



OPTIMIZING multiprocessor systems

EFFICIENT OPTIMIZATION OF MULTIPROCESSOR SYSTEMS

BEGINS WITH AVOIDING INEFFICIENCY IN THE FIRST PLACE.

OPTIMIZING A SYSTEM means more than getting as close as you can to 100% usage. It also involves getting close to 0% inefficiency. Designing systems using

multiple processors, each with multiple cores running multiple threads, presents a difficult optimization challenge. Higher order languages, framework environments, APIs, resource managers, and a slew of other abstractions enable you to step back from the complexity of a problem and view it from a higher, simpler, and more comprehensible level.

Abstraction's great advantage is that you can quickly sketch out your design and fine-tune it where you need better performance. In other words, optimizing a system means determining where abstraction introduces the most inefficiencies and then selectively replacing those abstractions with more concrete descriptions to remove those inefficiencies.

ONE PLUS ONE IS MUCH MORE THAN TWO

One challenge of designing multiprocessor systems is that, although you can fairly easily prove that an individual task falls within system constraints, doing so in the presence of additional tasks contending for the same resources, such as computational cycles or memory bandwidth, can be difficult.

Multiprocessor architectures solve small and large problems. In a network-processing application, for example, in which several processors concurrently process packets so that one of the packets completes before a new one arrives, you may need to solve a lot of little problems with several processors. You might also see such an architecture in a DSP or an encryption farm, which sends small problems to a central location for processing. With large prob-

<i>At a glance</i>	68
<i>Quantum math:</i> <i>2.5+3.5=7</i>	68
<i>Simulation across time and space</i>	70
<i>Tackling allocation</i>	72
<i>More on tools</i>	74
<i>For more information</i>	76

lems, such as image or signal processing, you need to break the problem into manageable pieces, with a separate processor core solving each piece. Note that some architectures employ both techniques: They break many small problems into pieces that separate processors solve.

Breaking a problem into pieces has the advantage of enabling more efficient use of resources. You can define how many cores run a process or task based on how performance-intensive the task is. A key disadvantage of breaking up a problem is that you have to stitch the pieces back together—a process that requires more overall processing than a single processor. It usually involves passing intermediate results to the next task and resolving seams; when you break an image into four quadrants, you have to resolve the seams that result between the quadrants.

Software partitioning is often arbitrary; thus, it artificially constrains the system. You base it on boundaries logical to human understanding rather than practical boundaries, such as the number of cycles to execute. It takes a lot of effort to create a software partition and even more effort to later shift boundaries. If you discover an imbalanced task—one that requires 2.5 cores' worth of performance—it may become difficult to repartition tasks to recapture the unused half core (see sidebar "Quantum math: $2.5 + 3.5 = 7$ "). You have to focus on other, unrelated abstractions that might be more efficient than the task-boundary abstraction. Optimizing such abstractions becomes more challenging, because

AT A GLANCE

- ▶ In multiprocessor designs, abstracted design tools allow you to trade performance for ease of design.
- ▶ Optimization is a process of determining which abstractions bring in the most inefficiencies and then piercing those abstractions to fine-tune performance.
- ▶ Multiprocessor-design tools let you focus on solving your problem at a logical level; however, such tools are far from optimally resolving problems at the physical level without human intervention.
- ▶ No matter how comprehensive your tool chain, the completeness of your design limits your optimization efforts.

you have to remove more abstractions to remove more inefficiencies.

WHAT'S INSIDE? WHO CARES?

Some environments use software constructs to abstractly access resources. For example, several NPUs (network-processing units) abstract table searches through an API, shielding the programmer from having to know which coprocessor the hardware team may select. It also enables code to be portable in case you need to change coprocessors. However, even a thin API comes at a cost. You don't use all coprocessors in the same way. If the API successfully captures differences, you can be sure it'll cost you.

Even if you're talking only two extra instructions per access, using the resource five times in 200 lines of executed code results in a 5% inefficiency cost.

A similar form of abstraction is using library functions. The more general the code, the more applications the library can service. Such generality comes at the cost of the inefficiency of entering and leaving the function, as well as telling it exactly what you need to do in each use. For established protocols or well-defined algorithms, for example, libraries give you robust code for a part of your system that offers little differentiation. But, unless you optimize the code for your processor of choice instead of merely porting it by altering a few internal I/O functions, the code will be less efficient than it could be. Additionally, consider the cost of using a function. Instead of directly using parameters in their original memory location, you need to pass them or pointers to them to the function and then retrieve results using additional cycles.

