

SOFTWARE:

the Achilles' heel of network processors?

PROGRAMMING NETWORK PROCESSORS and coprocessors is a difficult task. Many companies admit that their previous-generation parts were too difficult to program and are investing heavily in creating better development tools. The challenge is daunting for sever-

al reasons. First, the applications themselves are complex, and their design requires great ingenuity to develop techniques to process packets at wire speed. Another challenge is that the gamut of devices differ enough from each other to allow no coherent classification; for example, every "forwarding engine" assumes a different level of function and offers different means for offloading tasks to other devices to accelerate processing. Also, getting these devices to communicate and make full use of each other's unique features can take weeks to months. And, as much as the data plane (wire-speed processing) and control plane (system-level management) are independent, they must work together intimately.

One of the primary bottlenecks to design is learning how to use the various devices available. Each device or software suite takes a different approach to network pro-

cessing, offering unique methods for solving certain portions of the problem. The programmability of many network processors and coprocessors lets you build on these methods to also create your own methods for approaching these problems. This situation means that software plays an increasingly more important role.

Development tools and off-the-shelf software are key differentiators of ICs. As the race continues toward simplifying the programming of these devices, many vendors have come to rely on C as a way to avoid the complexities of assembly or microcode by abstracting the problem of understanding the internal workings of a device or the complexities of having two devices communicate with each other. And all vendors are turning more of their R&D dollars toward developing tools beyond assemblers and debuggers.

For much more on this topic, go to the Web version of this article at www.ednmag.com.

THE DIFFICULTY OF PROGRAMMING NETWORK PROCESSORS AND COPROCESSORS HAS FOR THE PAST FEW YEARS TRIPPED UP THE SUCCESS OF THESE DEVICES. HAS ANYTHING SIGNIFICANT CHANGED, OR IS THIS MARKET FATED TO CONTINUE TO ONLY SLOWLY STUMBLE FORWARD?

