ensures certainty regarding the average delay associated with a miss, and this average converges rapidly.

Therefore, the complex simulation (to characterize miss penalty) can be fairly short, and the lengthy simulation (to characterize miss rate) can be very simple. It is also true that a (good) performance practitioner can make a fairly accurate guess as to miss penalty for a given processor-memory structure if he has experience with similar structures (i.e., detailed simulation of derivative designs might be unnecessary for someone who is knowledgeable).

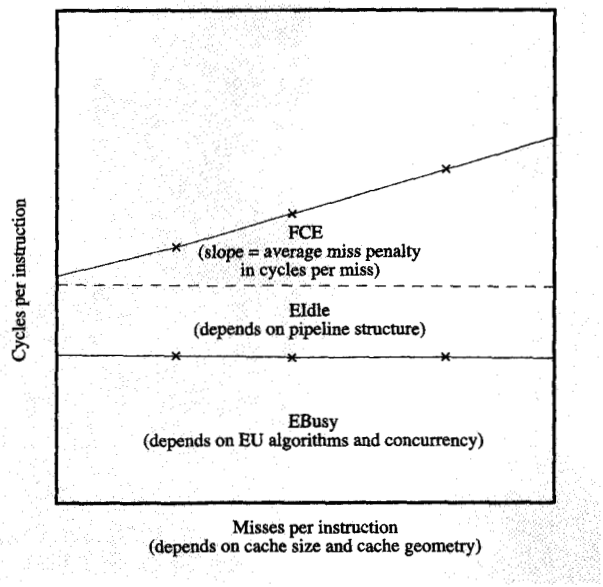

Figure 3

CPI as a function of miss rate.

Note that simulation is unnecessary in the trivial case of a blocking cache, because nothing overlaps the processing of the miss.

Figure 3 shows an idealized plot of a processor's CPI as a function of miss rate. Note that the slope of the finite-cache CPI is the miss penalty (cycles per miss), and the y-intercept for this line is the infinite-cache CPI. Since the infinite-cache CPI is independent of miss rate, it is horizontal.

Therefore, if the infinite-cache CPI can be determined for a processor, this fixes the y-intercept. If the miss penalty can be determined, this fixes the slope of the finite-cache CPI, and that line can be drawn. For a given miss rate (x coordinate), the CPI is determined.

• *Infinite-cache performance*

Measuring infinite-cache performance requires a detailed simulation of the processor core. This simulation (using no cache misses) can be relatively short if there is reasonable confidence that both the instruction mix and the software-module mix are representative of steady-state performance.

The infinite-cache CPI is the sum of two parts (each of which is independent of the miss rate), as is shown in Figures 2 and 3. These are the execute busy (EBusy) and the execute idle (EIdle) components.

EBusy represents the average intrinsic work (execution cycles) done per instruction. In a scalar processor, EBusy corresponds to the CPI that would be obtained if there

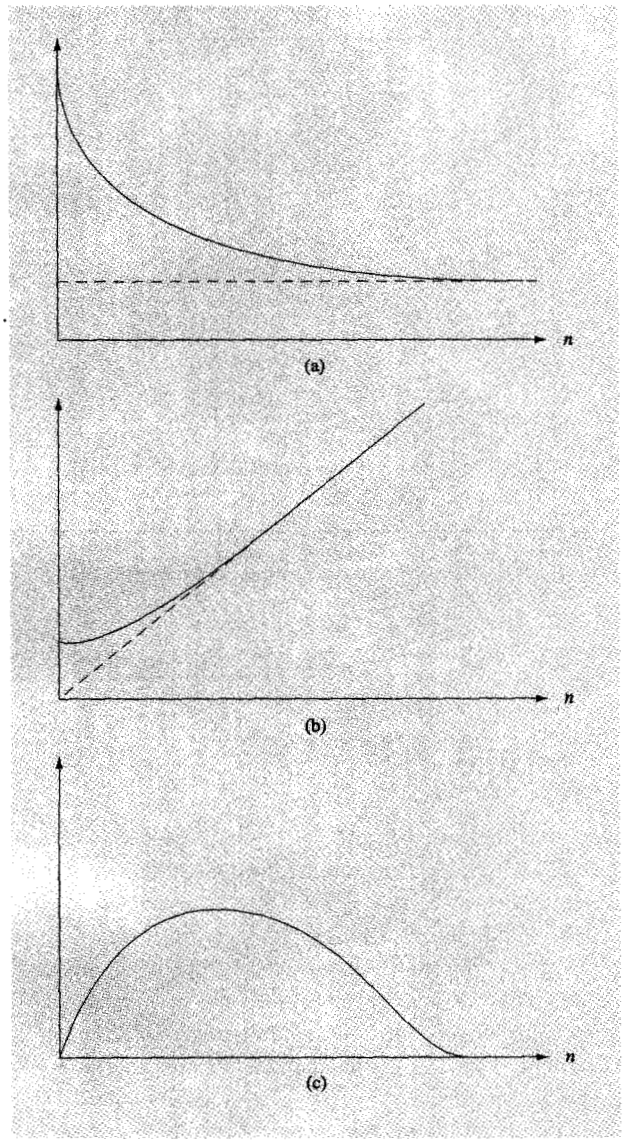

Figure 4

Performance as a function of pipeline granularity: (a) cycle time; (b) CPI; (c) MIPS.

were no pipeline delays. Another interpretation of EBusy is infinite-cache CPI with the EU utilization at 100%. For a scalar RISC machine, EBusy = 1; for a CISC machine, EBusy > 1. Note that EBusy is independent of pipeline delays. EBusy is the dot product of the instruction frequencies and their execution times; it is a pure measure of intrinsic work.

EIdle represents the average delay per instruction caused by pipeline effects. Very simply, EIdle is the difference between the infinite-cache CPI and EBusy. In general, it accounts for all of the numerous pipeline effects, but only a few of these are really significant. EIdle can also be separated into a number of relatively independent components [3], although this is not done in detail here.

As can be surmised from some of the previous discussion, the first crude separation of EIdle is as follows:

1. Events in which instructions are not available in time to keep the EU busy. These are branches, dynamic effects associated with cache utilization, and effects associated with the fetching of variable-length instructions (if applicable).
2. Events in which operand data are not ready in time to keep the EU busy. The most predominant is the address generate interlock (AGI) associated with indirect data fetching (e.g., pointer chasing), but execution dependencies in general fall into this category.
3. Serialization events in which the pipeline is deliberately drained between instructions to ensure correct function. These events are predominant in MP environments.

Note that each of these events has a frequency that can be measured for any benchmark, and that frequency is independent of the microarchitecture. Each event has an associated penalty that can be determined from the microarchitecture. The CPI contribution of each event type has the form

CPI = (cycles per event) · (events per instruction).

