Volume Multi-Processor Systems: Part 1

By Chris Rijk – May 2002

Introduction

The primary goal of this article is to look into the volume workstation and server markets to determine what the implications are for the hardware design and usage, for software and the marketplace in general, concentrating on multi-processor systems with 2-8 CPUs. Though various aspects of system design, performance and cost are looked into, this article is not intended to be a buyers' guide. Some basic working knowledge of CPU and system design is assumed, with the target audience being computer systems professionals and computer enthusiasts.

Mostly to help reduce the scope of the article, the main focus is on CPUs and systems designed to be at least reasonably cost effective and for decent volume. This certainly includes AMD, Apple (with PowerPCs) and Intel, who all have reasonably low-cost high volume desktop CPUs that are also used in multi-processor workstations and servers. Intel also has some CPU designs for multi-processor systems only, and AMD is planning to follow. UltraSPARC systems from Sun Microsystems have the highest volume of 64-bit systems and their recent 8-way systems compete well against similar Intel-based systems, and have some interesting lower-end designs in the pipeline.

We'll cover the topic in three separate articles. This is the first part, and we'll be covering single processor performance and scalability and workstation and server sizing, as well as several other CPU and system design aspects like: multiprocessor scalability, the usefulness of 64-bit CPUs, and volume effects in CPU manufacturing. We'll also be discussing server database workloads and the specific system design considerations that go into building servers optimized for such applications.

In the future, we'll discuss some of the current and upcoming multiprocessor system architectures for workstations and servers, including those based around upcoming processors like AMD's Opteron/Sledgehammer and Intel's Itanium 2. For the present, we'll begin with a quick introduction to multiprocessor systems.

About Multi-Processor Systems

Despite the continuing rapid improvement in CPU performance, having a computer with more than one CPU is, and will continue to be, a useful and often necessary way of achieving acceptable performance for many applications. Estimating how much faster a particular program will run on a CPU if the clock rate is doubled isn't simple but mostly depends on how much external communication the CPU needs - the less external memory or I/O requests that are needed, the more linear with clock-rate the performance improvement will be. However, going from a single CPU system to a dual processor system is less predictable and while in theory performance could double, few programs get even 90% faster. In fact, some programs may get a bit slower due to the extra complexities for the OS and system architecture.

Estimating performance when moving from a 2 processor system to a 4 process system tends to be somewhat more predictable and can be extrapolated from the performance increase going from a 1-way to 2-way system. How well an application's performance will improve when adding more CPUs will depend on what shared resources the application uses (main memory and I/O), how quickly those resources run out and how much locking is needed (to make sure the output is reliable). The higher-end RISC and mainframe systems tend to have huge resources in total and also many of the lower-end ones, which is what makes them useful for the higher-end tasks, and also expensive. However, this article is mostly about smaller multi-processor systems - systems that use only a single motherboard typically.

How well the operating system can manage the hardware resources is also important for performance and scalability, though much more so on large systems. A multi-processor capable OS is required in the first place, and while most general-purpose desktop and server OSs are now, older desktop orientated OSs such as Windows 95/98/ME and MacOS 9 make little or no use of extra CPUs.

While mainframe style OSs have been able to make use of multiple CPUs for a long, long time, it's only more recently (since the early 1990s) that multi-processor systems and OSs have become available in general. All the systems discussed in this article are SMP (Symmetric Multi-Processor), where all CPUs are equal, and the hardware and OS combine to give a (comparatively) simple and uniform way of using the system's resources. The first 2-way system from Sun Microsystems came out in 1991, and their first 4-way system came out in 1994 and the other Unix/RISC vendors had a similar time frame. While there were MP capable 486 and Pentium systems from third party vendors, MP systems for Intel really started becoming serious with the launch of the Pentium Pro, together with Windows NT 4.

For the other vendors included in this article, AMD and Apple, MP systems only truly began in 2001, with the AMD 760MP chipset and MacOS X respectively. Though AMD's K6 family was MP capable, no motherboards making use of it were developed. While Apple had dual processor PowerPC systems previously, because the OS itself use the second CPU prior to MacOS X, only specially optimized programs would use the extra CPU at all. Open source OSs like Linux and FreeBSD also have good support for MP systems, and so all current mainstream operating systems have good MP support today and are used this way too, which is a significant improvement on even years ago.

The number of dual processor capable systems is small compared to single processor systems, with around 1 to 1.5 million systems sold per year, or about $5Bn in revenue. While there is a small consumer market, most MP systems are bought by companies - the commercial market. 4-way systems are naturally even more rare, and brand new systems today are too expensive for almost all consumers. The big lure for CPU and system developers is that margins (and profits) are higher, especially for 4-way processor systems and above, and the market for services and support for these systems is also much higher.

The total server market is worth about $40Bn per year, most of which is for general business operations and about 20% is for the HPC market, which includes traditional "supercomputer" systems which concentrate on floating-point performance. The MP workstation market is also important, but easily dwarfed by the server market in revenue. Multi-processor systems also give companies a certain prestige factor, which attracts more demand for lower-end systems as well.

In other words, from both a profit and revenue point of view, it is well worth it to have competitive multi-processor systems.

Single processor performance and scalability

Scalability and "scaling" in general is a phrase loved by marketing but is important to users and system designers. One aspect of scalability is how much performance improves if a particular component is changed for a faster. For example, if a benchmark program (standard or custom) runs 10% faster with a CPU clocked 10% higher, then that CPU has linear scalability with that particular program scales, as clock rate increases. Another aspect of scalability is how performance changes as the amount of data needed for the tasks changes - how performance scales with the amount and complexity of the data. Multi-processor scalability is probably the most important and talked about, and is also related to single-processor scalability.

For some workstation applications, performance will be dominated by the achieved main memory bandwidth - with 1-2GHz CPUs, and over 1GByte/s of sustained read/write bandwidth, over 10 simple calculations could be performed per byte of data and performance would still mostly dependent on bandwidth. However, many workstation applications have a good enough cache hit rate that algorithm complexity and raw CPU speed is still important to performance. Because of this special instructions to improve performance (like SIMD operations) can be very useful, as are faster clock rates. This is overall however, and specific applications and benchmarks may be memory bandwidth dominated (in which case increasing CPU speed won't increase performance much) or raw CPU speed dominated (in which case increase memory speed won't increase performance much).

On another level, parts of applications like video editing may be dependent on storage performance - when doing processing operations on files over a gigabyte in size for example. For applications that make heavy use of 3D graphics, special graphics cards with dedicated performance features are often used, and may become the main bottleneck, depending on the application. Server applications may become dependent on storage, or network bandwidth as well.

For applications with little I/O, different projects will have different levels of complexity and detail resulting in different data sizes, and simpler ones may get good benefit from a CPU cache. Larger, more complex projects may be dependent on main memory performance though, if cache hit rates become low. The more cache a CPU has, the bigger project it can manage without becoming dependent on main memory performance. Beyond this, for truly complex workstation tasks the more main memory a system can use, the better, else performance becomes stalled on storage I/O.

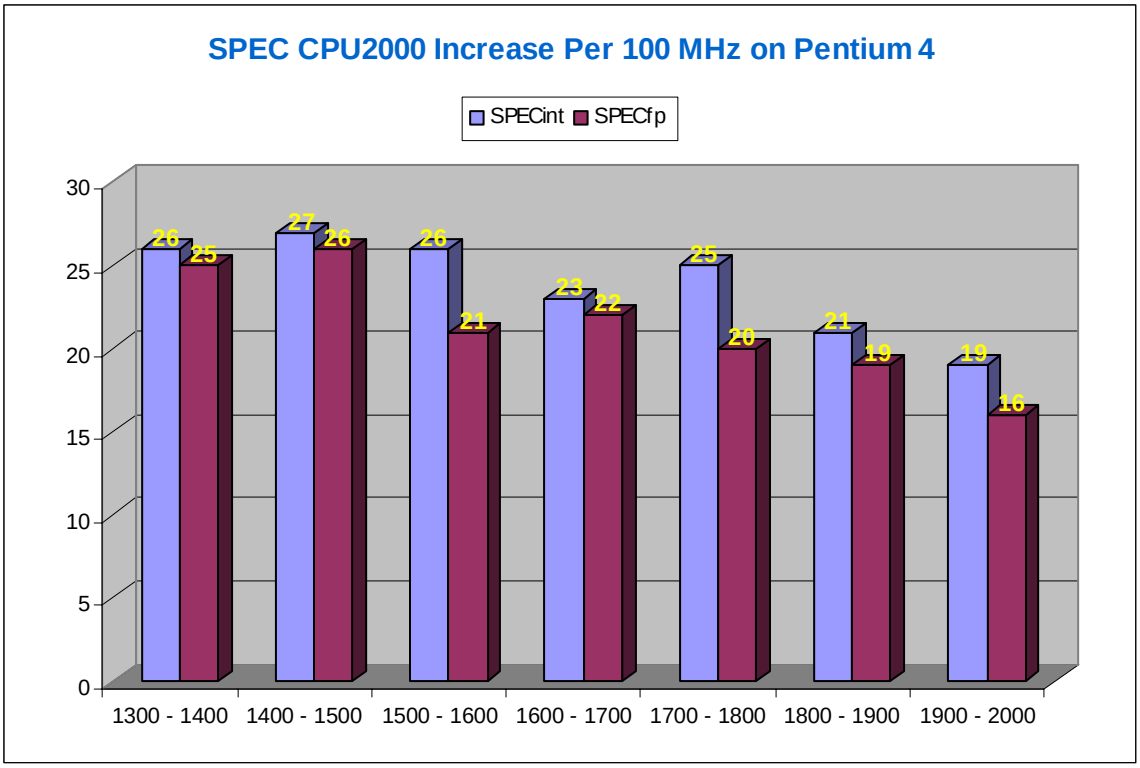
For SQL database performance, in general, performance would be a mix of core CPU integer performance, the cache and memory system and the I/O system (more on this later). A SQL database is way to manage and use data in general, and performance characteristics are highly dependent on the particular setup of the data - each setup tends to be somewhat unique. So for a database with very little data or only a little being used, cache hit rates would likely be nearly 100%. For a CPU where all the cache increases in clock rate linearly with the CPU clock rate, any program with 100% cache hit rate would get linearly performance scalability with the CPU clock rate.