At a glance34

Proprietary babble34

Customizing code: a serious prospect36

Marketing mistruths and other lies38

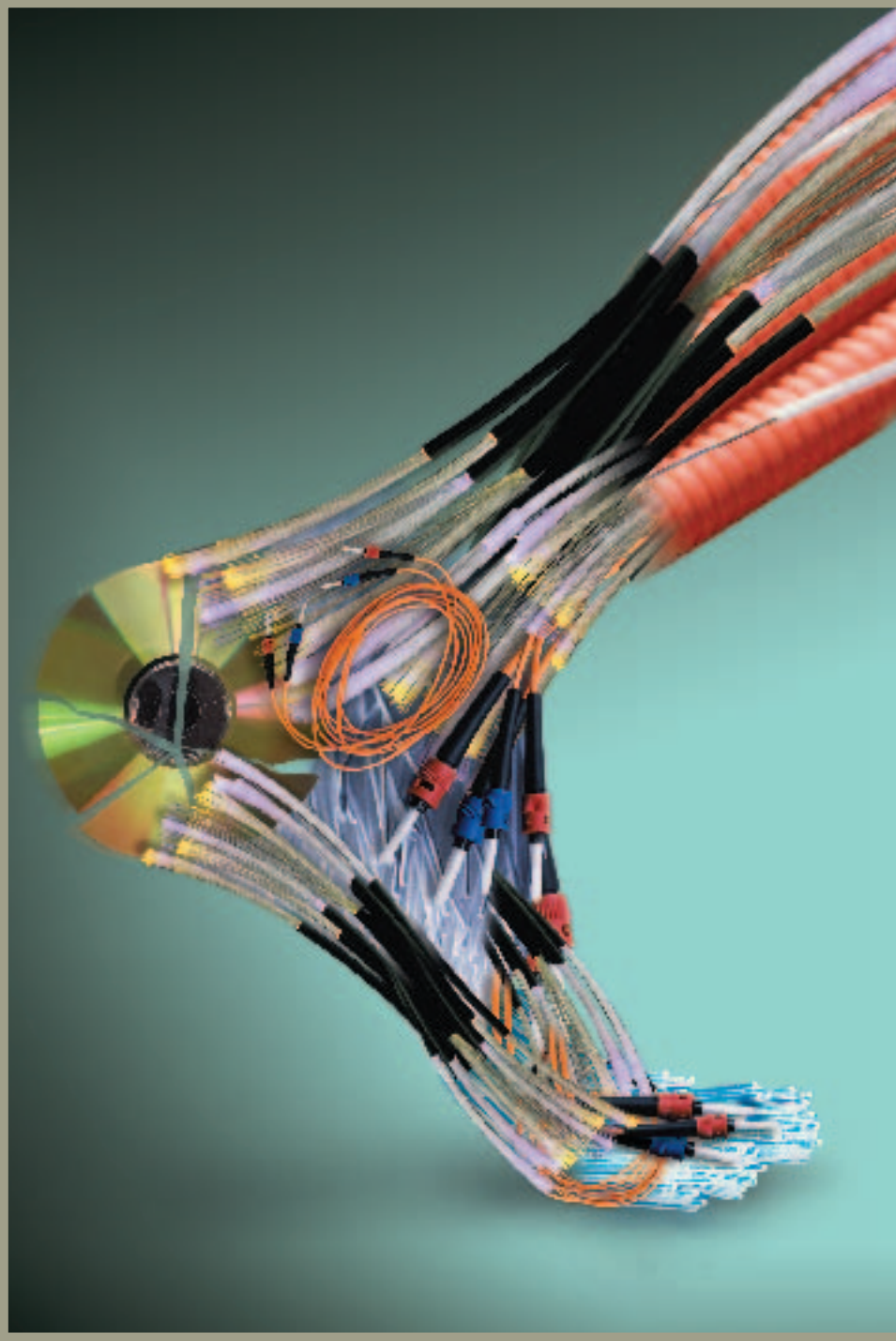
For more information.....44

Illustration by Mike O'Leary

Those vendors that offer C tools claim that developing in C is significantly easier than developing in assembly or microcode. They claim that their compilers create code that is on par with handwritten code. They also claim that those programmers familiar with C will somehow have an advantage over those programmers who don't.

The reality is that few of the vendors actually support C. First, the C they are citing has been stripped of huge blocks of features, such as floating-point instructions or pointers. Internally, the structure is also different. For example, functions are often resolved as inline code. Next, the C may offer a superset of specialized instructions that optimize certain tasks. These instructions may be intrinsics, pragmas, extensions, or macros that give you "direct" access to the hardware architecture or function calls to inline assembly blocks or direct commands to a hardware accelerator. At this point, you have to consider whether this C is a simple extension of C or more of a C**, a new language only somewhat related to C.

Some vendors pitch their versions of C** as languages with "guardrails" that won't let you do things you can't or shouldn't do. However, to understand how to avoid scraping these guardrails when you want your code to scream, you have to understand the programming model the assembly of the network processor expects. Note that this is not the programming model the C compiler expects. If the network processor really expects assembly and wasn't designed to support C-types of commands, the compiler forces you to program in





a manner that results in less efficient code. The underlying hardware with its specialized acceleration engines may require a unique method of programming to take full advantage of these engines. If you use intrinsics to access these accelerators, then you're not using C anyway and have probably added a layer of abstraction and inefficiency to your code by not directly using inline assembly.

Using C does not reduce the complexity of the network-processing problem. You can give the task a C syntax, but this approach doesn't reduce the internal complexity. Using C doesn't mean you don't have to understand what you're doing. For example, table-look-up algorithms depend upon the type of hardware acceleration available. Some coprocessors offer specialized functions that go beyond mere look-up and address issues, such as lowering power consumption by hashing look-up keys and segmenting large tables. Accessing these engines to their fullest capacity requires access to the engines themselves and understanding how they interact with the rest of the system. Using C cannot shield you from having to learn such details, but it can hinder you from taking full advantage of them. You may even go so far as to try to trick the compiler into doing things you want to do but don't know that you shouldn't be doing with the network processor. From another perspective, consider the differences between

AT A GLANCE

- ▶ The primary design bottleneck is learning to use the network-processor/coprocessors' tools, software, and reference code.
- ▶ Abstraction simplifies the programming model but comes at a cost in efficiency.
- ▶ Programming in C may not be such a crazy idea; you can trade off code efficiency for the opportunity to evaluate additional hardware devices.