A further interesting observation is that most of the EIdle events, and all of the predominant ones (branches, AGIs, and serialization), have penalties that are directly proportional to the number of stages in the pipeline (in general), and specifically to the number of cycles required for a cache access. That is, EIdle CPI varies linearly with the number of stages in the pipeline (i.e., with the degree of pipelining).

Superpipelining and wave pipelining are techniques that achieve a fast cycle time by making the pipeline granularity fine. It has been shown that processor cycle time decays to an asymptote when the pipeline granularity is increased, while CPI increases linearly, as shown respectively in **Figures 4(a)** and **4(b)** [4]. The performance (MIPS) is the inverse of the product of the two, and that product has a quadratic form (thus, a unique maximum), as shown in **Figure 4(c)**.

Because of unpredictable branching, and because of a high AGI frequency due to heavy pointer use, EIdle events in commercial code render superpipelining techniques fruitless. These techniques are better suited to the realm of scientific workloads.

• *Real workloads*

Figure 3 is an idealized plot, but in fact, real workloads exhibit a small correlation between EBusy and miss rate. This does not mean that miss rate is causal to EBusy; it merely means that there is a statistical dependence (correlation). This is why rigorous benchmark selection requires an accurate module mix. An accurate instruction mix alone is insufficient.

Furthermore, the actual finite-cache CPI becomes superlinear when the miss rate is very low or very high, for different reasons:

1. When the miss rate is very low, the misses do not cluster (in time) quite as much as they would in steady state; therefore, no portions of misses overlap each other. The net effect is that the average miss penalty is slightly larger.
2. When the miss rate is very high, the memory system and its buses become saturated servers. The misses serviced by the saturated system then incur inordinate queueing delays, and the average miss penalty increases. This effect is nonlinear, and is discussed later in this paper.

• *Out-of-order execution in modern pipelines*

Tomasulo described a "common data-bus algorithm" that permits out-of-order execution in the floating-point hardware of the System/360* Model 91 computer [5]. In the 1960s, floating-point operations required multiple execution cycles, so allowing a logically subsequent instruction to bypass an operation that was in progress helped performance.

The Tomasulo algorithm was extended for use in the S/390* ES/9000* family of processors to include register renaming and a means for taking precise interrupts while executing out of order [6]. The general scheme is described by Smith and Pleszkun [7], and is now used in many modern processors.

Two aspects of the general scheme can be compelling under certain circumstances:

1. If the number of architected registers is insufficient for supporting the maximum number of instructions that can be in progress at any time (proportional to the product of the number of pipeline stages and the number of instructions decoded per cycle), register renaming provides a means to increase the number of physical registers so as to support this peak rate.
2. Register renaming can simplify the control required to take precise interrupts and/or to initiate retry operations.

However, the out-of-order aspect of the broad scheme is generally not valuable in modern processors, where the number of execution cycles needed per instruction is almost always one (exceptions are divide, square root, and some decimal operations).

Out-of-order execution is not usually valuable because in-order flow is generally maintained in all of the pipeline except for the execution phase, which is usually a single cycle. Specifically, the in-order decoding of instructions is required so that the hardware can make sense of the logical instruction sequence, and in-order instruction completion is required so that exceptions are handled properly and interrupts are taken precisely. Between the decoding and the completion of an instruction, CISC pipelines access an operand and then execute the instruction, and RISC pipelines merely execute the instruction. In general, the ordering of operand accesses is preserved to satisfy constraints imposed by MP-coherency issues (specifically, fetches are kept in order, stores are kept in order, and fetches and stores to the same location are kept in order).

Therefore, the situation in modern processors is that order within the pipeline is maintained up until execution, it is restored immediately after execution, and execution takes a single cycle. Enabling out-of-order execution in this environment is not worthwhile, because it will seldom happen. If enabling out-of-order execution complicates a design, it should not be done.

Further, if register renaming is used specifically to enable out-of-order execution (i.e., if the two circumstances listed above are not motivating factors), it should certainly not be done. Not only will this mechanism fail to improve performance, but the renaming steps will lengthen the pipeline (or will have a cycle-time impact) which will increase CPI as previously described, thereby actually hurting performance.

In addition to arithmetic multicycle operations, there are other classes of multicycle operations in some instruction-set architectures. Specifically, there can be long moves, and there can be operating-system assist instructions. In both cases, the opportunity for them to overlap other operations is very slight. Long moves fully utilize critical resources (e.g., buses) and prevent much else from happening while they are in progress. Operating-system assists are usually serialization events which explicitly preclude overlap.

• *High-MIPS effects and workload characterization*

The initial velocity of a falling object is determined predominantly by gravity. As this predominant effect increases the velocity, second-order effects (which increase with velocity) gain significance, and work against the effect of gravity. In physics, this situation is clearly described by a simple differential equation in which the object reaches a terminal (constant) velocity. The falling object can go no faster than this.

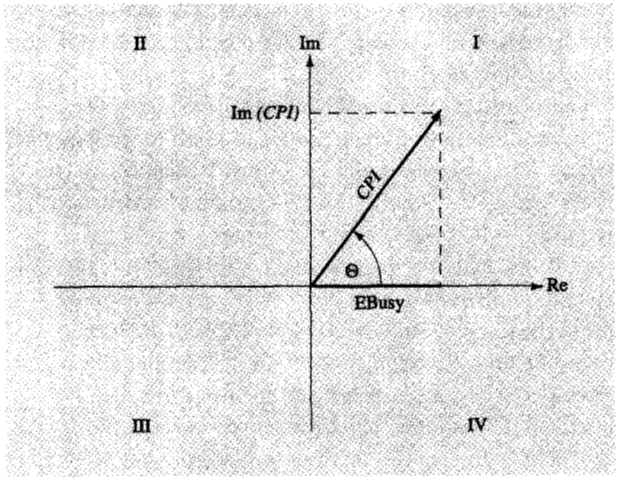

Figure 5

CPI in the complex number domain.

Similar phenomena affect processor speed, although the effects cannot be crisply described by differential equations. Collectively "high-MIPS" effects (HMEs) are effects that tend to slow a processor as it runs faster. Most of the HMEs arise because the latency (not the bandwidth) associated with the I/O subsystem does not scale with processor speed.

In a multiprogramming environment, as the processor speeds up (via faster circuits, higher ILP, or higher degree of MP), the latency of the I/O subsystem appears larger from the perspective of an executing program. If the multiprogramming level is maintained, the added speed causes the processor utilization to drop (i.e., the added speed is not utilized). To maintain a high utilization, the multiprogramming level is increased.

When the multiprogramming level increases, the software queues become longer, which causes dispatchers and other operating-system modules to utilize a higher percentage of the system time. Operating systems compensate for this by using different algorithms, which changes the basic instruction mix somewhat. Further, the larger number of coexistent processes generate a higher rate of synchronization events, and those events must synchronize with more processes. Since the aggregate "footprint" of the coexistent processes is larger, the cache miss rate and the TLB miss rate will be larger for each process. These effects all slow the system down.