Given that a single cache miss can cause processing to stall for over 100 clock cycles, if cache hit rate falls from 100% to 99%, overall performance can drop have half, or more. The more performance is dependent on the main memory system, the less effect increasing the CPU clock rate will have, so cache hit rate is very much tied to scalability as well as performance. This is the case for all applications, but it's particularly complex to predict cache hit rates for databases since most are custom to some extent.

Other types of server workloads tend to be roughly similar to databases, being either I/O dependent, memory latency dependent, or cache latency and CPU core performance dependent, based on how well the program and data caches. Some server applications like file serving, network traffic routing and firewalls tend to be always or nearly always I/O dependent.

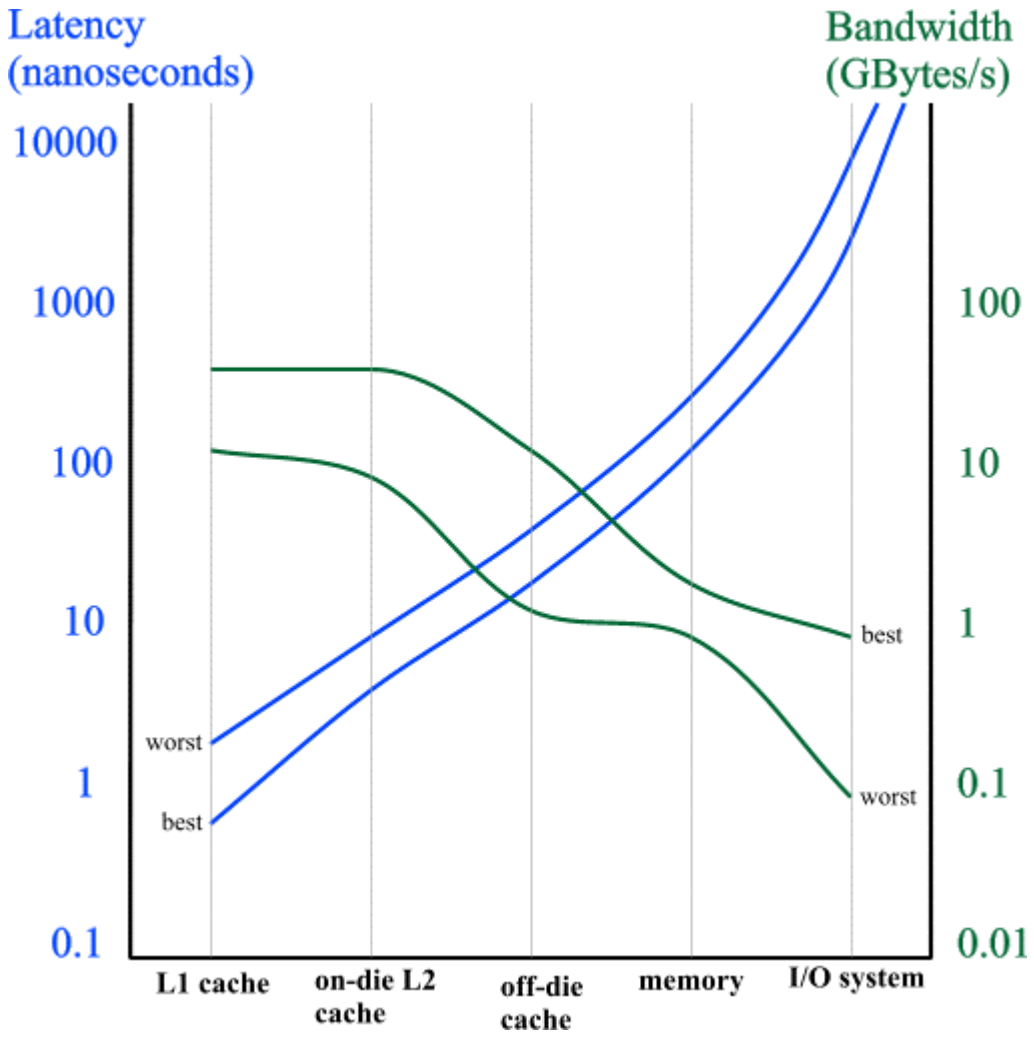
This is a generalization though - for example, fulfilling a single SQL database query requires parsing and checking the query, optimizing it, solving data locking issues, possibly reading or writing significant amounts of data, and preparing any results for output. Each of the sub-operations would likely have different performance characteristics, with some more dependent on the CPU, some dependent on the cache system and some on the main memory or I/O systems. So improving the memory system for example may affect some of these sub-tasks significantly, but others not at all. Different tasks on the same system (different queries to a SQL database for example) also have different profiles for each of the sub-tasks. Server applications such as SQL databases tend to be the most complex in terms of different combinations of performance for a single application.

Single processor clock-speed scalability and memory system



The above two tables show the performance increase for SPECint and SPECfp (peak values) for the Pentium 4, for each 100MHz increase from 1300MHz to 2000MHz. The benchmarking was done mostly at the same time, and though the general slope is downwards (scalability getting worse), the slight variations in the measurements is noticeable. It's interesting that for the SPECint and SPECfp results, scalability seems to get worse at about the same rate. SPECfp is much more bandwidth dominated than SPECint, but SPECint would be affected more by latency, probably. Either way, the reason scalability gets slowly worse is because the memory system performance is fixed, and the CPU spends an increasing percentage of the time waiting for main memory requests.

Comparing bandwidth and latency



The above graph is a rough outline of what kind of latency and bandwidth range modern CPUs see to various other components in a single processor system. Both on-die level 2 cache and off-die external cache (which may be level 2 or level 3) have been shown though not all CPUs will have both of these. It's interesting to note that the difference between worst and best latency (in terms of orders of magnitude) is a fair bit larger than for bandwidth.

Memory latency tends to improve quite slowly, by about 10% per year, while memory bandwidth improves much faster, perhaps around 30-40% per year, though this is still slower than how quickly CPU clock rates increase. Cache speed and size is helping compensate for this though. 15 years ago almost no CPUs had on-chip caches (or caches at all) and now it is becoming common to have 2 or 3 cache levels. Cache sizes will continue to rise to help offset the memory latency gap.

Main memory latency is linked to main memory bandwidth in subtle ways - for CPUs, it's mostly more efficient to cache a small block called a "cache line" (32 or 64 bytes typically) rather than just the 1, 2, 4, 8 or 16 bytes needed for a single memory request. But that means when any memory needs to be loaded from memory, 32 or 64 bytes needs to be loaded into the cache even if only 4 bytes will be immediately used. The specific data the CPU core is waiting for when loading a cache line is called the "critical word". With SDRAM and DDR SDRAM the critical word is fetched first when loading a cache line, while with Rambus the order is fixed, though it does have features to help improve bandwidth efficiency instead. Until a cache line is filled, another load to main memory can't start to pull transfer data to the CPU, so being able to load a whole cache line fast is useful for some aspects of latency. The time taken from loading the

first word of a cache line to loading the last is entirely dependent on bandwidth - double bandwidth and the time taken will halve.

For a given level of technology, increasing main memory bandwidth or reducing latency is hard without increasing the cost of the system significantly. The simplest way is to simply use faster memory chips, but memory latency improves quite slowly. Using a dual channel memory connection would double bandwidth over a single connection, but requires a faster connection to the CPU, more complicated and expensive motherboards and requiring at least two memory modules, which would be an unwelcome cost for entry level systems. For servers and workstations, the costs may well be worth it, but the same CPUs and chipsets will often be used for lower-end (desktop) systems. An alternative that is starting to be used more often is to put the main memory controller on the CPU itself (instead of on a supporting chip), which can reduce latency by about 20-40% (depending on implementation), and though it makes the CPU more expensive, the motherboards become cheaper and simpler.

A final aspect of scalability for single processor systems is how much physical and logical memory (via virtual memory) it can handle. In the case of some programs, even on single processor systems, more memory will mean better performance. This is either because a lot of caching is used or main memory is much faster than disc I/O (typical for server applications), or because very large amounts of data are being processed (typical of workstation applications). Most single processor systems today support 2-4GB of main memory, and this is almost always enough for single processor systems. However, a tiny amount of users would definitely use more if it was available, and this proportion is growing rapidly.

Sizing and Using Workstations

One way to describe a workstation would be a tool to help project development - begin a new project, import data and begin setup, make many small changes to get towards project completion, perform various testing and previewing, and outputting the final result. All these in response to a single user: the developer. Different parts of the process stress the system in different ways - setup may be simple, or (for video editing) may stress storage I/O, editing tends to be fairly light-weight in many cases but a fast response time is very useful, which may be particularly hard to achieve with large or complex 3D projects. Outputting the final result will often be quite intensive on the CPU, cache, memory or I/O system, and often over a long period of time too. Testing and previewing would likely be somewhere between final output and editing in terms of how the system is stressed.

For example, with video editing, setting up the project would probably start with some video capture (which could take several hours) and would stress the storage I/O mostly (disk write performance). Editing would stress the I/O a bit (depending on in-memory caching of the video), the CPU and also main memory bandwidth. Previewing and testing would be similar to outputting the final project, which would likely be CPU, main memory bandwidth and storage I/O intensive. Which particular parts are stressed most would depend on the workstation software being used, the project and the system hardware.

Something like software development (which is classified as a workstation environment) would be quite lightweight all around - little or nothing to do for setup, editing would be a bit like running an "office" package - like a large desktop application with a fair bit of data. Testing would depend on compile speed and where the program being developed is executed (could be on the workstation or could be a remote server), and there's little to do for final output.

For 3D animation work, there'd be little to setup, editing would most likely tax the CPU, cache, main memory parts of the system, the 3D acceleration hardware and for very complex projects, the storage system. Testing, previewing and final output would depend on the environment - animation would likely be previewed on the workstation, while the final output might be performed on a remote "render farm" - compute server.

From the point of view of a company planning to buy new workstations when they already have some for existing projects, estimating the hardware requirements would be simple enough in most cases that this can be done in-house - without needing outside consultants for example. A simple way to estimate performance would be to try to get some demo systems from various makers and benchmark them in-house - the more valuable the potential order the easier this becomes. Another way would be to look relevant benchmarks for their current workstations, and look at how

current systems on offer compare in those benchmarks. However, pre-packaged benchmarks don't always show up the performance for user editing operations for example, which is important as it affects productivity. There's also a problem with finding relevant benchmarks sometimes.