Java and C. On the surface, the two languages look similar. Yet, what is good programming in C is not necessarily good programming in Java. C allows the use of pointers, which are fundamental data-management tools. Java, on the other hand, has garbage collection, which colors many design decisions. In some cases, the fact that you know a similar language, C, may actually increase the time it takes you to learn a variant of the language, C**, as you struggle to “unlearn” common structures in C that are cumbersome in the world of C**.

C doesn't easily handle, for example, the basic task of bit manipulation. If you want to do a multi-tuple look-up, you need to extract fields from different parts of a packet to create a single look-up key. Likewise, the result of the look-up may contain several results that you need to

extract and store. In such a case, C is not an optimal language. Pulling bits takes about as many lines of code as assembly would, and, because of the abstraction from the underlying hardware, you cannot easily take advantage of bit-manipulation engines. A representative from one vendor claims that bit manipulation is straightforward and simple in C, stating “testing the 15th bit in the second word of an IP header...would compile to no more than five machine instructions.” Another vendor talks about using string-copy instructions to rip bits from a tag and compress them. Is this a good approach for programming or for execution?

Bit manipulation is a common enough task to warrant some kind of abstraction. Some vendors provide this abstraction via templates with header constants defined for particular protocols. Some have created microcode libraries optimizing protocol-specific functions, such as stripping fields to create a look-up key. Others offer C functions or pragmas that provide inline assembly for standard, common, protocol-specific functions using on-chip acceleration engines. However, if you want to do something nonstandard, such as creating a hash key for accelerating table look-up, you'll have to drill down to the assembly yourself. Some proprietary languages recognize the frequency of such operations by offering an instruction that generates look-

PROPRIETARY BABBLE

Some network processors support proprietary languages, which may also be specific to a task, such as classification. A vendor may claim that one line of such a language can represent tens of lines of C and hundreds of lines of assembly. These claims may not tell the entire story. Such a language necessarily limits what you can do. For example, how many ways can you configure a single line of a high-level language versus hundreds of lines of assembly? Consider the following analogy: Even with different tenses, suffixes, and prefixes, the amount of detail you can impart with a single word cannot match

the variety of expressions you could impart with 100 letters.

The question to ask is how much detail do you need to solve a task? Some packet-processing tasks, for example, are so well-defined and limited in scope that gross instructions may be sufficient and actually simplify the programming task. On the other hand, the simplicity of the language may be an illusion. Each instruction may have so many parameters and options that it takes you longer to write a single line of code than it would to have written the equivalent C code. Some instructions are more like fixed functions with long parameter lists,

which are fine for a well-defined task but a barrier if you have a unique implementation for an algorithm and lack the necessary detail to outline it. Interestingly enough, these vendors still talk about how designers will want to optimize their code. Yet, with such limits of language, how creative and clever can your algorithms be? Then again, just how clever do they need to be? The specialized language should implicitly promote the creation of optimized code if it was designed to be intimate with the hardware.

In some respects, it's unclear why the vendors think that programming in assembly is so diffi-

cult. Some of the languages and C implementations the vendors tout as panaceas to programming seem less than simple. One purpose of higher level languages is to promote readability and understanding of code. However, the syntax necessary to cram enough parameters into a single line of code to accomplish the work of 100 lines of assembly requires abbreviations and tight spacing to create a wholly unreadable listing. Three lines of code would be easier to read. In some respects, the one-line metric—how many functions you can cram in a single line of code—is a race toward incomprehensibility.



up keys; its arguments are the positions of fields in the packet and the registers you want to store the results in. Such syntax allows you to create a key in one line of code and efficiently uses acceleration engines (see sidebar “Proprietary babble”). Note that each of these approaches may not long remain a competitive advantage because other vendors will carry these advances to their next generation of tools, as well.

The underlying fallacy of the C-language argument lies in that vendors claim that, as programmers move to a high-level language, the skill of the programmer has less impact on code than

at the assembly level; that is, C evens the playing field. To some degree, abstraction frees programmers from understanding what is happening “under the hood” and enables them to focus on what they want the network processor to do. However, most designers didn’t build their network processors with C in mind, and bridging a general-purpose language, such as C, to a highly specialized assembly is bound to result in inefficiencies. This restriction means that when you’re trying to squeeze another few cycles from an algorithm, you’ll find it more difficult to figure out how to pull bits out in 24 steps instead of 27 if you don’t understand what the

network processor can and is doing. If the vendor offers no access or fails to maintain its assembly-level tools, you may be unable to recapture this lost efficiency.