Note that as processes (mainstream and I/O) and processors (mainstream and I/O) interact, they necessarily spend some portion of their time in wait-loops waiting for asynchronous events to complete (e.g., a page-in). As the number of processors in a multiprocessor increases, as those processors run faster, and as the degree of multiprogramming increases, the portion of time spent waiting increases.

When measuring the real performance of a running system by measuring instruction throughput (MIPS), it is important to subtract out that portion of the performance that is spent in wait loops (or other analogous code). Not only are wait loops nonproductive (and should not be included in any measure of "work"), but they can actually run very fast, and can bias a measurement of MIPS in an artificially favorable way.

5. IPC and other metrics that defy intuition

Thus far, the case has been made for CPI as the performance metric that most naturally lends itself to intuitive interpretation and simple estimation techniques. CPI is a dot product involving a set of event types, where the contribution of each event type is the product of its frequency and its associated penalty. The frequencies are intrinsic to workloads, and the penalties can be readily derived from the microarchitecture.

An experienced designer should know approximate numbers for key events, key workloads, and basic microarchitectures. Changes to a microarchitecture rarely affect more than one or two event types, and that effect should be readily understood by an experienced designer. Quick estimation using this dot product is simple and remarkably accurate.

IPC is the inverse of CPI. It defies intuition while hinting at high performance. Merely intoning "instructions per cycle" creates a subliminal feeling that massive instruction-level parallelism (ILP) is being achieved. It is not.

When performance is expressed as IPC, EBusy and EIdle become intermingled and inseparable. This does not clarify performance; it shrouds it. The use of IPC has been the largest impediment to the understanding of pipeline effects, and the largest contributor to the notion that EIdle defies intuitive quantification. It has been the largest impediment to simple analyses of superscalar and VLIW processor designs.

In fact, so many papers quote some theoretical maximum speed in the absence of any possible accounting for pipeline effects as "performance," that for several years we assumed that most of the industry was measuring CPI in the complex-number domain, as shown in **Figure 5**.

- *Complex CPI*

As was explained in the previous section on infinite-cache performance,

$$CPI = EBusy + EIdle.$$

On the basis of the numbers quoted in many papers, the only reasonable conclusion that is possible is that

published performance is just EBusy, which can be interpreted as a projection of complex CPI onto the real axis,

$$\mathrm{Re}\ (CPI) = EBusy,$$

where the angle associated with the projection is

$$\Theta = \arccos\left(\frac{EBusy}{EBusy + EIdle}\right).$$

The corresponding projection onto the imaginary axis is

$$\mathrm{Im}\ (CPI) = (EBusy + EIdle)\sin(\Theta).$$

Think of the imaginary component of CPI as that component of performance many authors would like to imagine they need not deal with. Complex CPI is

$$CPI = EBusy + i\ \mathrm{Im}\ (CPI).$$

A good indicator of the usefulness of a measurement is the sensitivity of that measurement to a parameter of interest. Since changes to a microarchitecture usually target EIdle (i.e., improving EU utilization), the sensitivity of CPI to EIdle is this indicator. This is

$$\frac{d(CPI)}{d(EIdle)} = i\left[\sin\left(\arccos\frac{EBusy}{EBusy + EIdle}\right) + \frac{EBusy^2}{(EBusy + EIdle)\sqrt{EIdle(2EBusy + EIdle)}}\right],$$

which is a purely imaginary complex trigonometric function that is messy, hence unintuitive. (It also happens that if EBusy and EIdle are comparable, this sensitivity is small, so complex CPI is not a useful measurement anyway.)

Thus, just as for the inverse of CPI (IPC), it is apparent that the complex form of CPI does not lead to clear intuition. Nonetheless, computing complex CPI from published performance numbers does yield another useful metric. In particular, Θ is a direct measure of the degree to which the published number is out of phase with reality.

6. Limits of superscalar performance as understood from scalar components

Thus far, discussion has focused on performance for pipelined machines that issue a single instruction per cycle. It was argued that CPI can be partitioned into independent pieces for such a machine, and those pieces can be easily understood.

Superscalar processors decode and execute multiple instructions per cycle, and the execution can be done out of order with respect to the decoding. Understanding the pipelined flow of multiple instructions per cycle (possibly out of order) is more difficult than understanding the flow of single instructions per cycle. Similarly, implementing a superscalar simulator is more difficult than implementing a scalar simulator.

For some proposed microarchitectures, the tasks of understanding performance and of simulating performance seem overwhelming. While the dissection of superscalar performance into its components is not done in this paper, its key is a solid grasp of the analogous scalar flow. Frequently, this grasp is sufficient for making very good estimates. Understanding superscalar performance without understanding the analogous scalar flow is hopeless.

Every designer and performance analyst who is working on a superscalar processor without a detailed model of such and without the resources or time to construct one should construct (at least) the analogous scalar model. That is, construct a model of the same pipeline, and study that pipeline running in scalar mode.

The scalar performance components are hard bounds (fixed CPI limits) for the superscalar components; they give definite indications of what levels of performance are and are not achievable in superscalar mode. In particular, the FCE is identical; scalar EBusy is an upper bound; and scalar EIdle is a lower bound.

Recall that FCE is the product of the miss rate and the miss penalty. The miss penalty depends on the memory hierarchy, not on the microarchitecture of the processor. The miss rate (misses per instruction) is a characteristic of the workload, and is independent of the rate at which instructions are executed. (Note that prefetching mechanisms, branch prediction mechanisms, and explicit speculation can increase the miss rate, but that increase can be accurately estimated without a detailed model of the processor.) Therefore, it is an excellent approximation that the FCE in superscalar mode is identical to the FCE in scalar mode. (This paper does not address multithreading; suffice it to say that it makes the FCE component worse, and it should not be used if a primary goal is high performance.)

EBusy is the inherent work done by the program. This work is identical in scalar and superscalar modes. The difference is that in superscalar mode, some of the work is done in parallel. As such, the scalar EBusy is an upper bound, and the superscalar EBusy should be smaller. If there are n functional units, it cannot be more than n times smaller. Estimates that approach EBusy/n are probably wrong.

EIdle is (primarily) a measure of the functional interlocks that are intrinsic to the program, e.g., branches, AGIs, and serialization events. The penalties associated with these intrinsic interlocks depend on the pipeline, and on the temporal proximities of the interlocks to the events that resolve them (e.g., AGI).

The interlocks do not disappear in superscalar mode; on the contrary, their effects are amplified. If superscalar

operation is having the desired effect (i.e., the execution rate is increased), the temporal proximities of interlocks are shortened with respect to the events that resolve them. That is, sequences of instructions are compressed into a smaller temporal "window," and the resolving events have less time to resolve the interlocks before the interlocks stop pipeline flow.