Once a particular workstation has been decided upon, there are setup and optimization issues but these tend to be comparatively simple - the main issue would be making sure the latest (stable) software versions and drivers are being used for critical components. For workstations that require significant storage requirements there can be some somewhat complicated setup and configuration issues, to optimize the storage I/O performance. There may well be ways to improve performance by choosing different project settings but these are often about trading off performance with quality (or quantity).

Sizing and Using Servers

Though there are many classes of server setups, the big market is for SQL databases - many servers run just one SQL database program, with separate servers actually making use of it. For databases it's far more complicated to size servers entirely in advance, though it's just about possible to make some guidelines. The large database and systems companies like Oracle and Sun among others offer server sizing services and centers for free, to help customers determine their hardware requirements for their application and performance demands. The particular reason why such things are needed is that a database is useless without data, and the structure, amount and usage of the data is somewhat unique to every setup - there's only one database that is setup and used like the one running on Ace's Hardware for example.

A blank database is a bit like a computer with a basic OS and compiler, and nothing else - the rest is up to whoever owns the system, which is why it's hard to predict performance in advance. Databases are also very complicated to setup (Oracle has more options than most operating systems and it has most of the functions of one too) and provide rich pickings for consultants. For SQL database systems, there is little in the way of "pre-packaged solutions", and a huge consultancy industry available to help. There are some large off-the-shelf applications that use SQL databases though (like those from SAP), but each has different characteristics.

There are many more types of servers than just SQL databases (and more databases than just SQL) - for example, web-servers, network file servers, network traffic routers, firewalls, e-mail servers, directory services, cache servers, and various types that involve user management, authentication and administration. All these servers tend to use a lot of network I/O when being pushed hard, and maybe use a lot of file I/O as well. However, the more I/O intensive server types (file servers, routers, firewalls, caches) rarely need that much CPU power and are often single CPU systems, which also helps reduce costs.

Server types like web-servers, e-mail servers, and those involving user services and management also often make heavy use of SQL databases. There's also a whole area of applications to help run businesses (enterprise applications) - purchase order management, supply chain management, resource management, payroll and so on, which make very heavy use of SQL databases. A final server category is compute servers, which mostly just need high floating-point (and sometimes integer performance) with good available memory bandwidth, and sometimes a large amount of fast storage.

Compared to workstations, SQL servers have several extra complexities to deal with when determining performance. It can often be hard to estimate how much demand for the database there will be and how this will change over time. Consider a database for a web site - it can be hard to predict the most popular parts of the web site (each of which would stress the database in different ways), how demand will increase over time, and amount of data will generally grow. New features will also often get added over time, adding to the complexity.

Except for already existing applications, until the database and the programs that use it have been setup, it's hard to estimate any kind of performance - the programs and setup will often change over time too. It can be hard to estimate the size of the data, and how this will change over time - this partly depends on how much demand there is too. Setting up and optimizing the database configuration (computer system, storage, OS and the database software itself and applications in particular) is often quite complicated and difficult.

Sizing Servers, Continued

In real world cases, databases and the applications that use them are often setup and used quite inefficiently (even for simple things) - I have certainly come across many badly setup databases and database applications over the years. Additionally, there is also a problem of reliability, as many databases servers would ideally be able to be providing a service 100% of the time. The time taken to shut down a system, upgrade the hardware and do a restart, may only be 5 minutes, but it's not uncommon that is too much downtime. For projects that need high levels of uptime, this makes it all the more important to estimate how much performance will be needed before the system goes into production, because if the hardware was underestimated, the system will have to be upgraded after going live.

Despite all the performance problems, many database operations (executing a SQL statement) are actually quite simple and often take a few fractions of a second to execute - the hard bit is often making sure many can be run quickly at once. Taking Ace's Hardware as an example, loading up a page for an article is very simple and easy to optimize and only outputs 20-50KBytes of data. Loading user preferences, for pages that have such options, is about as simple and requires much less data. These operations all involve just loading one easily identifiable row of data from the database.

Displaying the threaded listing of posts to the forums from the last few days is quite complicated however, and outputs about 200KBytes worth (depending on recent usage), after running through hundreds of rows in the database and doing some sorting as well. This operation used to take 5 seconds or more on the old SPARC Station 20, which is actually pretty good for the hardware - coding an efficient (or even simple to write) fully threaded message display is actually a lot harder than it seems, which is perhaps why they aren't used so often. Most pages on the site simply depend on a single database query (or a few simple ones), but the front page (as is very common with web-sites) is a big exception - it requires database operations on just about every aspect of the site. However, all the in-memory caching by the web application drastically reduces the number of database operations required for all parts of the site, and the database rarely takes up more than 5% of the CPU.

The amount of "active data" (commonly used data) for Ace's Hardware is actually quite small and would probably fit quite nicely in 8MBytes of level 2 cache, which is quite common for the multi-processor 64-bit RISC servers out there. However, Ace's Hardware is running quite fine on a small, low-end single processor with just 0.25MBytes of cache while this article is mostly concerned with 2-way and 4-way capable systems. To really need a modern multi-processor system for an application similar to Ace's Hardware, many active discussion forums would be needed and many more articles, or a much less efficient implementation. However, that much active data would definitely not fit inside even 8MB of cache, and instead the memory system would start to be stressed quite a bit, and quite possibly the storage I/O system.

Different types of web sites will have different patterns of database usage - news orientated sites, auction sites, on-line shops, share dealing sites and so on. There's often also less visible aspects of database usage - for example, putting the web access logs into a database for data-mining, which could be so intensive that the processing for this would have to be done on a separate database so as not to interfere with any publicly running operations. Typical operations for more e-commerce sites include getting daily reports on customer buying patterns as well as simply revenue, most popular (and unpopular) products and services, shipping information etc.

Plain "bricks and mortar" companies would have similar daily, weekly, monthly, quarterly and yearly report demands. Companies like Wal-Mart have small database applications in each of their stores, which keep a track of stock in the store and can also be used for making purchase orders to suppliers, and keeping a track of customer buying patterns. These smaller databases would likely feed back to a large centralized data-center on a regular basis. In Wal-Mart's case, they also make customer-buying information for their products available to the suppliers of those products in real-time, which allows the suppliers to more efficiently manage their supply chain and delivery times, as well as production, if they're a manufacturer.

There's also banking and credit-card systems, which require huge central databases (or many decentralized ones) which can handle a large number of transactions. Dealing with frequent writes to a database can be quite hard to handle - I have seen older versions of MySQL, which is more designed around fast reads, slow down significantly when writes

become common, but in many cases slower databases like Oracle run faster in such situations. This may have changed somewhat with recent versions however - it's mostly a matter of optimization and scalability of the software design.

General-purpose databases (including all SQL ones that I am aware of) are much less efficient than what a custom database (built and optimized for a specific application) could be. This in turn means that the hardware costs would be higher for a general-purpose database. However, general purpose databases are almost always a better choice - the cost of developing, testing and maintaining a custom database will almost always far outweigh the higher hardware cost, and programmers who are good enough to write a reliable, fast custom database are also very rare. Unless a project is likely to live or die by the database performance, for businesses, it's better to go with a general-purpose database. For more details on how a custom database approach works, please see:

SPECmine – A Case Study on Optimization
http://www.aceshardware.com/read.jsp?id=45000251

In the end, there are two main reasons for buying a new database server - one is for a new database system, and the other is to upgrade an existing one. For existing database systems, it's much easier to do benchmarking, because the software will already have been written, and there is plenty of usage information. In comparison, new database systems have neither. One way help get around this is to roughly estimate the data size needed and look at benchmarks for similarly large systems and discard any poorly performing systems. Another way would be to buy a development and testing server early on, and when the project is ready to go live, to buy a production server, based on benchmarking of the current development environment.

Single CPU system cache coherency - DMA (Direct Memory Access)

Most modern computer systems allow the I/O system to issue read requests to main memory and to write directly to main memory without going via the CPU. Without DMA, when the I/O system wants to perform a read or a write it will trigger an "interrupt" signal, which will cause the CPU to suspend what it's doing, switch to some code that services that interrupt, perform any necessary reads or writes between the main memory and I/O systems, and then return to the previously running task. With DMA, the CPU is not interrupted at all, so this increases efficiency of both CPU tasks and I/O tasks.

DMA introduces two problems however - if the I/O system uses DMA to write to the memory system, and changes some data that the CPU has cached, and then unless the CPU re-fetches the cache line from memory, it will not get the new data. In practice, this is solved by some additional logic in the memory controller that informs the CPU of a DMA write, and the CPU invalidates (flushes) the relevant cache line. Alternatively, the I/O system could copy the data for updates to the CPU's bus, so that the CPU can pick up changes immediately. The second method has lower latency, but uses bandwidth inefficiently.

The second problem is DMA reads - the cache may contain data more recent than main memory, and in such cases the DMA system needs to read the cache instead of main memory. In practice, the DMA system will pass all read requests to the CPU as well, and the CPU will then return the cached line if it has it.

Multi-processor systems have similar problems to deal with except that they have multiple caches as well as multiple readers and writers.

Keeping Data In Line With Multiple CPUs and Caches

In a system with two CPUs where neither has caches, if both CPUs are simultaneously running a program that loads the same location in memory, adds 1 to it, and stores it back 1 million times, then the final result is not predictable. If the times when the CPUs do this do not overlap at all, then the number in memory will be predictable - 2 million higher. If there is an overlap, then quite often, just after CPU-1 loads from the location, CPU-2 will write to that location, meaning the data CPU-1 just loaded is out of date, so any calculations on this are invalid. However, it's not up to the system to correct this sort of problem, so in practice the data will not be processed correctly.

To enable more predictable calculations in such situations, the OS provides an API to programmers to lock access to parts of memory, and the OS uses special instructions in CPUs that enable a reliable (atomic) lock operation. So, programs will then lock access to the piece of data (waiting until it's free if the lock is being used currently), perform the calculations and then release the lock. It's up to the programmer to use locks correctly.