Does abstracting resources make sense? Consider that each generation of coprocessor offers new features and commands that change how you use it. Current-generation coprocessors support macros, in which a single command executes several commands. The only way to take advantage of the macro optimization is to rewrite the code using the legacy API. The old API will still work, but using it will cost you dearly. Consider that you're going to need to reoptimize your system anyway, because all of the task-contention dependencies will

QUANTUM MATH: $2.5 + 3.5 = 7$

Instead of basing partitioning on function, try to partition on a smaller, more granular scale. For example, you might shoot for a performance quantum of 50 cycles, which becomes the currency of your time budget; every task should be some factor of 50 cycles, and tasks should be as close as possible to one or more than one quantum. Now, when you allocate cores, you can lump together several tasks into a single metatask. Say that you have partitioned two consecutive metatasks, such that you need

to allocate 2.5 cores for MetaTask A and 3.5 cores for MetaTask B. With traditional partitioning, you have to allocate seven cores unless you do a lot of handwork. Using quanta, you can increase the size of MetaTask A and reducing MetaTask B or vice versa to achieve six cores with perhaps only minimal handwork. Remember that boundaries are largely arbitrary, so you should treat them as options, not constraints.

Note that you might choose from several quanta, given that

performance is only one of the many limited resources for which tasks compete. Additionally, you can profile resource use to determine how to partition tasks; knowing that one metatask is more memory-intensive than another can help you determine where to shift the tasks.

A key concept worth highlighting is that using metatasking with most tool sets requires some cleverness. You can manually manage any abstraction that a tool could offer but doesn't. You can write object-orient-

ed code in assembly, but you won't have a compiler telling you when you've violated the rules. Thus, you can create small tasks and abstract the stitching between them yourself. If you place two tasks on the same core, much of the stitching between them may become unnecessary. Rather than hand off intermediate results, you can continue to use them. Such flexibility is useful because you may not discover until debugging that repartitioning an algorithm might increase efficiency.

change. The API is worth using if you want to support a second coprocessor as a backup, but you have to consider the thickness of the API and how much inefficiency it adds.

INTERCORE COMMUNICATION

Abstracting resources hides many of the relevant characteristics of accessing that resource. For example, you could store a value in local core memory, another core's memory, another processor's memory, or external memory. To the programmer, all accesses appear equal, but they are not. Each access has a different latency.

Working solely at the abstraction level, you solve problems on only a logical level, leaving your tools to solve them at the physical level. In many cases, the tools can manage resources with more detail than can a person, but only if the abstracted representation doesn't cripple optimization efforts. For example, say you need to allocate three cores for the processing of Task A. If the tool clumps Task A onto three of a processor's four cores, you may overload any core resources that this task uses intensively. Ad-

ditionally, when Task A hands off intermediate results to Task B residing on another processor, it may have to pass through memory. This technique is less efficient than sending results to another core on the same processor using registers. By collocating Task A and Task B together onto three separate processors, you improve task handoff.

Note the added complexity that working on the abstraction level brings to profiling. Say that Task B requires only two cores. The first two instantiations of Task A can use core registers to hand off intermediate results to Task B on the same processor. The third Task A might have to use memory to hand off intermediate results to either of the Task B instantiations. Your profiling tools must be able to treat the three instantiations of Task A as individual tasks. If the tool aggregates or averages results, you lose the ability to see where inefficiency lies—in the handoff and in certain instantiations. Breaking the processing into many tasks running on their own cores makes the allocation problem extremely complex (see sidebar "Simulation across time and space").

Profiling is one area in which abstrac-

tion really does help. Because Task A supports two kinds of handoff, being able to abstractly define the handoffs and compile different versions of the task based on its allocation frees you from having to manually manage the process. Even if your tools don't directly support such abstraction, and, depending on where a task resides, you can indirectly use macros to create alternative code and compile several versions of the code as necessary.