Therefore, the intrinsic part of scalar EIdle is a lower bound. As the execution rate increases, EIdle can only become larger.

There are also EIdle effects that arise because of resource conflicts (e.g., insufficient buffering, buses, or ports). These are called "bottlenecks." They are (generally) extrinsic to the program, and should be relatively small. If these effects are manifest in a scalar design, the design is poor. Design resource should be directed at these bottlenecks instead of at a superscalar control structure.

If these effects are ignorable in a scalar design, they nonetheless might be manifest in a superscalar design if buffers and bandwidth-related resources are not scaled accordingly. Therefore, as was true of the intrinsic EIdle, extrinsic EIdle in a scalar design (if any) can only become worse in a superscalar design. Scalar EIdle is a lower bound.

Of the three scalar bounds, EIdle is the most difficult to extrapolate to the superscalar realm. As was discussed in the subsection on infinite-cache performance, the principal components of EIdle have penalties that are directly proportional to the number of cycles required for a cache access. For this reason, a designer should know the coefficient of infinite-cache CPI with respect to this number. Recall that this is strictly linear, so the increase in CPI per cycle increase in the cache-access time is constant. This coefficient is useful in quickly evaluating whether a potential reduction in miss rate from a larger cache justifies the potential exposure of a longer cache-access time.

- *Small fast L0 caches*

A caveat in regard to the aforementioned pipeline coefficient is that it should not be used to extrapolate CPI downward into the realm of L0 caches. L0 caches cannot be conceptualized in the same manner as L1 caches because their (rare) benefits cannot be quantified in terms of hit rate. This is because L0 caches are very small and have hit rates that are below the "critical mass" required to conceptualize their steady-state operation.

Specifically, the quanta of pipeline flow are basic blocks which are sequences of instructions between successive taken branches (or between successive mispredicted branches). If all data and instruction references in a basic block are satisfied by the L1 cache, that basic block "sees" the pipeline flow involving the L1 access path. If L1 misses are rare (relative to the number of instructions in the basic block), "the pipeline" is the pipeline associated with the L1, and L1 misses can be treated as isolated events that are accounted for in the FCE.

The L0 cache allows the basic block to start early (assuming that the initial access of the block hits in the L0), and therefore to finish early (assuming that all references in the basic block hit in the L0). If any access in the basic block misses in the L0 and requires an L1 access, the flow for that block reverts to the L1 pipeline flow, and the "head start" afforded by the L0 is lost.

Therefore, only those basic blocks that live completely in the L0 benefit from the L0. For typical L0 sizes and miss rates, most workloads do not have a predominance of basic blocks with this characteristic. In any case, the probability of losing performance due to L0 misses is not proportional to the L0 miss rate; instead, it is closer to a geometric distribution of the miss rate whose degree is proportional to the average number of accesses per basic block.

- *The disadvantage of decoupling the I and E phases*

A model of high-performance computing that has become pervasive in the last several years has the I phase decoupled from the E phase. The two phases are implemented with engines that are autonomous with respect to each other, and they operate on a common instruction queue.

In this model, the I engine is directed by an extremely accurate branch-prediction mechanism, and it is able to run far ahead of the E engine, fetching instructions at a high rate and dispatching them to the common instruction queue. The E engine is a superscalar dataflow machine. The dataflow analyzer chooses instructions that are not interlocked, and dispatches them out of order (if necessary) to a set of parallel execution elements at a hypothetically high rate. The cache and the register file are assumed to be highly multiported so that they do not bottleneck execution.

Many studies that involve this model ignore the details of the I phase and assume that it can keep ahead of the E phase. The focus then shifts to the dataflow analyzer, which is a more tractable problem. If the constraints imposed by considerations for an MP environment are loosened (specifically, the ordering of fetches and the completion of stores), and if the cache miss rate is extremely low, the execution time is a very impressive (and very unsurprising) number that is proportional to the depth of the dataflow graph.

This model yields very impressive results that are flawed on several levels. The first level is a pure "catch-22" that does not require any understanding of microarchitecture: If the E phase accelerates to its theoretical speed limit (determined by the dataflow graph), it must catch up with

the I phase. Therefore, there cannot be instructions in the common queue to dispatch (i.e., if the E phase goes as fast as it is supposed to, the processor runs out of instructions). The point is that the I phase is not ignorable.

In real commercial workloads, the frequency of unpredictable branches prevents the I phase from getting far ahead of the E phase. Given the best techniques available, a branch will be guessed wrong every 20–30 instructions in real commercial code. (In SPEC**, the numbers are much better.) Therefore, the theoretical maximum execution rate can only be maintained for intervals of 20–30 instructions, and each interval is followed by a wrong-guess recovery action that requires restarting the I phase. For real hardware to approach the bounds implied by this model, branch prediction must become much better than it is today (except in SPEC).

In real commercial code, an L1 cache miss occurs every 10–20 instructions for L1 sizes in the 64KB to 256KB range (in SPEC, the numbers are much better); thus, maintaining a high execution rate requires moving cache lines at a rate corresponding to the resulting miss frequency. This is not realistic at very high execution rates. For example, if a processor can execute five instructions per cycle, and there is an L1 miss every ten instructions, the memory hierarchy must be able to move an entire cache line in two processor cycles for the processor to sustain this rate. (This is still an unrealistically optimistic statement, because it assumes that there is no miss latency, which would require perfect prefetching.)

Another characteristic of real commercial code is that a majority of conditional branches are resolved by instructions that have an AGI interlock with an immediately preceding instruction. (An example is a loop that searches for a record in a linked list.) When this is the case, it is disadvantageous to separate (temporally) the I and E phases by adding pipeline stages (the cycles required to put instructions into a queue, analyze them, and redispatch them). Instead, it is desirable to couple the phases tightly so that there is minimal latency between the initial dispatch of a load instruction and its execution outcome. (This is much less of a constraint in SPEC.)

7. Future trends in cache miss penalty

In high-performance systems of the 1980s, cost was a less crucial factor than it has become in the 1990s. Symmetric multiprocessor (SMP) systems of the 1980s typically had private unidirectional point-to-point buses between processors and the memory system. Those buses were driven by water-cooled circuits onto low-dielectric packaging, and they ran at the same speed as the processor. Cache line sizes were moderate by today's standards.

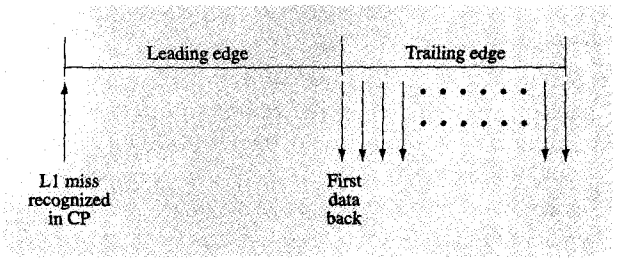

Figure 6

Temporal components of a cache miss.