The problem becomes rather more complicated when the CPUs have caches. Since the lock itself is stored in main memory, lock functions may ignore the cache, which isn't too hard to implement since special instructions are used for locking. However, the real problem is the actual data processed during the lock - the CPU can't easily track this so it can't be treated specially. Actually, the problem is the same for any data, whether accessed by locks or not - for example, if a particular program starts running on CPU-1 but then the OS changes it to run on CPU-2, any data written to CPU-1's cache but not yet written to main memory will never be in CPU-2's cache.

For a dual processor system, if CPU-1 begins a load, but the data isn't in it's cache, it naturally needs to do a request to main memory. However, if CPU-2 recently wrote to the same memory address in it's cache, but hasn't yet written this back to main memory, then the data in main memory is old. So while doing a request to main memory, CPU-1 has to also check to see if any of the other CPUs in the system have a more up to date version.

The most common method is to do a cache snoop to all the CPUs in the system - a broadcast method. An alternative is to keep a central directory tracking in which CPUs have cached which memory addresses, so a cache check only needs to deal with any CPUs that may have updated a particular address. CPUs keep a track of whether they have updated cached data, but not yet saved that back to main memory, so it's fairly easy for them to determine if they have more up to date data.

In MP systems, a CPU needs to do a cache coherency check whenever it has a cache miss. A simple method would be to send a cache snoop request, wait to see if there are any replies, and then perform a regular memory load if there were no snoop hits. Rather than all CPUs replying with a "snoop miss" signal, which could take up a lot of address bandwidth, most algorithms use a timeout based on the worst case latency, which could add a fair bit of latency to all memory requests. An alternative algorithm would be to run the cache snoop and memory load at the same time, ignoring the memory load if a cache snoop hit occurs.

However, consider a situation where two CPUs are both often writing to the same address in memory - without a coherency protocol, both CPUs would cache the address in their private caches, and update only that. With both CPUs operating independently, the final result will be unpredictable. There are many ways of solving this problem, though the general idea is when one CPU starts repeatedly writing to the same address, other CPUs will be prevented from caching that same address. For the other CPUs, loads become a lot slower for that address, but fortunately it's fairly rare for more than one CPU to be regularly writing to the same block of memory.

There are many ways of implementing cache coherency protocols - it's a heavily researched area of system design. However, specifics on the various protocols is a bit beyond the scope of this article, though see this page on MESI and MOESI.

Multi-Processor Scalability

How well programs (or benchmarks) in general will make use of extra CPUs depends on the program itself a lot - some programs can be trivial to scale across many CPUs and separate systems ("embarrassingly parallel" they're sometimes called), some can be hard and some can be nearly impossible. Workstation tasks that take a while to run tend to revolve around processing a large group of data - making this run in parallel is generally done by making each CPU process a portion of the data, grouping all the results together at the end. Server tasks generally involve responding to separate requests, and making this run in parallel is generally done by handing each request to whichever CPU is free first.

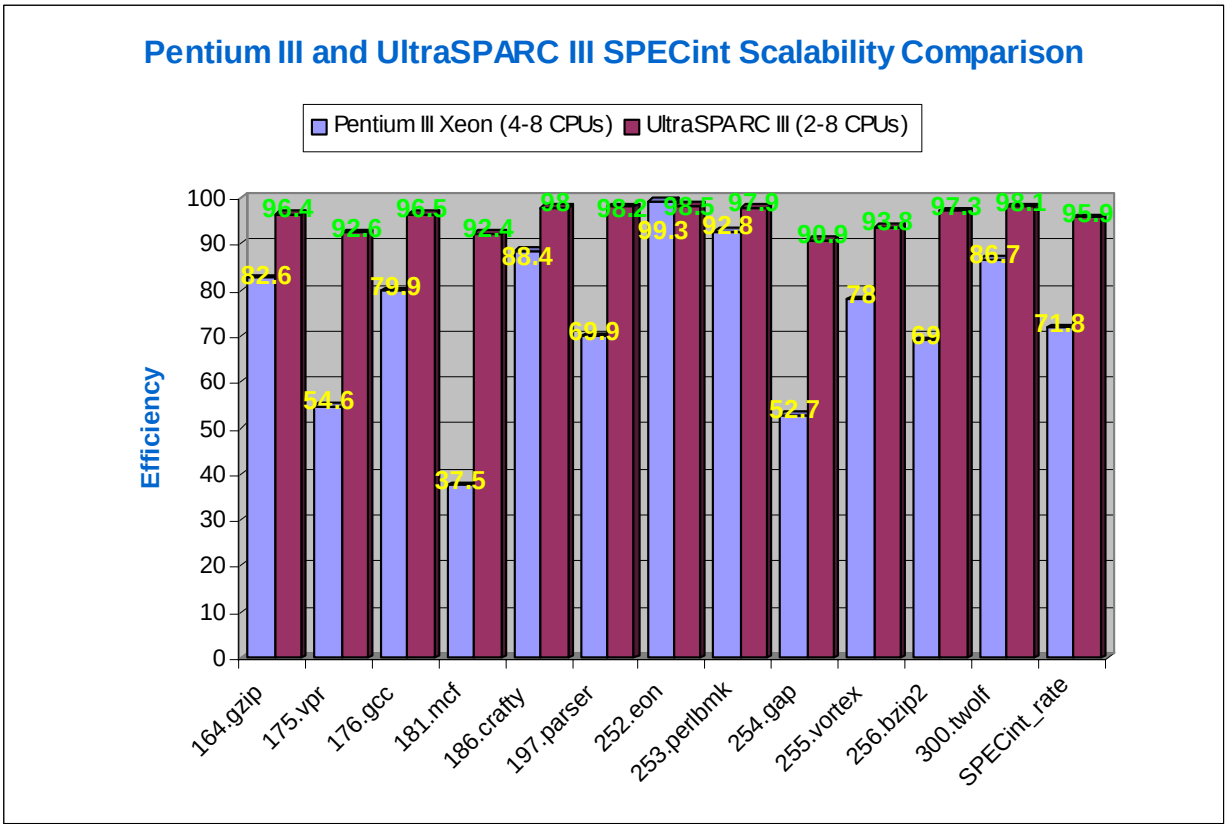
Separate operations (threads and processes) on different CPUs on the same system will compete for shared resources, such as total memory, memory bandwidth, CPU time and I/O. On a 4-way system, if all CPUs are active the most memory bandwidth each CPU can sustain is only 25% of the total system bandwidth. The more resources a system has the greater the maximum performance can be.

When adding more CPUs, whether a system is considered to have good scalability or not is rather subjective. For example however, if the number of CPUs in a system is increased by 100% (doubled) and performance (for a given benchmark) increases by 90%, most people would consider that system (and application) to be "scalable". In other words, performance per CPU is relatively constant. However, a "scalable" system design doesn't necessarily have higher maximum performance than one that isn't "scalable". Having a scalable system is nice as it makes predicting performance much easier, and just about everyone touts his or her systems or software as being "scalable."

If a particular CPU and system design scales well to 2-way, then a similar system design with the same CPUs will probably have good performance with a 4-way design - if it scales well at 2-way it's probably got a decent amount of resources (e.g. memory bandwidth) left, so a 4-way design will be able to make use of those spare resources. The 4-way system could also have extra resources, though this would obviously increase costs too. A more scalable system would also be more upgradeable - faster CPUs, which will become available with time, will use more resources (a 10% increase in the speed of all parts of the CPU will mean cache misses 10% quicker, so main memory demands increase), to the more spare resources a system has the better use it will make of faster CPUs. It's quite common for servers to be used for over 5 years, so the more they can be upgraded the better, even though many do not in fact get upgraded.

To see the difference between a system with a lot of resources per CPU (cache and main memory bandwidth) and one with not so much, consider SPECint results for the systems below - scaling from 4-way Pentium 3 Xeon system to 8-way system, and scaling from a 2-way UltraSPARC-III system to an 8-way system. Unfortunately, at time of writing, no 4-way UltraSPARC-III results are available, and only 750MHz results are available for 8-way systems so far. In each case, the CPU is the same - 700MHz Pentium 3 Xeons with 2MBytes on-die level 2 cache, and a 750MHz UltraSPARC-IIIs with 8MBytes off-die level 2 cache.

For the graph below, the "efficiency" is shown - an efficiency rating of 100 means that performance increased perfectly linearly, a value of 0 means performance dropped to zero, and a value of 50 means that performance was unchanged.



In absolute terms, the 8-way Pentium 3 Xeon systems are only 44% faster than the 4-way ones, which means that with the 4 extra CPUs, the system only gets 1.76 CPUs worth of extra performance -- rather poor value for money. This level of scalability is not that surprising since each group of 4 CPUs share 0.8GByte/s of memory bandwidth. As a side note, it seems likely though that 252.eon fits almost perfectly into the 2MByte cache the Pentium 3 Xeons have as it gets nearly linear scalability - the higher the cache hit rate, the less main memory is needed, which leaves more for the other CPUs.

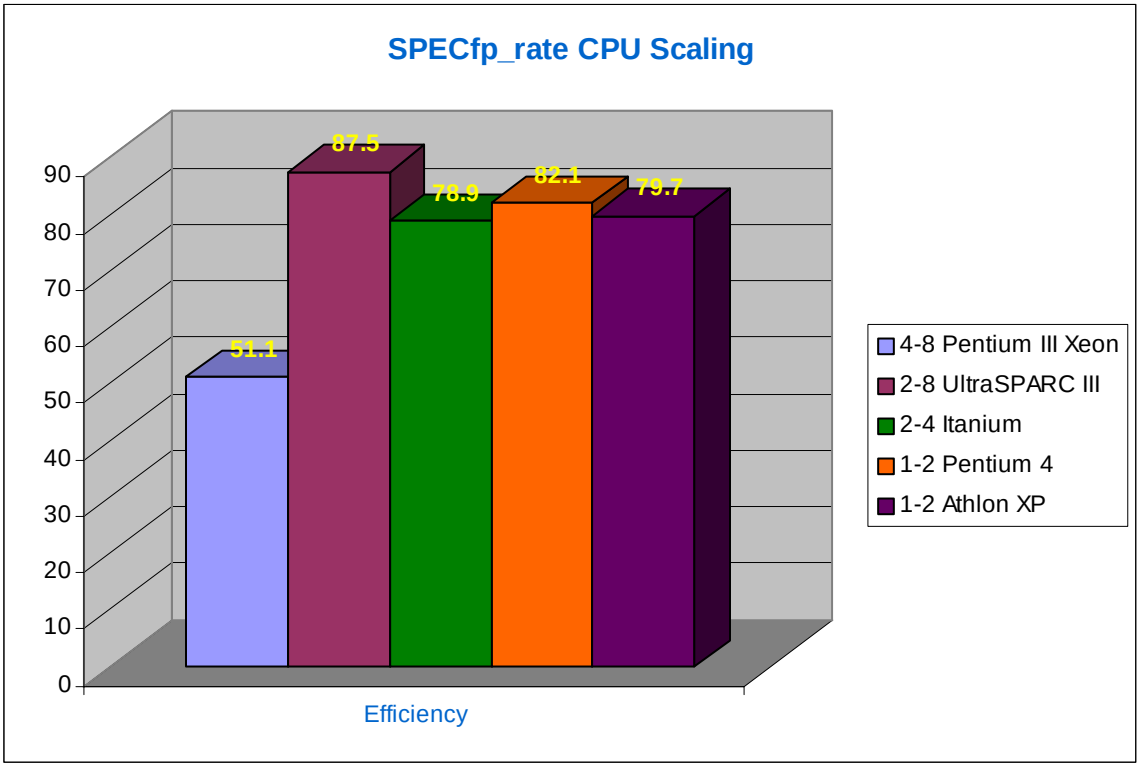
Even worse, in some tests, the 8-way system actually does worse than the 4-way system, and this could possibly be due to differences in the chipsets or because the extra contention itself on the shared Pentium system bus causes efficiency to drop. It's unlikely that the compilers/OS would have made much difference as for each CPU type the tests were done at similar times with the same compilers.