Changing the physical allocation of tasks between cores and processors creates different profiles (see sidebar "Tackling allocation"). If the design tools, not the programmer, handle intercore communication, then the tools hide from the programmer the true latency of communication and the ability to optimize that communication. Thus, abstracted tools need their own tools—in this case, tools that enable them to optimize the physical configuration. In other words, the compiler can no longer look at code as a static list of instructions; it must view the code as dynamic instructions with a history and performance profile. The compiler must be a companion tool with the debugger, simulator, and so on. You no

SIMULATION ACROSS TIME AND SPACE

One important shortcoming of simulation tools is their determinism: You run the same test data; you get the same results. However, in real-world multiprocessor systems, when two tasks simultaneously access the same resource, Task A sometimes wins, and Task B sometimes wins, affecting latencies for both tasks. This phenomenon is known as variability over space: identical initial starting conditions with different results. It is an issue, because a deterministic simulator allows one task or the other to win every time two tasks contend for a resource.

Space variability is important, because any data offers a spectrum of results. A single simulation yields a result within the spectrum. However, if you get the same result each time you simulate, you cannot determine

the width of the spectrum, where in the spectrum this result falls, or the probability of the result you do see. If you base worst-case calculations on this single result, you will underestimate or overestimate system needs and not know which way you erred.

Most simulation environments employ statistical-analysis techniques that assume a sample space having hundreds of thousands of lines of code. In DSP applications, most processing takes place in a limited code set. For network processing, code sizes are 100 to 1000 lines. Statistically, miscounting latency by a single cycle in 1 million lines of code is 0%. On an NPU with 200 lines of code, it's 0.5% per instance.

If Task A always wins contention for a resource in simulation, Task B will profile with

more latency than it might in real-world execution, and Task A will profile with a corresponding amount less. In other words, if Task A wins 50% in the real world, Task A always winning represents an extreme case. As a consequence, you might allocate too many cycles to Task B and not enough to Task A. One argument says the Task A/B issue is negligible, so you need neither measure nor worry about it. However, you won't know it is negligible unless you measure it.

The more data points you have, the more confidently you can estimate the result spectrum and the probability of a particular result. Because running multiple sessions with the same data yields no more information than running a single session, you need either multiple data sets or longer simulations.

Note that Task A's always winning may actually reflect real-world results. Given different trace lengths between cores, Task A may be physically closer to a resource and always win race conditions. However, few tools have a methodology for describing physical-core relationships, such as whether a core is closer than another core to memory than it is to a search engine. Cores equidistant from a resource alternately win race conditions. Additionally, a task may have instantiations on several cores; each has a different real-world profile, given its ability to access resources. Thus, you must be able to collect simulation results for each instantiation of a task. Don't just execute one instantiation and assume that it represents all cases.

longer use tools at only a single point in time; you need to be able to use all of them together.

Profiling intercore communication is complex. Your ability to monitor messages between internal cores depends on how much visibility the processor offers, as well as how intrusive such monitoring is. Measuring communication efficiency between processors presents a different problem. Four processors in a mesh communicate over six channels, requiring several logic analyzers to simultaneously look at all traffic. You might consider using an FPGA inline during development to analyze traffic.

Keeping common values local not only reduces latency, but also conserves bus bandwidth. Unless the simulator can profile variable use for the compiler to optimally map memory, you'll be stuck trying to define an efficient map by hand. Consider two processors, P1 and P4, connected through processors P2 and P3.

If tasks running on P1 and P4 exchange a lot of data, you can reduce latency by physically connecting P1 to P4. Although profiling tools may help you uncover information that suggests optimizing allocations in this way, most do not assist in the optimization. The compiler does not take into account the physical relationship between cores, and quickly testing new maps (optimizing using an abstract map of task dependencies to eliminate "obviously" inefficient combinations rather than completely simulating the system for each combination) is not an option.

You need to be able to redefine, repartition, and reallocate the system as you simulate, debug, profile, and learn new information. Current tools resist repartitioning by making it difficult to carry concrete optimizations back to abstracted models. Optimized code pierces the abstraction bubble, and the more concrete details you add, the more difficult it

is to maintain the abstraction. At some point, your investment in the hybrid model forces you to commit to it.

TOO MUCH AND NOT ENOUGH

As you increase the number of processors or cores, each processing unit becomes more of a black box. Collecting aggregate results is relatively straightforward, but being able to examine these results at the individual data-point level can open the door to more efficiency. For example, one data type may stall more readily the system than another. Knowing what kind of data causes these stalls gives you better insight than simply knowing where the stalls occur.