In the 1990s, processors have become faster (shorter cycle time) and buses have not kept pace. There are many systems in which the processor runs at a higher frequency than the bus, and that difference is becoming larger. The number of processors in SMPs has increased, and to contain costs, many systems are shared-bus systems. With the growth of object-oriented programming, miss rates have increased, and with higher instruction-level parallelism, miss frequencies (in time) are larger for the same miss rates. Further, it is becoming the fashion to increase line size.

Each of these trends increases bus utilization, and thus exacerbates the effects of bus utilization. Bus queueing (which was an ignorable effect in high-performance systems of the 1980s) will dominate system-level performance in the next decade if new protocols are not adopted to mitigate it.

Figure 6 is a conceptual drawing of the temporal components of a cache miss. The leading-edge component is the time it takes the memory system to deliver the first datum of a miss. Since a miss causes the transfer of both the datum that caused the miss and all other data in the same line, the trailing-edge component accounts for those cycles that are lost because an entire line was transferred, i.e., the cycles following the leading edge.

Note that the trailing-edge delay that affects the processor is less than the total number of cycles that it takes to transfer a line, but it is directly proportional to that number, and thus directly proportional to the line size. Roughly speaking, trailing-edge effects fall into three categories:

- Since spatial locality exists, there is frequently an upstream reference (immediately following the miss event) to a datum that is in the same line as the datum that caused the miss. This should not be counted as a second miss, but the second reference will experience a delay if the associated datum is still in transit at the time of the reference event.

- The incoming line consumes bandwidth at the L1 cache and interferes with the running processor.
- There are finitely many systems at the L1 interface (e.g., only one system in many processors) for containing the necessary state for controlling the processing of each miss in progress. That is, there is a maximum number of misses that can be in progress at any time, and that number is typically small. If all of these systems are occupied, a new miss cannot be started until a miss in progress has completed in its entirety (i.e., until an entire trailing edge is over).

Assessment of the leading-edge penalty is more straightforward. When a miss is recognized by the L1 cache, there can be queueing at the address bus in some systems. Once the processor gets control of the address bus, it transmits an address to the L2 cache (or whatever is next in the hierarchy). At the receiving end, there could be an ECC verification cycle, and queueing at the L2. Once the request is accepted, there is a (fixed) L2 access time, perhaps followed by another ECC verification cycle. Again, there can be queueing at the data bus that returns the first datum to the processor, followed by the transfer of that datum (and perhaps another ECC verification cycle at the receiving end). The trailing edge follows this.

Therefore, leading edge comprises several fixed delays that can be estimated directly from the microarchitecture of the cache hierarchy, and a few chances for queueing. Whenever there is a chance for queueing, the average delay incurred is a nonlinear function of the utilization of the subject resource (in this case the address bus, the L2, and the data bus).

The address bus is used for only one bus cycle every miss, so its utilization is small. The L2 utilization can be large, but there are many straightforward techniques for mitigating these effects, e.g., interleaving. The utilization of the data bus is strongly related to the trailing edge. All current system-level design trends tend toward increasing this utilization, hence its associated queueing delay. As is described below, queueing delay explodes if utilization is driven past some general thresholds.

For the sake of illustration, the general trend in queueing effects is analyzed below, using an open queueing model. While this is not accurate because the real system is closed, the solution for the open system is simple, and yields valuable intuition as to the general trend. (The solution for a closed system is very complex, and does not yield to intuition.) The notation used is as follows:

- *Description of bus rate*

  f_P = frequency of the processor in MHz.
  f_B = frequency of the bus in MHz.
  $BR = f_P/f_R$ = bus ratio (number of processor cycles per bus cycle).

- *Description of miss rates*

  mr = miss rate for a single processor (misses per instruction).
  CPI = cycles per instruction for a single processor.
  P = number of processors that share the bus.
  $BMF = (mr \times P)/CPI$ = bus miss frequency (misses per cycle on the bus).

- *Description of trailing edge*

  L = line size (bytes per line).
  W = width of the bus (bytes).
  $P_L = L/W$ = number of "packets" per line.

- *Service time for a miss*

  $TE = P_L \times BR$ = trailing edge (number of processor cycles to transfer a line).

- *Bus utilization*

  $U = BMF \times TE$ = bus utilization (a probability).

Note that the trailing edge is the service time for a miss when the bus is the "server" in the queueing model. (Keep in mind that the trailing edge is distinct from the trailing-edge effect, as was previously described.)

The open aspect of the model is easily seen in the definition of utilization above. In particular, since utilization is a probability, it is physically bounded between zero and one. In the open definition, it is the product of a rate and a service time which are assumed to be constant; hence, there is no feedback and no implicit bounding. In the real closed system, there is feedback to bound the utilization. Specifically, increasing bus utilization increases the queueing effect, which increases CPI, which decreases BMF, which keeps the utilization less than one.

Nevertheless, the queueing delay (Q) is calculated for the open system below. The calculation is expressed in two forms to illustrate a point (as explained later). Q is the average number of cycles that each miss spends waiting to get control of the data bus. This is part of the leading-edge penalty.

$$Q = -\frac{U \times TE}{2} + \sum_{i=1}^{n-1} U^i \times i \times TE$$

$$= -\frac{BMF \times TE^2}{2} + \sum_{i=1}^{n-1} i \times BMF^i \times TE^{i+1}.$$

The first form of the equation is written directly from the following intuition. If a miss requires service, there are i misses in progress with probability U^i, and the new miss must wait for the trailing edges of all i misses to complete. On the average, the first trailing edge is half over when the new miss finds that the bus is busy (hence, the negative term before the summation to correct for this). There can be as many as n misses outstanding in the system, so a new miss can have no more than $n - 1$ misses ahead of it (which is the limit of the summation). Most processors today can only have a single miss outstanding, so in most shared-bus MP systems, $n = P$.

The first expression for Q shows that the average queueing delay is a polynomial in U, and the degree of the polynomial is one less than the number of misses that can be simultaneously outstanding in the system. In most systems, the degree is $P - 1$. The second expression for Q is the same as the first, but it has been rewritten to show that the first term is proportional to the square of the trailing edge, hence, the square of the line size.

If U is kept relatively small, U^i dies out, and the queueing delay is proportional to the square of the line size. If U is allowed to grow large (say, >0.5), U^i does not die out, and the queueing delay explodes.

For example, consider a four-processor system where each processor runs at 2 CPI, and each processor can have (at most) one miss outstanding. Assume that the bus is 32 bytes wide, and that it runs at half the speed of the processor. Let the line size be 128 bytes. If the miss rate is 0.02 (i.e., if the intermiss distance is one miss every 50 instructions), the equations above yield a queueing delay of 3.71 cycles per miss. These system-level assumptions are fairly conservative for most systems that are being projected, and this queueing delay is quite significant.