The UltraSPARC-III systems scale pretty effortlessly in this benchmark suite, aided by a combination of larger caches (though for SPECint there's not much difference from 2MB to 8MB I think) and a design that increases memory bandwidth (nearly) linearly with CPU count - 2.4GByte/s per CPU. With 6 extra CPUs, the system gets 5.75 CPUs worth of extra performance.

To some extent, this comparison is rather unfair to the Pentium 3 Xeon as it's much older than the UltraSPARC-III, CPU performance having outgrown a architecture originally used for 200MHz CPUs. On the other hand, it's harder to scale linearly from 2 CPUs to 8 CPUs - for scaling from 4-way UltraSPARC-III results to 8-way, the inefficiency rating could halve compared from 2-way to 8-way. In absolute performance, the 750MHz UltraSPARC-IIIs are about 40% faster than the 700MHz Pentium 3 Xeons.

Scaling with SPECfp_rate

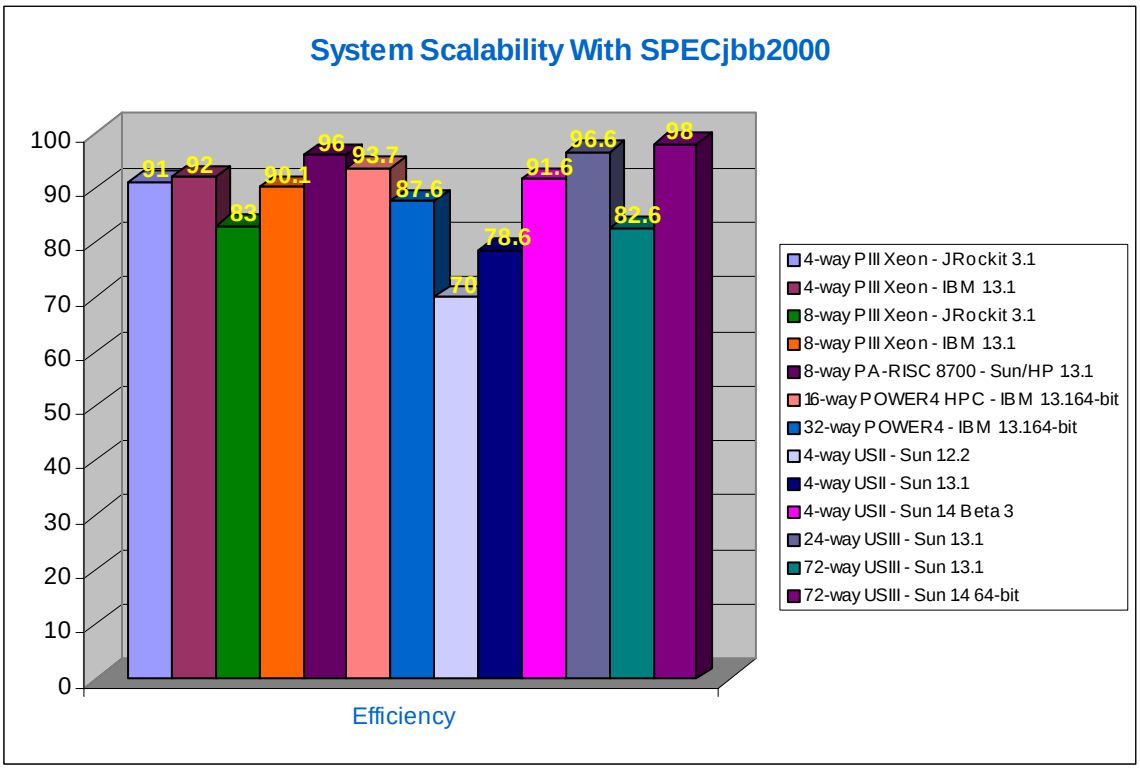
There are more complete results available for SPECfp_rate, and below are some overall scalability inefficiency ratings for a number of systems.



Overall, the UltraSPARC-III comes top again in scalability, despite having to scale from 2 to 8-way. The Pentium 3 Xeon does terribly on SPECfp_rate - the 8-way result is only slightly ahead of the 4-way result. It should be remembered however, that the sub-test results for both SPECint and SPECfp vary significantly; so predicting scalability in advance (without having profiled the benchmark on the systems) is pretty much a shot in the dark apart from some general conclusions.

Server Benchmark Scalability - SPECjbb2000

SPECjbb2000, a server-side Java benchmark somewhat based on TPC-C that IBM developed, is a useful benchmark for scalability. This is because with a single benchmark run, it starts with a test with just one active "warehouse" (thread) and then increases the number of concurrent warehouses (threads) until overall throughput stops increasing, then takes a set of measurements, using this for the final benchmark score. Here, the scalability result is taken from the performance gained when doubling the number of active threads (i.e. active CPUs) to the maximum CPUs for the system.



The results are interesting in a number of ways - the results on the Pentium 3 Xeon systems are quite close despite using completely different JVMs, which suggests that the hardware is the limiting factor, or maybe the operating system. The 8-way PA-RISC results are useful to compare, as the PA-RISC 8700 has a total of 2.25MBytes of cache, only slightly more than the Pentium 3 Xeon. The PA-RISC system, which seems to have a port of Sun's HotSpot JVM, scales much better to 8-way, which is probably as a result of having more system bandwidth, though OS optimizations may help a bit too. It will be interesting to compare with 8-way Pentium 4 Xeon scores, when they become available, as they have much more system bandwidth than the Pentium 3 Xeon systems.

The IBM POWER4 results are particularly interesting - though both results used 16 POWER4 chips, with the 16-way "HPC" result, only 1 of the CPU cores in each POWER4 was active while with the 32-way result, both CPU cores in each POWER4 were active. With just one thread active on a dual-core POWER4, it would have all the cache to itself, while with 2 threads active on the same core, they would be fighting over the shared cache, which is probably why the scalability isn't as linear.

The results with the UltraSPARC-II systems were run by a 3rd party company called Gcocco, which is perhaps why the results were not done with the latest (and more scalable) memory management options and thread libraries that Solaris 8 provides. Anyway, with the results being performed on the same 4-way UltraSPARC-II system, but with different JVMs, it can be clearly seen that there's a large improvement from the 1.2.2 and the 1.4 JVMs in terms of scalability. This is particularly obvious with the 72-way UltraSPARC-III scores from Sun, which show scalability improving dramatically from the 32-bit 1.3.1 JVM to the 64-bit 1.4 JVM.

As the results above show, system scalability is about the hardware, the software (OS and application) and also how well the software has been optimized. Overall, SPECjbb scales more easily on the Pentium 3 Xeon systems than SPECint, which suggests it's not quite as bandwidth heavy, and from what I know, the benchmark uses only a few gigabytes of memory even on the largest systems. Still, systems with more cache or more bandwidth definitely scale better here. One thing SPECjbb doesn't show (and in fact very few benchmarks show) is how well performance scales as the complexity of the data is increased. This is important because it's common for data for real world applications to become increasingly complex and bigger over time.

Other Server Benchmarks

It's a lot easier to criticize benchmarks than write them, and what benchmarks are available do not always have up to date or comprehensive results, making comparisons difficult. One general problems I see is that most benchmarks focus only on performance or price/performance, which is fine for most HPC applications, and many workstation applications, while business customers are more interested in "what's the cheapest or best solution that can achieve these performance requirements". While it's useful to know the maximum performance of a server, in actual deployments, most actually average about 20-30% CPU utilization - how "real world" does a benchmark have to be to be useful?

For this article, it would obviously be more relevant to include some database benchmarks, and though there's several database benchmarks out there (the most well known being the TPC ones), some analysis suggests these benchmarks don't necessarily reflect most real world implementations. For example, even the "low end" systems for the TPC-C, TPC-H and TPC-W have around 2TBytes of storage, which is much more than what would be expected for 4-way and 8-way servers. I haven't encountered any experienced professionals who consider the TPC benchmarks to be good either.

Anyway, for TPC-C the 8-way Intel systems get nearly twice the throughput as similar 4-way Intel systems. However, with TPC-H, the 8-way systems were only about 50% faster, maybe because TPC-H has much more complex and difficult to optimize operations. The TPC-W results seem more difficult to interpret as it seems the vendors are still getting used to optimizing the setup for it. There aren't enough submissions for low-end RISC based systems to make a proper estimate of their performance and scalability though. Despite double the clock speed, far more memory bandwidth, Hyperthreading and better I/O the Pentium 4 Xeon based servers only improve on the previous generation by about 30%.

The easiest and most common way to optimize real-world database setups is to minimize the amount of storage I/O by caching as much as possible in main memory and optimizing the queries and application to be cache friendly. So database administrators or programmers tend to become pretty familiar with whatever tools the OS or database provides to show how frequent storage I/O requests are. So benchmarks or benchmarked systems where the bottleneck is the I/O is not necessarily representative of how real world equivalents would be run. Some high-end database applications really are limited by I/O though.