Note that, if you base task execution on a scheduling algorithm, a task contends with all other tasks in myriad configurations, so the aggregate does give you a useful measure. However, it is difficult to optimize over a wide range of configurations. Consider, then, limiting the tasks

TACKLING ALLOCATION

Optimizing a system based on traffic patterns gets tricky if patterns change based on time of day. For example, data traffic may exceed VOIP (voice-over-IP) traffic during the day and do the opposite at night. Instead of forcing you to simultaneously allocate cores for both worst cases, a dynamic system allows you to reallocate cores and thus use a shared subset of cores that serve VOIP when it is burdened and data when it isn't. Note that you'll want to optimize your system for both time-of-day configurations.

Optimizations become problematic as an application matures. For example, you may anticipate a limited use of a new protocol and set aside a certain number of cores to handle the load. However, once the new protocol is available, demand for it may rise while demand for other protocols drops. Have you implemented a means to adjust core allocation in the field to extend product life?

One way of handling QOS (quality of service) is to dedicate specific cores to for certain priorities. Thus, if the low-priority cores are full, a low-priority request will not consume slots reserved for high-priority requests. You can also guarantee a minimum bandwidth for certain protocols by dedicating cores to the protocol rather than having cores that can handle any protocol. The challenges are profiling such a system and determining the cost of dedicated cores on overall throughput.

If you have to pay royalties for code, the number of cores and threads you dedicate to a process directly affects the cost of your system. Although you might not save the most cycles optimizing a system around this code, you could save a significant amount of money, because the more times per second you can run the code, the fewer instantiations you have to pay for.

Running a single task on a core significantly reduces design

complexity. For example, a core has only so much local (read: fast) memory available to it. A core running several threads must map out the local memory so that there is no crossover and corruption between tasks. Thus, each potential task that can run on the processor gets only so much local memory. Limiting which tasks run on a particular core frees more local memory for those tasks.

Multithreaded systems usually limit the number of threads running concurrently, basing the number on a hard system constraint, such as the number of register banks available for zero-context switching. With dynamic data, the processing load of each thread differs. Schedulers that focus only on priority place pending tasks in the next available slot without regard to how the core is loaded. For example, placing a memory-intensive task with a processor that is already running several memory-intensive tasks will more likely cause the processor to stall. The

processor will execute no code, because all threads are blocked. Scheduling must take into account typical profiles of each task, accounting for loads on various resources and grouping dissimilar tasks to reduce overall internal contention.

For real-time data, the ability to control scheduling is important in managing the latency of high-priority flows. If a higher priority task arrives, can it take precedence? In other words, can you reflect the priority of the data in the priority of the thread processing it, or is the priority locked once a process starts? At a more granular level, if the task manager determines that a resource is under high contention—for example, during a memory request or when the search queue is full—can you identify tasks that avoid the bottlenecks and give them priority while the condition lasts instead of letting high-priority tasks add contention and lead to a processor stall?

a cluster of cores can handle. You can reduce the number of combinations and begin to identify which configurations present the most inefficiencies. You can eliminate these configurations by differently limiting the tasks. The configura-

tions you leave will occur more often, so optimizations you make for a task combination will more often yield benefits.

Efficient simulation is a trade-off between accuracy, observability, and simulation speed. As you increase one com-

MORE ON TOOLS

The worst-case execution path is a complex calculation, based on a definition of what you are measuring (cycles, resource usage, and others) and the frequency of the spectrum of critical paths. Overestimate your worst case, and you overallocate resources in your design or spend time optimizing a system you don't need to. One problem is that most simulation and profiling tools lack direct means for finding the critical path.

Counting cycles along a linear execution path doesn't show the whole picture. It doesn't include cycles that you lose to contention, and they're the wild cards, because latency varies from run to run. In most cases, you can collect task-execution times, but you have to add them together to figure out how long processing a data set takes. Doing so assumes that you can break the profiles for each task into times for each data set and that you can correlate such times across tasks—a tedious task to perform by hand.

A vendor may update tools before you release your final product. An improved compiler, however, could create significantly different code, seriously skewing the profiles you created to determine how to optimize code. If you use these tools—and you may have no choice—you have to go back and reprofile your system. To ease this process, be sure to document the profiling process, so you don't have to figure it out again. Also, leave some headroom in case you end up with slower code—that is, a tool that optimizes an operation based on code size and cycles at the expense of other operations. You might also leave most of your profiling work until you have final tools. Consider the following: The new compiler reduces a task's cycles from 95% to 75% usage—taking you from 5% overhead to a 25% cycle burn.