Figure 7 is a plot of queueing delay as a function of trailing edge for various miss frequencies. The dotted lines show constant utilizations. This shows that if U is kept reasonably small (e.g., <0.3), the queueing delay is fairly flat, but if U is large (e.g., >0.5), the delay becomes exceedingly large.

The example above is shown as point A on the plot, and it demonstrates that modern systems are on the edge of a nonlinear range in bus queueing. A new approach to transferring data is needed in the coming decade. The effect of queueing delay on the base CPI is

$$\Delta(CPI) = \frac{Q \times BMF \times CPI}{P} = \frac{CPI}{P}$$

$$= \frac{CPI}{P}\left(-\frac{U^2}{2} + \sum_{i=1}^{n-1} i \times U^{i+1}\right).$$

In the example above, $\Delta(CPI) = 0.074$, which is nearly 4% of the total system performance. Figure 7 shows that this is headed in a very dramatic direction as utilization is increased further.

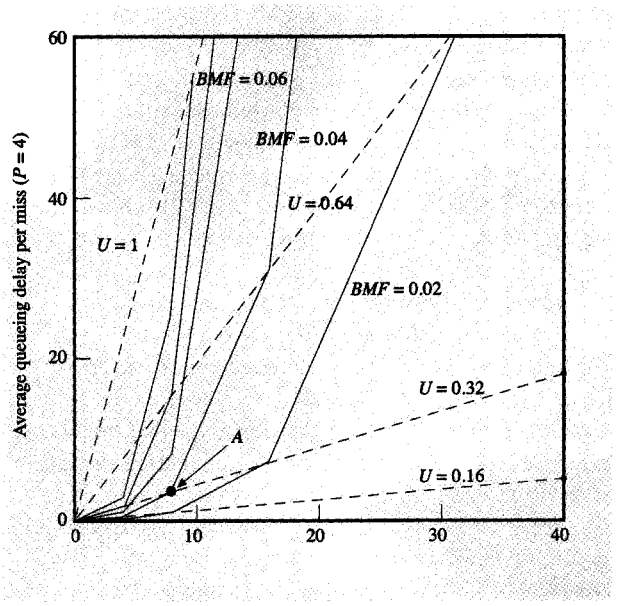

Figure 7

Bus queueing delay as a function of trailing edge for various miss frequencies.

8. Future trends in bus protocols

In the previous section, the various trends in system design that drive bus utilization were discussed, and the effects of utilization on queueing delay were shown. To summarize, the trends that drive bus utilization are the following:

- More processors in a system.
- More processors sharing a bus.
- Faster processors (cycle time) and higher bus ratios.
- Larger line sizes (= larger trailing edge).
- Lower CPI (= higher miss frequency for the same miss rate).
- Higher miss rates (new code with larger working sets).
- Multiple misses outstanding per processor (larger n).
- Speculative execution (= higher miss rate).
- Prefetching (= higher miss rate).

These are all good trends, but they make a previously ignorable effect significant. It is very simple to propose solutions such as making the buses faster and/or wider, adding more buses to the system, or reducing the miss rate. However, these are not real solutions. Much effort is already spent making buses as fast as possible and as wide

as possible. They cannot be made faster and wider than what is possible. Miss rates are inherent, and they are getting bigger.

The problem with a highly utilized bus is that when a demand miss occurs, the exigent datum (the word that the processor needs immediately) is queued behind nonessential and nonurgent traffic that is flowing in the system. Those data fall into three categories:

1. Trailing edges that follow other exigent data.
2. Speculatively fetched data that are not actually needed.
3. Prefetched data that are not needed anytime soon.

Current bus protocols treat the transfers of cache lines as atomic events, and allow exigent data to queue behind nonexigent data.

Since data bus utilization will become large in the next decade, a family of protocols is needed that allows the transfers of lines to be broken up so that exigent data can be transferred on demand. This requires that a small amount of control information be added to a miss request that identifies the urgency and/or priority of the miss. In current protocols, limited control information is already sent that tells the hierarchy the nature of the request (e.g., read, write, fetch exclusive). The new information requires a few more bits depending on how exotic the protocol is.

When future memory systems send data back to a set of processors, they will have to arbitrate each bus cycle. This will allow the trailing edges of multiple misses to be interleaved. Specifically, future protocols must allow the first datum of a demand miss to interrupt the returning traffic (other trailing edges) to get that urgently needed datum to the requesting processor, to then resume transferring the interrupted stream, and to merge the trailing edge of the new stream into the existing stream.

In future systems, misses will not be atomic transactions. Instead, they will be divisible strings of contiguous data. The returning control bus must be active every cycle to handle this; i.e., since a miss will not be returned as an atomic transaction, control data are required to identify the data on the bus for every cycle in which there are data on the bus. (Minimally, it must indicate what processor the data are for, and the miss ID number, as it currently must for each atomic transaction.)

This will be required in shared-bus systems, and will obviate mainstream point-to-point systems because it enables point-to-point bus performance (electrical considerations aside) in a shared-bus environment by minimizing queueing delay even at very high bus utilizations.

Line buffers at both the memory and the processor ends of a bus will be required to time-multiplex the bus among multiple trailing edges. At the memory end, the buffer must hold as many lines as there can be misses outstanding in the system. At the processor end, the buffer must hold as many incoming lines as the processor can have misses outstanding.

In current bus protocols, when a processor issues a miss request, it sends the following three pieces of information to the memory hierarchy:

1. Address (real) of the datum that is to be returned first (and thereby, explicitly, the address of the line that contains the datum).
2. A miss ID, which (on a shared-bus system) comprises two parts:
   - The ID of the processor that is issuing the miss.
   - The ID of the miss (in case the processor can have more than one miss outstanding).
3. The miss type, which tells the memory what status is needed for the line (e.g., shared, exclusive).

When existing memory systems return data, they send back the second field above with the data. In this way, the processors on the bus can tell what the data are (i.e., who they are for, and the miss to which they correspond).

Existing protocols treat each miss as an atomic transaction, and they lock up the bus for the duration of the associated line transfer. For these protocols, the second field (above) need not be sent back on every bus cycle; it is just sent at the beginning of each transaction. In future bus protocols, each bus cycle will be used for a unique transaction, and the second field (above) will be sent on every cycle to identify the data for that cycle.

In future protocols, there could be as many as three additional control fields sent with each miss request. All three together would provide a much richer set of protocols than is actually needed. Nonetheless, all fields are listed below to provide a complete specification for future protocols. Each field is optional, and each can be arbitrarily simple or complex. The new fields are the following:

1. A priority level is used to distinguish as many of the following types of misses as desired:
   - Demand data fetch (exigent).
   - Demand instruction fetch (exigent).
   - Demand data fetch down conditional path (conditionally exigent).
   - Demand instruction fetch down conditional path (conditionally exigent).
   - Data fetch down speculative path (speculatively exigent).
   - Instruction fetch down speculative path (speculatively exigent).