However, even with database systems that perform very few I/O read requests due to caching, being able to sustain high I/O rates is still useful - for rarer events such as backing up (or restoring) the database or rarely run queries (end of month, end of quarter, end of year kind of thing) that use a huge amount of data. Also, most databases run on dedicated servers, meaning that all requests and data goes over networking, so networking I/O can potentially be a bottleneck even if storage I/O is light.

Above it is suggested that most (well optimized) real world database systems will not be limited by storage I/O, making benchmarking an optimized database more of a CPU and system benchmark. However, this doesn't mean that SPECint (an integer heavy CPU and system benchmark) would be a meaningful substitute, especially for MP systems. None of the SPECint (or SPECfp) programs are multi-threaded (or at least, not used this way), only one seems to have a direct bearing on databases, which is an in-memory object database, which would be a lot simpler than an enterprise level SQL database. Also, they all seek to minimize shared library and OS usage while real databases would stress about every part of an OS' kernel and certainly make significant use of it's libraries.

For MP systems, the "rate" versions of the SPECcpu benchmarks are not much more than running the same benchmark multiple times at the same time - they don't have any dependencies like locking for example. Also, a MP system running a database would be running multiple types of queries at once regularly, and the data set sizes for each would be different, so any demands on the memory system will vary quite widely from query to query. However, with the SPECint_rate and SPECfp_rate benchmarks, the same benchmark, with the same memory access patterns will be running at the same time.

Though some micro-benchmarks could be used to attempt to probe specific aspects of database performance, developing more complex benchmarks is somewhat futile I feel, as many real world implementations vary so widely. Many real world systems would also make use of vendor specific features while an "industry standard" benchmark would have to be cross-platform and run on all databases. However, a lot of companies do use database application packages, which would have more predictable data structures and access patterns. There are benchmarks based on these packages, and perhaps make the most realistic database benchmarks. Please see the list at the end of this article for URLs to a number of server and workstation benchmarks.

Designing CPUs and systems for workstations and servers

When considering how to design a new processor and accompanying systems architecture, the designers will be guided by what parts of the computing market that are considered important for it. There are many broad computer markets - desktop, workstation, mobile, and server as well as embedded, and many price and performance points in each. As it takes a long time (and a lot of money) to develop high performance workstation and server CPUs, most designs get adapted to a number of tasks over time. For example a CPU designed for high-end MP systems might get slimmed down (less cache and system bandwidth) and turned into a desktop or workstation CPU and vice-versa.

Fortunately for processor designers, design features which are useful for workstations can also be useful for servers - faster clock rate, better branch prediction, executing more instructions at once, lower cache latency, more cache bandwidth, lower memory latency, more memory bandwidth, faster system design (I/O and multi-processor connections). However, features like SIMD instructions (processing multiple blocks of data with a single instruction, useful for 2D and 3D graphics, video, sound and "multimedia" in general) are of little or no use to almost all server tasks (except compute servers). This is because floating-point processing in general is often quite rare and the data structure and types are much more complicated with server applications.

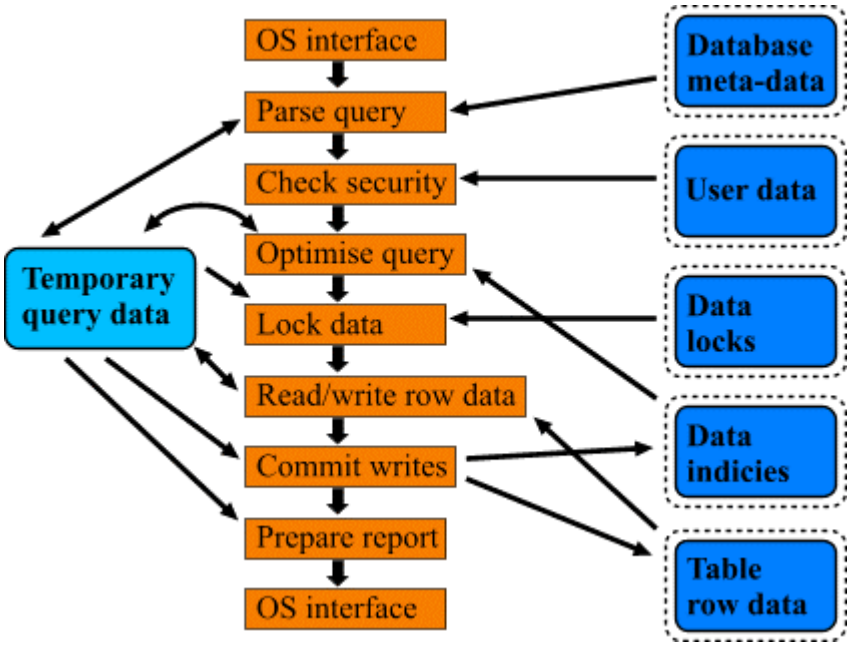
Programs on servers also have more complex logic and options, meaning they have more branch instructions and harder to predict branches than typical workstation programs. Also, the memory access patterns are mostly much too complicated for automated memory pre-fetching, which is useful for most workstation tasks. However, decent floating-point and SIMD performance can often be implemented without taking up more than 10-20% of the chip's die-size because CPU cache systems are taking up more and more space.

Data for workstation programs can be quite simple, and also very regular - for a simple 2D graphics package, it would just have thousands or maybe a few million pixels per image, each up to 32-bits in size (sometimes more for medical applications), as one big array. As far as the data is concerned, that's more or less it - more advanced programs would use tiles though - groups of smaller arrays. Something like a 3D animation program would be rather more complex and would have several different data groups - lists of 3D data points describing the scene (arrays of floating point data), and information for various effects, such as textures (2D images).

For 2D images the main types are editing are changing some of the pixel values, which would depend on the tool being used, and effects that change the whole image. Some of the calculations required for these edits may be relatively complicated, but tend to be localized and hence cache well for the tools. For effects that work on the whole image, it is often relatively easy to stream in the data for scene processing because the data is laid out simply - often just a long array of 32-bit values, which is also why SIMD instructions help. A 2D image is basically a fixed size array - changing the size just means making a new array and copying the data. Editing 3D data is more difficult to deal with because it's unbounded - the 3D scene takes up more data as more detail is added.

Designing for database applications

Data structures for SQL databases are much nastier to handle than those typical for workstations. A SQL database is a group of "tables" each of which has a variable number of "rows" with a fixed number of "columns", each of which can be of any data type the database supports. When including all the possible options a database may support this may be in the order of 100 different data types even for the simpler databases. The quantity and datatypes of the columns is entirely up to the application and not all SQL databases support changing the structure of a "live" table. In concept this is a bit like changing the size of a 2D image, but one reason why in practice it's much more complex to change the structure of a SQL table is that any table and any database can have multiple read or write operations running at any one time, while workstation applications are just responding to one user.



The above diagram roughly illustrates the main operations and groups of data for processing common SQL requests, heavily simplified. The blocks on the right shows persistent data, which must also be kept up to date with the permanent storage. The dashed line around them indicates that changes to this data goes uses implicit lock (set by the database) to make sure updates by multiple CPUs can occur at the same time. The temporary data for each query is only accessed by one CPU at a time, so doesn't need such locks (though actually some databases allow a query to be spread across multiple CPUs).

Though some databases support proprietary alternatives, the general way in which a SQL database lookup or edit is performed is with a SQL statement, which is a human readable text string, almost like a sentence in English. This shows another crucial difference between SQL servers and workstation applications - in desktop applications an edit or view operation is tied to a tool that can easily be tied to a specific piece of code that performs the required operation. However, a SQL engine is rather like the core part of an interpreted language as it has to decode and optimize each operation one at a time - the proprietary alternatives some SQL databases provide essentially involves pre-compiling queries.

A lot of lookup queries (and write operations) simply affect one row in a SQL table, where the particular row (or rows) affected is based on a set of compare operations on specific columns in the table with the data in individual rows. A common compare being a match on a unique identifier number each row can have - for example, a message id number for a discussion board. The hopelessly slow way of finding where this data is actually stored would be to go through all the rows in the entire database (if necessary) loading up the data and comparing it. Believe it or not, this is in fact the default operation. To get decent performance, when creating (of modifying) a table, an option can be given to "index" a particular table - construct a fast lookup, including for doing range checks. Providing indexes are setup to help the query, most lookups can be done in a small fraction of a second, providing the database has the data required cached in main memory.

The various processing tasks involved with executing a SQL query would mostly involve running a loop, processing a small list of parameters or using that list of parameters to look through some or all of large groups of data. This is not unlike workstation tasks such as processing a 2D image or a 3D scene, which would loop through elements in an image or scene graph. There are many differences though - there's little floating-point processing in databases, and most operations (on a high level) involve building up lists of data, rather than processing data. The code is also often heavy in branch instructions, and there is generally a lot of code as well - many loops of code, mostly large and complex.

Within a single loop, later operations would mostly depend on earlier operations, which can limit the effect of techniques such as out of order execution - too many data dependencies or too much code to do across loop iterations. For the same reason, a cache miss can be quite painful as it's much more likely to cause a pipeline stall due to data dependencies. Unlike many workstation algorithms, the data is rarely accessed in a long sequence (as a stream), due to using more complex data structures like hashes and linked lists (more efficient than the alternatives though), which means data pre-fetching is unlikely to help much.

A database with very little data could fit in the top level caches on many CPUs, which would mean stalls waiting on cache misses would be small so performance would mostly depend on the core CPU performance (clock rate, and IPC for the code). For large databases (those with gigabytes of data or more), if most queries just re-use a small amount of data, then again cache-hit rates will be good. Even if the cache hit rate on the stored data itself (the data the programmers/users see), information about the database structure, list of keywords for parsing a query, indices, locks, and temporary result data, will be a lot smaller, so should cache reasonably well in most cases.

Overall, average memory latency is perhaps the most significant factor for databases. Keeping average latency low means having fast caches (low latency), large caches to keep the hit-rate high, low memory latency for larger databases and in the worst case, low latency I/O for databases that fit poorly into main memory. For MP systems, the higher the cache hit rate, the less the memory system is used, leaving more for other CPUs, improving scalability. Of course, the larger the cache the higher the expense, so larger caches are better suited to higher-end products. By way of example, though it's hard to say how representative it is, but in this recent ZD Serverbench 4.1 (TPS) benchmarking article, the Pentium 4s with 512KByte level 2 cache got 50% higher performance than those with a 256KByte level 2 cache.