If an operating system manages scheduling, you face another level of abstraction and inefficiency. Be sure to strip the operating system to the bare functions you need to keep the inefficiency cost as low as possible. A single nice-but-unnecessary operating-system call could bring in additional libraries and overhead across all functions, costing

more in overhead than you'll save by using the call in only a few places.

Is the reference platform a “little-brother” version of the multicore device you plan to use—say, with only one of six cores? If so, you can profile only the ideal execution of tasks. Without a full system, you don't experience resource contention, which is responsible for most of the counterintuitive effects in a multicore system. No core is an island.

Being able to toggle between a GUI representation of your design and C code sounds useful. In most cases, the GUI representation will generate C. However, if you can't easily propagate changes in C back to the GUI model, you'll double the work for the sake of the abstraction.

Tools from Consystant and Teja allow you to model a program using a GUI. These tools convert the model to C, which a compiler then converts to assembly—a double abstraction with regard to potential inefficiency.

When you bring your own code into framework environments, you need a wrapper to describe the code to the framework and connect all the necessary hooks, adding yet another layer of abstraction and inefficiency.

Although more data points can improve your confidence in simulation results, simulation rates are slow. One vendor quoted two to five cycles per second for a gate-level simulation of 70 million gates and 1000 cycles per second for a functional simulation rate using abstracted models. At that rate, it takes more than 27 hours to simulate one second of operation for a 100-MHz device, and that time is for only one processor. Note as well that abstracted contention may not resemble real contention, so the accuracy of the functional simulation is suspect.

Profiling accurately is a difficult chore, because you must use trial and error on the full system. Working with only a subset of the system gives you information for that subset that doesn't necessarily scale or correlate with the full set of tasks.



ponent, you reduce the others. Given the processing speeds of devices in multiprocessor systems, it is generally unfeasible to collect overall system-trace data while hunting down a bug or creating a profile, because there is more data available than you can collect. Thus, even though you can more quickly exercise a system with hardware, you lose the ability to see the details of what's happening.

For single-processor systems, you can set a breakpoint and take a snapshot of the system. You might even be able to unobtrusively perform this task, so that it has no impact on system execution. If the processor is running at high speeds, unless you stop it, you can collect only so much data before more processing takes place and the snapshot changes.

In multiprocessor architectures, you may be unable to stop all processors on the same cycle. In this kind of breakpoint, called chaining, each processor passes the breakpoint on to the next processor. Thus, a processor stopped late in the chain may alter relevant data. Additionally, you may be unable to restart execution from the breakpoint and so must reset the system. You also face the issue of uploading tremendous amounts of data from so many processors. The ability to limit what you upload may save you significant time over the course of a day's debugging.

Some processor vendors targeting multiple-processor applications offer design tools beyond compiler and simulator. For example, Intel's Architectural Development Tool helps you budget and estimate memory bandwidth, pipeline performance, core efficiency, and overhead before you begin modeling and simulation. Most processors form the

core of their own simulation environment. If you are using different processors in a system, such as control and data (microprocessor unit plus DSP or NPU plus coprocessor), determine how much you'll need to massage the models to simulate processors together.

Several nonprocessor companies also offer multiprocessor-design tools. MLDesigner, from MLDesign, uses detailed or abstract models for system-performance modeling. Matlab and Simulink from The Mathworks are common platforms for DSP applications. And Axyx offers MaxSim, a tool set for modeling and verifying multicore systems on chips.

You'll soon be able to leave your compiler instead of just your simulator on all weekend. Tools arriving next year promise to iteratively recompile code based on profiles run on previous compilations, thus testing out multiple configurations. For example, the compiler could determine the critical code path, identify the most common variables and registers, and place dependent values contiguously in memory so that you can retrieve them with one memory look-up rather than two.

Note that, no matter how comprehensive your profiling tools are, the results are only as accurate and relevant as the completeness of your code.

THE TRUE COST OF ABSTRACTION

What is the true cost of abstraction? To determine it, you have to consider all of the levels of abstraction of which you might take advantage. Again, abstraction is not bad. Ease of design is worth introducing some inefficiency into your design. But if an abstraction causes you to puncture the sphere of ease of design,

FOR MORE INFORMATION...