- Data prefetch initiated by prefetch mechanism (prefetch).
- Instruction prefetch initiated by prefetch mechanism (prefetch).

2. Multiplex control specifies how this data stream is to be multiplexed with respect to the other active streams. Assuming that Field 1 above is not used, Field 2 might specify one of the following actions:

   - Put the entire stream behind the currently active traffic. (This is the current default; i.e., this is what current protocols do.)
   - Suspend all other active traffic for one bus cycle to return the first datum from this miss immediately, then resume the suspended traffic, and put the remainder of this stream at the end of the queue.
   - Suspend all other active traffic to return this entire stream, then resume the suspended traffic.
   - Suspend all other active traffic for one bus cycle to return the first datum from this miss immediately, then interleave the packets of this stream with the ongoing traffic.
   - Return the packets from this stream only if there is no other traffic present.
   - Other things deemed useful.

   If Field 1 above is used, Field 2 would specify the same control functions listed above, except that it would enforce them with respect to priority levels, e.g.:

   - Suspend traffic of equal or lower priority for one cycle to return the first packet of this stream, then queue the remainder of this stream behind traffic of higher or equal priority.
   - Suspend all traffic for one cycle to return the first packet of this stream, then queue the remainder of this stream behind traffic of higher or equal priority.

3. A command field rescinds or changes instructions as to how previously issued misses are to be handled. This allows for the changing of characteristics of misses that have not yet begun, or of misses that are in progress. For example, Field 3 could specify the following:

   - Change the type of request of an outstanding miss. This allows the upgrade of a read-only request to an exclusive request, or the downgrade of an exclusive request.
   - Change Field 1 on an outstanding miss. This allows the upgrade of the priority of a miss, which would be appropriate if a speculative prefetch became a known demand miss.
   - Rescind miss request (i.e., cancel a previous request). This would be useful if it were found that a prefetch that had previously been issued was down a path that is now known not taken.
   - Reset starting address. This is useful if a previously initiated prefetch for a line was done without specific knowledge of the address of the datum that would be needed first, and the address of that datum is now known.
   - Add a new starting address. This is useful if an upstream reference to an incoming line is discovered. It is different from the previous command because it does not change the first doubleword address; it just says that instead of returning the line sequentially, the line should be returned starting with the first two explicitly named (nonsequential) words.
   - Other things deemed useful.

This new family of protocols will emerge in the coming decade. It enables high performance on a shared bus, and it allows the bus to be run at very high utilization with minimal queueing delays.

There is currently a resurgence of interest in sectored caches for some of the reasons outlined above. In a future generation of processors, sectoring will re-emerge as an intermediate step in the evolution toward these protocols.

9. Power and microarchitecture for high-frequency design

In the description of the IAS machine, the power budget was shown to be partly responsible for a basic aspect of many modern ISAs known as the von Neumann bottleneck. In fact, power will play an important role in shaping microarchitecture in the coming decade.

A popular viewpoint is that power is an unfortunate constraint which causes a divergence in the microarchitectures that target the server and client spaces. The reasoning is that the client requires low power, and thus a simple core, and the server requires a high degree of ILP, hence high power.

Instead, consider power to be a metric that is useful for guiding microarchitectural development into the highest performance realm for the server. The client and server spaces will converge to the same microarchitectural core with the same physical floorplans, albeit with different circuit designs.

Specifically, there has been confusion in the last decade because there is no general methodology for assessing the impact of a microarchitectural feature on cycle time. Most proposed changes to a microarchitecture target ILP. It is known that adding microarchitectural features to a processor makes it bigger, and that making a processor bigger probably does not improve its cycle time, but the extent to which an added microarchitectural feature hurts cycle time is generally not known.

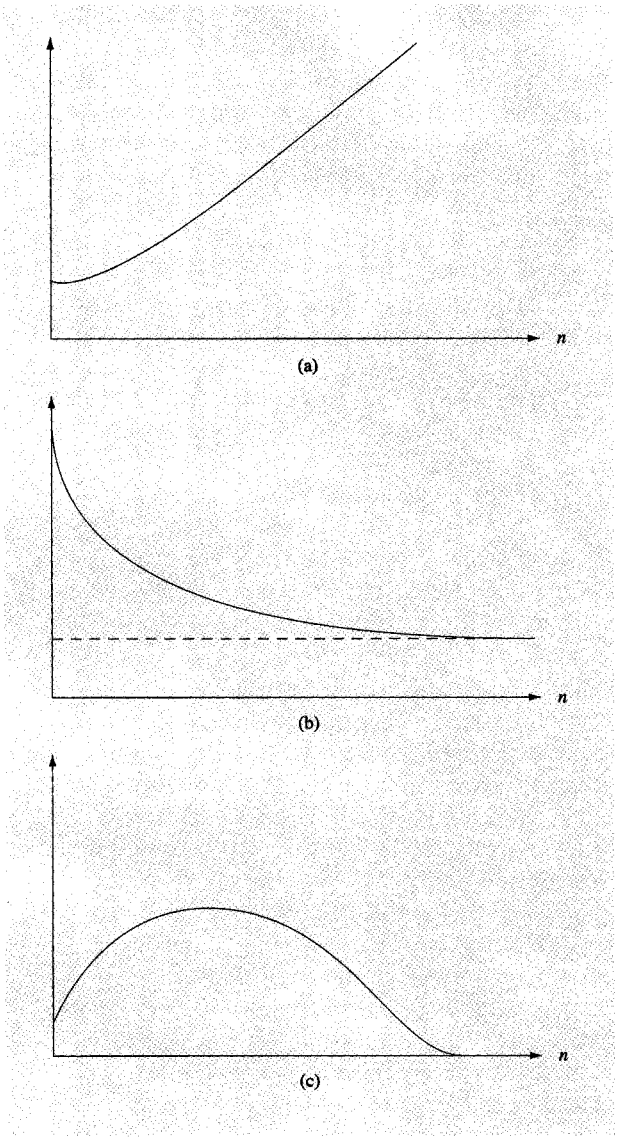

Figure 8

Performance as a function of instruction-level parallelism: (a) cycle time; (b) CPI; (c) MIPS.

It is human nature to give credit to a design where there is a tangible benefit (quantifiable CPI reduction), and to discount the unknown. That is, if the cycle-time impact of an added feature might be negligible, we tend to assume that it is negligible; if the CPI benefit is known, adding a feature to a machine to achieve higher ILP is perceived to be desirable on balance.