The temporary data for processing a query would cache well in most cases, and since one CPU would only use the data, there wouldn't be problems in MP systems. However, writes to the persistent data would need to be shared with the other CPUs - writes will cause cache misses even for small databases on large caches. For application types that perform writes very frequently (a credit card payment system for example, or a share trading system), the cache side effects of writes will have a noticeable effect on performance. Optimizing this requires keeping latency between all caches low (with decent bandwidth) as well as keeping main memory low.

Data integrity with databases

Locking helps deal with problems and side effects of updating or adding data to the database. When a row is added or deleted, or when data is changed, the change has to be written back to a hard disc (or other permanent storage) and in memory caches have to be updated, and also any indexes. Because writes can be quite common, and also overlap with other reads and writes, it's important that they be quick and also scalable. Unfortunately, data structures that are quick to update even for very large data sets (lists, hashes, structures trees) are often slow on lookups and also have a lot of padding or hidden data which takes up extra memory.

Part of the problem with designing databases, is that for writes to be absolutely reliable (in case the server crashes or there's a power failure for example), a write operation can only complete when completely written to the storage system. This can take a while, from the CPU's point of view. Higher-end storage systems will have large amounts of NVRAM (Non Volatile - doesn't lose any information when power is lost) so that as soon as incoming data has been written to the NVRAM cache, the write operation is "complete". In some RAID systems, the memory cache on the hard discs may actually be turned off - if the hard discs don't use NVRAM for cache, for reliability reasons, the volatile cache has to be turned off which is why higher-end SCSI and FC-AL hard discs mostly come with NVRAM.

It's also important of course that the database and the operating system can ensure that writes are performed correctly, and in case of a serious failure, that the database can be rolled-back to a known and correct state. I have been unfortunate enough to be in a situation where a database for an e-commerce site got corrupted due to file system or operating system bug, and it took about 3 days to get the database back into a reasonably working state - the database couldn't handle this itself. In many situations, that just wouldn't be acceptable - imagine if a stock trading system lost share dealings.

At the highest-end of storage systems there are features like having a "mirror" storage array a long distance from the main one, but where updates are kept in full synchronization - writing data to the main array won't be complete until the spare is written to as well. This means that if some disaster befalls the main storage array, the backup array is completely up to date and can take over handling requests immediately. This is very hard to do while keeping performance high - often requires high-speed dedicated fiber-optical links.

Because of the complexity data integrity introduces, an additional problem is to make sure that changes to the in-memory (and on disc) data don't "collide" with reads or other writes. For example, the data in an index may have to be shuffled around a bit on an update, but if a separate operation (for a different query) accesses that data while it's being updated, invalid or inconsistent data could be read. To stop the database tripping over itself, implicit locks are required. Explicit locks are used by database programmers to ensure that combinations of read operations followed by writes do not interfere with each other.

A fairly simple way to implement implicit (or explicit) locks is to lock the whole table when doing writes. However, this means that no reads (let alone writes) can run while the write lock is in place. Even a 2 or 4-way database servers can manage 1000 simple read queries per second, but if a write operation (which has to be reliably committed to the storage device) takes 0.1 second and reads can't occur during that time, that same database will be able to manage 1 write and 900 reads per second. If 9 write operations are occurring every second, that'd leave only enough time for 100 read operations.

With some of the cheaper or older SQL databases out there, this is pretty close to exactly what can happen. The more scalable solution is to use finer grained locking (for example, row-level, instead of a whole table) and more flexible data structures that can allow reads (and maybe writes) to overlap with writes to the same table when the particular data being updated doesn't overlap. The extra complexity means the sustained read rate might drop to 500-700 queries per second, but writes would then only have a small effect on overall performance.

With a single database having multiple tables, and with each table having multiple data structures for it (locking, indexing, and the data itself), a single database will have many data structures of different sizes, access patterns and cache hit rates. Many of the data structures would be quite small, so would benefit from small fast caches, except that they would be "fighting" for cache hits with some of the larger data structures. The larger data structures should benefit more from large slow caches more than small fast caches. With databases benefiting both from large and smaller caches, the lower-end CPUs will likely just go with a small but fast on-die level 2 cache, while the higher-end CPUs (for 4-way and above) would most likely come with three cache levels.

Usefulness of 64-bits

A CPU with a 32-bit memory addressing (virtual memory system) can only support 2^{32} bytes of data (4 GBytes) in total, though OS limitations may reduce the practical limit to 2GBytes total or to 2GBytes per application, or per thread in each application. If a CPU supports more than 32-bits for the virtual memory system, then more than 4GBytes of memory can be installed and "seen" by the OS, but if the OS is still 32-bit (doesn't use 64-bit memory pointers) then each application or thread will still be limited to 4GBytes. To some extent this is what the Intel Xeon CPUs do - they have a 36-bit virtual memory system (64GBytes) but no thread can use more than 4GBytes - though a single application can have multiple threads with separate address spaces, meaning the application as a whole can use more than 4GBytes of memory.

In the "bad old days" of 16-bit CPUs, the limitations of a 16-bit address space became critical quite quickly and the CPUs back then supported extended memory methods to allow 20/24-bits of addressable memory. Application writers also used special instructions to move between address spaces so that a single thread could in fact manipulate more than 16-bits worth of memory, though this is rather nasty to program. Additionally, since a 16-bit address space is only 64KB, which is pretty small even 10 years ago, switching address spaces constantly is inefficient, and 32-bit arithmetic is also needed often enough that moving from 16-bit to 32-bit CPUs provides a modest speed advantage.

Similar tricks could be used to "emulate" a 64-bit address space and memory pointers with a 32-bit CPU, but unless the OS and compiler could cleanly and transparently manage this (which might be impossible in practice with all the important legacy code out there) it may be useless in practice. Currently, only high-end workstations, and heavy duty servers really need more than 4GB of memory - main memory is much faster than I/O, and when 32-bit pointers or array indexes become insufficient using 64-bit versions is much simpler, efficient and more portable than using hacks to get around the problem. In most high-end environments, support for legacy code will be very important, as will be reliability and maintenance costs.

For the most part, on modern CPUs, adding 64-bit integer registers and functions and extending the virtual memory system beyond 32-bits adds little extra size or design time to the CPU - a few percent to the size and maybe 10% or so to the design time. Some parts of the chipset would also need 64-bit addressing, which would add a bit to system costs. For embedded CPUs, where cost and hence die area are very important, many CPU designers intend to provide 64-bit registers and data operations for 0.09um designs, as the overhead is becoming so small.

Adding extra instructions to use the new 64-bit registers while still supporting the old 32-bit versions, and getting compilers to support this can take more effort. Still, in the end, the extra complexity for the CPU itself is quite small, but very valuable. Adding 64-bit support is a bit like adding MMX, VIS (Visual Instruction Set - Sun's "multimedia" instruction extensions), SSE, 3D-Now! or AltiVec - uses new instructions to make certain applications faster, when the old method required multiple slow instructions for the same operation, while adding only small amount to the CPU's complexity.

The difference between 32-bit and 64-bit systems is more social and economic than anything else - it's the bigger companies (in the "enterprise" space) that need 64-bits really. Even in the "enterprise" space not all running systems truly need 64-bits either, but it's a lot easier to support a single architecture (or fewer architectures) internally.

Large memory systems

Having a 64-bit CPU is just the start though - the OS, developer tools and applications also need a lot of development work. Developers need 64-bit compilers, and 64-bit OSs to run them on, and also enough demand for such products to justify the effort of porting, testing and supporting the 64-bit version.

Having an OS that can efficiently support many gigabytes of memory (or hundreds of gigabytes for the larger systems) is actually quite hard. The physical memory (and the I/O address space) in a computer is mapped to logical addresses by the OS, and the applications only see the logical addresses. The physical memory is split into blocks called pages, where the logical addresses for all data in a page map to the physical addresses as a group. A page would typically be in physical memory but it can also be stored to disc temporarily ("paged out") using a virtual memory system.

The CPU needs to be aware of the pages so that it can determine if a memory address request is valid or not, and a cache of page address mappings is kept (often called a TLB, Translation Lookaside Buffer) on the CPU to optimise this. A cache is used because the number of possible pages is too large to keep permanently on the CPU - the default page size on x86 CPUs is 4KBytes, so a CPU with a 32-bit address space can have up to 1 million pages. The largest TLB most x86 CPUs have is 256 entries currently, which is only enough to map 1MByte of data with a 4KByte page size. A cache miss on a TLB means doing a extra load from memory and can significantly slow down applications with large data sizes.

For this reason most CPUs support much larger page sizes, though the OS needs to support these larger page sizes for it to be used. Some OS provide APIs that developers can use to allocate memory with larger page sizes, which needs developer support to be used. The task of simply allocating, delocating and relocating pages in a scalable and efficient is actually quite hard, requiring complex algorithms. For OSs with NUMA (Non Uniform Memory Architecture) optimisations the page allocation algorithm will also attempt to allocate pages "local" to the CPU that needs them. In most large systems today, which have many CPU/memory boards, memory on the same board as the CPU can be accessed much faster than memory elsewhere, so the more often local memory is accessed the better. The next major revisions to server version of Windows (Windows .NET Server) and Linux (2.6 kernel) have various NUMA optimisations, and an update to Solaris 9 will get "Memory Placement Optimization."

Volume effects on CPU manufacturing

CPUs like the AthlonXP, Celeron, Duron, Pentium 3, Pentium 4, and PowerPC are all in "high volume" - some shipping over 1 million every month. CPUs like the AthlonMP and the dual-processor Pentiums use the same design as the single-processor versions, but have stricter qualifications, and are generally in much smaller volume as a product but from a manufacturing point of view, they are the same. These CPUs can be in high volume because they're used in desktop systems, which are in much higher volume than workstations or servers, though of course "high volume" and "low volume" are relative terms. CPUs like all Itaniums, Pentium 3 Xeon, Xeon MP, SledgeHammer, all UltraSPARCs have a different design to any volume parts and are in volume of about 1000 - 10,000 chips per month.