For more information on products such as those discussed in this article, contact any of the following manufacturers directly, and please let them know you read about their products in *EDN*.

ACE

www.ace.nl

Analog Devices

www.adi.com

ARC

www.arc.com

Axyx Design Automation

www.axysdesign.com

Consyant

www.consyant.com

Cypress

www.cypress.com

Express Logic

www.expresslogic.com

Green Hills Software

www.ghs.com

IDT

www.idt.com

Intel

www.intel.com

LSI Logic

www.lsilogic.com

Mathworks

www.mathworks.com

MLDesigner Technologies

www.mldesigner.com/

Teja Technologies

www.teja.com

Tensilica

www.tensilica.com

Texas Instruments

www.ti.com

you might question whether it's costing you more than you'll gain.

Understand the trade-off that each abstraction entails. For example, a design environment may abstract resources that could physically be software or a hardware-accelerator engine, depending on your application. Thus, the final code could be a set of instructions, such as a macro; a single instruction; or the setting of a few parameters, such as a configurable resource.

Development tools may appear to handle the physical allocation of these resources, but they often do not. You might still need to arbitrarily define thread boundaries, how elements will communicate, how to share resources, and so on. In other words, you still have to manage all of the hard stuff. The development environment gives you a limited ability to profile how you define these characteristics, but it does not directly help you determine the best way to define them. You pay the full abstraction penalty to use the tool without gaining the full benefits possible from such an abstraction.

As the tools evolve, they bring new challenges to accurately determining the real profile of a design (see **sidebar** "More on tools"). However, each generation of tools more tightly bounds the limits of inefficiency. One day, you'll reach the point when you won't care about inefficiencies, because you'll be willing to accept them at their worst. Who knows? One day, the ultimate abstraction tool might emerge, which would allow marketing executives to design products directly in PowerPoint, bypassing the engineers altogether.

Most vendors admit, once you push them to the wall, that programming in C costs you a certain percentage of efficiency (see **sidebar** "What's the metric for 'easy to program'?" on the Web version of this article at www.edn.com). But programming in C is only one of the myriad abstractions that this article mentions (see **sidebar** "Other considerations" on the Web version of this article at www.edn.com). If the cumulative cost

of abstraction were a mere 10%, not choosing abstraction would be insane. However, it is unclear just how costly abstraction is, and the tools to accurately measure the cost are simply not available today.

The bottom line is that you can trade performance efficiency for the potential ease of design that comes through abstraction. You'll have your product up and running faster, but you'll also have to overprovide your design to accommodate all of the inefficiencies abstraction brings.

Ask yourself how much time to market costs you. And consider that 50% inefficiency in a six-processor system means you could optimize down to five or four processors if you're willing to make the effort. The question is not whether the abstraction trade-off is worth making. You need to look at the real cost of abstraction on a case-by-case basis. □



You can reach Contributing Technical Editor Nicholas Cravotta at e-mail editor@nicholascravotta.com.

OTHER CONSIDERATIONS

Many processors, even so-called proprietary ones, are based on cores from IP (intellectual-property) vendors. The Tensilica core, for example, forms the foundation for several NPUs (network-processing units). When selecting an NPU, find out which core, if any, the vendor based the device on. Although the device contains customized IP, chances are the development tools will be the same or similar for a different NPU based on the same core. Thus, if your NPU vendor goes out of business, you may be able to carry over a significant portion of code to a seemingly unrelated NPU and avoid learning a new tool chain.

Knowing the underlying core can also broaden your choices when respecifying a control-path processor if you need to decrease cost or increase performance. For example, the ARCTangent core from ARC supports ARM, MIPS, and PowerPC processors in its multiprocessor-debugging environment.

Is the zero-latency context switch also a zero-cycle switch? For example, when a thread accesses a resource from which it awaits a response, does the thread have to use a cycle to yield control? Although yielding may be only a one-cycle instruction, over an extreme number of context switches, these yields add up. For example, five memory accesses in 100 instructions require five yields, a total of 5%

of code executed for this task. Switching tasks using interrupts avoids yield losses but more than compensates by causing pipeline flushes.

One way to stress-test a system is to overload it until it breaks and then back off a certain percentage so that you have confidence that it will run under extreme circumstances. For example, you could create a task that exacerbates resource contention until you experience failure. If only 10% additional contention causes breakdown, then your system is reaching its limits. Given the extreme amount of time it takes to simulate even a second of real time for complex systems, consider stress tests that preload resource queues and simulate engine stalls then simulate corner-case situations.