This is a dangerous trend, particularly in CMOS, because CMOS is a wiring-driven technology. In CMOS, the area of a machine is (almost) unrelated to the number of gates in the machine, and is more strongly dependent on the number of wires that must be run to connect them. Things that drive ILP tend to drive wiring-track usage in an exponential manner. When parallel execution units, parallel buses, and multiple ports are added to parts of a machine, wiring (hence area) increases dramatically.

While it is true that planarization technology allows for more levels of wiring, more than a few levels are useless for increasing interconnectivity. Very simply, once the lower levels of metal are heavily congested, vias cannot be dropped through them, and the upper levels of metal cannot be connected to the silicon surface. Sai-Halasz advocates using upper levels of metal for "fat wires" which have lower resistance, hence higher speed for signals that must travel more than a few millimeters [8].

Figures 8(a) and **8(b)** respectively show the cycle time and the CPI as a function of the degree of ILP in a superscalar processor. In this case, CPI decreases to an asymptote, and cycle time increases linearly as more parallelism is added to the processor. This is the dual of Figure 4 (the superpipelining case), so MIPS as shown in **Figure 8(c)** has the same form as in Figure 4(c). The lesson is that there is an optimal level of ILP, and the level appears to be small (say 2 to 4, or when pressed for an exact number, π), mostly because of the sensitivity of cycle time to adding wires to the machine.

While the coefficient of the line in Figure 8(b) is very difficult to assess, and is highly debatable, power can be the metric that provides real guidance in achieving high performance. When hardware is added to a machine, the power impact is readily tangible. If the goal of a microarchitecture is low power and, more specifically, low work (the product of power and time as it affects battery life), only those features that pervasively provide low CPI are included. Features that only help CPI sometimes (and that hurt cycle time all of the time) are eliminated if low power is a goal. Those elements should also be eliminated if high performance is a goal.

As the industry pushes processor design into the GHz range and beyond, there will be a resurgence of the RISC approach. While superscalar design is very fashionable, it remains so largely because its impact on cycle time is not well understood. Complex superscalar design stands in the path of the highest performance; he who achieves the highest MHz runs the fastest.

A clear focus on power yields a clear focus on high performance. This trend will make the microarchitectures of the client and the server converge. In the client processor, the circuit design will be optimized for low power. Since CMOS is wiring-driven, adding active silicon area via resizing devices can usually be done with little impact to the physical floorplan. The high-performance server can then be derived directly from the client core by resizing devices to optimize for speed.

• *Complex CPI, relativity, and adiabatics*

In a previous subsection on complex CPI, the discussion was limited to values of CPI in the first quadrant only. Now that the discussion has turned to power, the other quadrants in Figure 5 take on significant physical interest. In particular, points in quadrants II and III have negative real components. The only reasonable interpretation of such points is that they represent the performance of a processor that is running the program backward.

One possible interpretation of quadrants III and IV, which have negative complex components, is that they represent a new paradigm in circuit performance; in particular, they represent processors that run faster than the speed of light. According to simple relativistic theory, when the machine runs faster than light, time moves backward relative to our inertial frame of reference. According to this theory, quadrants I and III are indistinguishable, since quadrant III has the computation being run in reverse while time moves backward. As such, quadrant III is uninteresting.

Quadrants II and IV are of real interest, particularly with the recent advent of adiabatic computing. A processor that can run adiabatically in quadrant II acts as a power source, hence a perpetual motion machine. In quadrant IV, if a machine enters an adiabatic realm, it becomes a black hole. If this happens, it will change the world as we know it.

10. Conclusion

In this paper, several points were made that are antithetical to some of the modern philosophy in processor microarchitecture. These points are based on simple observations relating to the machinations of electronic von Neumann computers, which have been in existence since the onset of this industry.

First, the most popular performance metric, IPC (instructions per cycle), is the reciprocal of the metric that should be used, CPI (cycles per instruction). This is primarily because CPI is a simple dot product of a few numbers that any experienced designer should have at his fingertips. It is intuitive, and it makes for remarkably quick and remarkably accurate estimates.

On the other hand, IPC does not yield to intuition. Instead, it shrouds fundamental issues in mystery, and it has much of the industry (and academia) running down blind corridors in a state of general confusion.

Second, the separability of CPI into three independent components was demonstrated. The three components account for the intrinsic work done by the computer, the pipeline structure of the computer, and the memory hierarchy. It was argued that a solid grasp of each of these three components is necessary in understanding the performance of a superscalar processor, because the scalar components are hard bounds for the analogous superscalar components. Essentially, the argument is that one must have a grasp of the simple case before one can hope to understand the general case.

Third, attention was focused on a trend in future systems in which data bus utilizations cross a threshold that will make queueing at the memory bus a limitation of system performance. A new family of bus protocols that can mitigate this effect was proposed. These protocols will emerge in the coming decade because of the impending delays due to queueing.

Finally, an argument was made that power consumption will drive the development of microarchitecture in the coming decade, and that the aspects of a microarchitecture that result in low power also result in high performance. This is particularly true in CMOS, which is a wiring-driven technology. This trend will cause the client microarchitecture and the server microarchitecture to converge.

References

1. J. P. Hayes, *Computer Architecture and Organization*, McGraw-Hill Book Co., Inc., New York, 1988.
2. J. von Neumann, *Collected Works*, Vol. 5, *Design of Computers, Theory of Automata and Numerical Analysis*, The Macmillan Company, New York, 1963.
3. P. Emma, J. Knight, J. Pomerene, R. Rechtschaffen, and F. Sparacio, "Components of Uniprocessor Performance," *Research Report RC-12203*, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 1986.
4. P. Emma and E. Davidson, "Characterization of Branch and Data Dependencies in Programs for Evaluating Pipeline Performance," *IEEE Trans. Computers* **C-36,** No. 7, 859–875 (July 1987).
5. R. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Res. Develop.* **11,** No. 1, 25–33 (January 1967).
6. J. Liptay, "Computer System with Logic for Writing Instruction Identifying Data into Array Control Lists for Precise Post Branch Recoveries," U.S. Patent 5,134,561, July 1992.
7. J. Smith and A. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," presented at the 12th Annual International Symposium on Computer Architecture, June 1986.
8. G. Sai-Halasz, "Performance Trends in High-End Processors," *Proc. IEEE* **83,** 20–36 (1995).

Received August 8, 1996; accepted for publication February 19, 1997

Philip G. Emma *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (pemma@watson.ibm.com).* Dr. Emma received his B.S., M.S., and Ph.D. degrees in electrical engineering from the University of Illinois at Urbana-Champaign. In 1983, he joined the Advanced Computer Architecture group at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York, where he did research in high-performance computer architecture. In 1992, he became Manager of VLSI Systems and Design, and was a design-team leader on a future IBM CMOS mainframe processor. He is currently the Technical Assistant to the Vice President of Systems Technology and Science. Dr. Emma has achieved 18 Invention Plateaus and has earned the IBM Research Division honorary title of "Master Inventor." He is a Fellow of the IEEE.