For a new CPU design, it normally takes at least 9 months of testing, correction (from the first "tape out", i.e. first attempt at silicon) and optimisation to get it ready for production and available to customers in volume. This costs a lot, and there is also the design time for the CPU, which would cost a lot more for a completely new design. For a completely new CPU design, including chipsets and whole systems, development costs can be upwards of $1Bn over a 4-6 year period. Selling more CPUs based on the same core design means spreading the costs out more, so each part can be a bit cheaper.

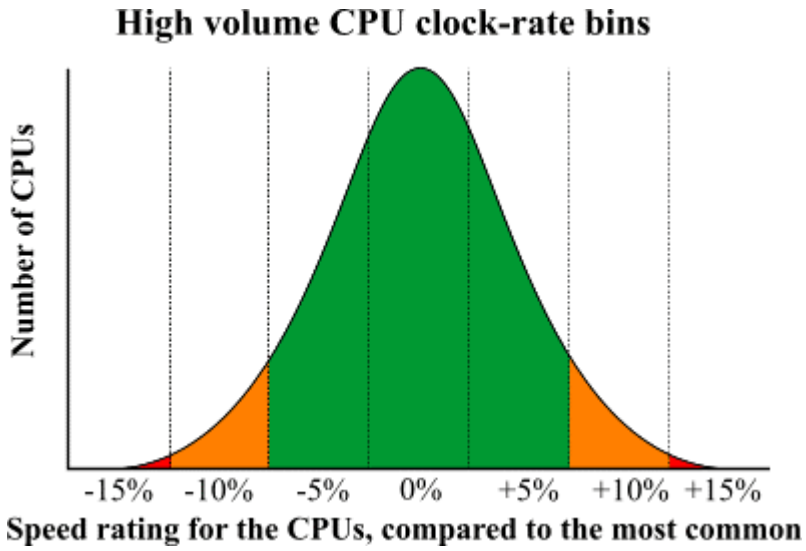
CPUs in higher volume have some other advantages as well as improving performance and yields for CPUs (and chips in general) is a statistics game in many ways. For example, it's not always clear if a minor revision in the manufacturing process or CPU design is better or if it just got "lucky" in the first batch. Having more samples enables more accurate measurements, and also means more variations can be tested at once. Naturally, having more resources for research and development helps significantly, and Intel, who makes more money on CPUs than everyone else put together, definitely have an advantage.

Genetic algorithms are used to improve yields (and performance) over time - run many small variations at once for production, pick the best one after gained enough test data, then using that as a base, run more small variations again, and keep on doing this in a feed-back loop. CPUs in low volume that are based on a high-volume part can get much of these benefits as well as the main difference is often the amount of on-die cache which is comparatively simple. The high-end Pentium 3 and 4 (Xeon MP) designs and SledgeHammer would be examples of this.

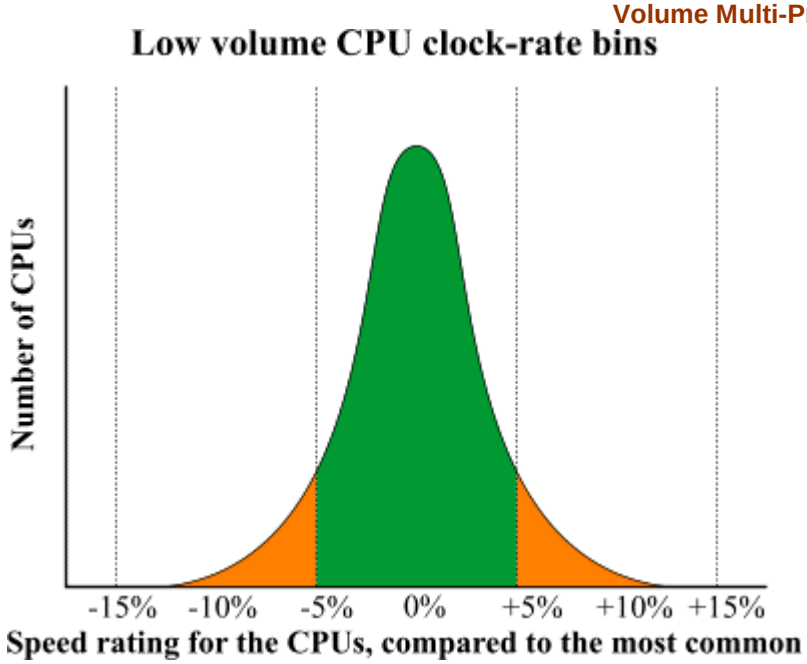
AMD and Intel's primary business is selling high-performance CPUs, so naturally they focus most of their research and development there. IBM (who manufacture their POWER CPUs internally as well as HP's PA-RISC and the Alpha), Motorola (who manufacture the PowerPCs for Apple's systems) and Texas Instruments (who manufacture Sun's UltraSPARCs) have a much wider variety of chip manufacturing processes, and for them high-performance CPUs is a small volume business, but with high margins. High-performance CPUs require more advanced process technology compared to ASICs, DSPs and embedded chips in general, but the manufacturing tools used are mostly the same, just programmed differently. So for the more general manufacturers, low-cost high volume chips are made along-side higher-cost lower-volume CPUs, and the lower-end chips benefit from the advanced process technology for CPUs trickling down and the CPUs benefit from the volume. But overall the CPU specialists, AMD and Intel, have an advantage.

CPU speed and volume

To properly ship any CPU at a particular speed grade, a minimum amount of parts at that speed are needed for testing and qualification. CPUs in higher volume would likely have testing spread out over many OEMs, so new faster speed grades would have a higher minimum requirement in absolute numbers. But because total volume is so much higher, the percentage is rather small and smaller than low volume products. My best estimates suggest companies like AMD and Intel announce new top-end speed grades for high volume products when they have 0.5-2% of parts yielding at that speed. This is also why the top speed grade comes at a large premium, often costing more than twice that of a CPU only 5-10% slower. This adds a lot to the cost for desktop systems, so demand for the top speed grades is also low, in comparison.



The above graph roughly illustrates the frequency (the vertical scale) at which a particular CPU hits a particular clock speed (the horizontal scale). The dashed vertical lines indicate how CPUs in high volume may be "binned" (which frequency rating they will be given when sold to customers), showing the percentage of the total volume that get graded in each bin. Instead of giving an actual clock-rate, the speed difference (compared to the most common speed) is shown. The green colored regions are for speed grades that will definitely be shipped (to customers), while the yellow ones might, and the red ones rarely. For high-volume CPUs, speed improvements are not applied to all chips in production at the same time (would be impossible anyway as it takes 3 months to manufacture a CPU and changes can't always be applied part-way through) to help reduce risks. This flattens the curve somewhat, compared to low-volume CPUs.



Higher-end CPUs are always in low volume as are lower-end CPUs when they're just released or when they've just been moved to a more advanced process technology (eg from 0.18um to 0.13um). The bin splits for low volume CPUs may look like the above, though in some cases 90% or more may just be in one bin. Most high-end CPUs ship at just one or two frequencies. Unlike high-volume CPUs, low volume ones would almost all be exactly the same process variant and CPU design variant (though they change less often), so there will be less variance in the speed distribution.

For high-end systems, the CPUs are often a relatively small part of the system cost, and high-end customers are also more willing to use the fastest speed grade available. So, announcing availability of a fast speed grade when little volume is available can cause customers to be upset. For higher end CPUs, vendors will often sell the fastest parts to special customers while (relative) volume of them is low, and delay an announcement of public availability until the volume of those parts improves. For these systems, customers also prefer a small set of speed grades, and for example Intel once reduced the planned number of speed grades for Pentium 3 Xeon CPUs because of complaints. Being only able to publicly offer new top-speed parts when (say) 20-50% of CPUs reach that speed also reduces the top-speed they can ship at. Looking at the top shipping speeds for high-volume Pentiums compared to low-volume high-end Xeon designs suggests the get a clock rate advantage of about 10-15% overall.

When the Pentium 4 initially moved to 0.13um, the top speed only improved by 10% (from 2GHz to 2.2GHz), with the minimum improving from 1.3GHz to 1.6GHz. The average improvement initially with 0.13um was probably 20-25% however. Probably about 5% of the 0.13um Pentium 4s reached 2.4GHz at the time, but with the low volume at the time, this wouldn't be enough for a launch, and instead were probably used for testing and qualification with OEMs and other special customers. With a combination of increasing volume and process improvements, the top clock speed is increasing. Similar things should happen as AMD moves the Athlon to 0.13um. For higher-end CPUs, which are always in low volume, a significant process improvement tends to have a more clear and immediate impact.

List of Server and Workstation Benchmarks

SPECweb99 Relatively simple webserver benchmark.
http://www.spec.org/osg/web99/
SPECweb99_SSL Secure Socket Layer version of the above
http://www.spec.org/osg/web99ssl/
SPEC SFS97 File server benchmark based on NFS
http://www.spec.org/osg/sfs97r1/
SPECmail2001 E-mail benchmark
http://www.spec.org/osg/mail2001/
SPECjbb2000 Java Business Benchmark
http://www.spec.org/osg/jbb2000/
ECperf Enterprise JavaBeans benchmark - will become SPECjAppServer2001
http://ecperf.theserverside.com/ecperf/
http://www.spec.org/osg/jAppServer/
SPEC CPU2000 Integer and Floating Point benchmark
http://www.spec.org/osg/cpu2000/
SPEC OMP2001 HTPC benchmark
http://www.spec.org/hpg/omp2001/
TPC-C Old transaction processing benchmark
http://www.tpc.org/tpcc/default.asp
TPC-H Ad-hoc query and decision support benchmark
http://www.tpc.org/tpch/default.asp
TPC-R Business reporting and decision support benchmark
http://www.tpc.org/tpcr/default.asp
TPC-W Transactional web e-Commerce benchmark
http://www.tpc.org/tpcw/default.asp
Oracle Applications benchmark
http://www.oracle.com/apps_benchmark/html/index.html?results.html
Fluent CFD benchmarks Computational Fluid Dynamics is important in HPTC and Fluent is the market leader
http://www.fluent.com/software/fluent/fl5bench/index.htm
Server and workstation benchmark records at Ideas International SAP, Lotus Notes and more
http://www.ideasinternational.com/benchmark/bench.html