Consider the impact of a resource's failing to respond within a "guaranteed" limit. For example, contention for memory or a search engine under extreme conditions you hadn't considered may cause latencies to exceed assumed limits. You need some kind of failsafe or failure mechanism to address these possibilities.

Beware of optimizing too heavily. For example, with an old incarnation of the x86, it took one fewer cycle to move a constant to the accumulator and then move the accumulator to a memory address, than it did to put a constant directly into mem-

ory. However, as the x86 family advanced, the accumulator advantage disappeared.

Suddenly, using the accumulator took an extra cycle as well as extra memory, and the advantage became a disadvantage. If you exploit such specialized nuances of a processor, you might consider using a macro that you can adjust if the processor changes.

When profiling a system, you need to consider the effect of the control and data planes on each other. Real-world events, such as table updates, can stall data throughput, and throwing too many exceptions to the control plane can cause bottlenecks. Unfortunately, the control and data planes are developed not only using independent environments, but also generally by different teams. Thus, you can't profile the impact of one plane on the other until hardware exists, a point too late in the design cycle to tolerate much adjustment.

Memory coherency also poses a particularly sticky profiling problem. If the data does not reside in two locations, one task considering changing only the data may lock out another task that wants to look at only the data. The effects of coherency could potentially push latency issues beyond cores merely sharing a resource to significant stalls as locked tasks wait for access for indeterminate periods of time.

Given that latency varies when you access a shared resource, such as memory, cores usually yield control and wait for a response. Some processors wait a set period before determining whether a response is available; others wait for the resource to push a response and set a flag or interrupt. The difficulty with polling for a response is that if you make the polling frequency too high, you burn cycles and bus bandwidth checking for responses that aren't ready. If the frequency is too short, you risk stalling the system while threads wait for responses that *are* ready.

Deciding whether an issue is a bottleneck or just noise requires testing—not just promises from a marketing rep. Ask for a copy of the study that marketing reps use to make such promises. Verify that in simplifying the problem, the researchers didn't unwittingly eliminate the problem under test or include an even more inefficient abstraction against which your problem pales.

Under some circumstances, some multiprocessor-design issues may have negligible impact on your system. You may be tempted to assume the relevancy of a particular abstraction cost based on experience with single-processor systems. This tack can be foolish, given many of the counterintuitive effects of multiprocessor architectures.

WHAT'S THE METRIC FOR "EASY TO PROGRAM"?

Is writing optimized assembly that much more difficult than using abstract C? When you code in assembly, you exploit common data structures and functions. For example, if you want to add two complex numbers, you develop a data type and function, both of which become part of your personal abstract tool set. Companies that offer framework tools outline these data types and functions for you; you program in C but use their assembly-reducible framework.

Developing efficient frameworks is time-consuming, but you create the framework only once. A framework you create yourself is less abstract than a general framework that has to accommodate myriad application variants, so it gives you greater visibility into your design. As you continue to uncover more patterns and structures, you can refine (optimize) the framework. You can also spin off variants from a vendor's framework, but you'll have to start at a comparatively higher level of inefficiency.

Framework tools don't define the framework but rather assist you in developing one. Here's a favorite vendor "oversight": Vendors measure the time to code without their abstract framework tools from the description of the problem and the creation of data and process structures to complete coding;

they measure timing for the abstraction tool from the creation of the structures to the creation of abstracted code. What's the difference? Abstraction time doesn't include the time to describe the problem and create structures, a time-consuming part of design. In other words, they already knew what structures to look for when they began work using the framework tools.

You have only the word of the vendor that its tools create code that is almost as efficient as code from a human programmer. There's a good chance that the same programmers who wrote the tools—not an engineer who works day in and day out with the algorithms—wrote the handcrafted code they use as a baseline. Often, someone years ago crafted the handwritten code; it's no surprise that the compiler eventually catches up to static code against which it is benchmarked.

Validate efficiency. Create a test suite exercising compiler and a simulator and compare it with your own handwritten code.

Intentionally slam resources and compare how the compiler extracts parallelism and exploits nondependencies. You pay a stiff price in loss of control and ability to optimize when you abstract programming, especially when you use frameworks. Make sure the gains are worth it.