C OR ASSEMBLY: A QUESTION OF MATURITY

Despite these limitations, one can say a lot in favor of programming in C. Code optimized for a device can suddenly become less so when the vendor releases the next generation. For example, one of the old x86 variants took one less clock cycle to execute storage through the accumulator rather than directly through a register. When Intel released the next-

CUSTOMIZING CODE: A SERIOUS PROSPECT

Altering code erodes the code’s portability in several ways. Redesigning code to take advantage of particular hardware optimizations means that you cannot change hardware without unmaking and then remaking these changes. It also reduces your ability to use the vendor’s next release of code. And don’t forget those 800-pg manuals and thousands of lines of code you’ll need to sift through and understand.

Given that you will most likely customize any code you receive, you should understand the challenges of managing and tracking such customizations. The vendor will probably offer a revised update of the code you are using and have customized. You can ignore these revisions if you don’t want them but not if they are revisions to fix errors. To adopt revisions, you have to either carry over customizations changes to the revision or carry the changes in the revisions to the customized code. If the code is fairly stable, many teams choose to “freeze” the version of vendor code they use, taking full ownership for changes and maintenance.

Well-designed code takes a modular approach that effectively separates stable sections of code from code that continues to mutate, resulting in revisions

that clump in expected modules of code, leaving the rest of the code intact. One way to protect stable code from later optimizations and revisions is through APIs. The tasks you want to perform, such as a table look-up, don’t change, but some underlying fundamental function changes. Such modularity can define the line where vendor and engineer change code: The vendor controls the code above the API, and the engineer controls the code below it. Such modularity can also allow the extension of new functions and protocols without affecting the code base. Thus, the structure of code plays a significant role in how difficult it will be to carry custom changes through generations of code.

Nearly every customer wants to customize code, so most vendors supply source code, even if they would rather not. Ask to see a previous generation of code and how the vendor distributed the revisions during the upgrade. Does the vendor expect you to compare the different revisions, or does the company clearly identify and document the changes? Also, check how often and for what reasons the vendor has revised code. Enhancements, however nice, can be a nightmare to reintegrate with customized code.

Implementing revisions can temporarily halt code development as you validate the changes. If you carry your customizations to the revised code, clearly track your changes, because you may be repeating the process the next time the vendor issues an upgrade. Also, if you ported the code to an operating system of your choice, you may have to port part of the revisions, as well.

Also, consider whether the network-processor vendor, coprocessor vendor, or a third party wrote the reference code. Of those, which has the incentive to ensure that it works? Do the pieces fit well into a modular architecture, or did the vendor hastily stitch them together? In general, the more defined the application, the more focused the code, and the more likely it is that you will be able to use less of it, depending on how much your application differs from the reference design.

Choosing whether to change code in the middle of an important block of core code or in an abstracted low-level function requires an intimate understanding of the design and structure of the code base. Abstraction of the code facilitates this process, but it adds layers of complexity to the code and reduces the code’s execution efficiency.

No network-processor vendor now offers tools for tracking and migrating customized code to revised code. Some vendors offer source-control tools, many available as freeware, for tracking a team’s work on a large code base. But these tools are available only for that one team. Remember that two teams are working on code. One team, the vendor, has no knowledge of and little regard for the changes that the second team, yours, makes. Some vendors claim that tracking changes is simple; others admit that it isn’t. Much depends on how deeply you expect to customize the code. Bottom line: If you want to make custom changes—and you will—you have to manage the process.

Carefully consider what customizations you make. For example, if the vendor plans to offer software in a future revision covering a function you want to add, you shouldn’t spend a lot of time developing and optimizing this function when you could focus your efforts elsewhere. However, you could license your changes or ports to the vendor. You may to some degree be helping your competition, but you also have a hand in what you consider a key section of code and will be able to somewhat direct its course.



MARKETING MISTRUTHS AND OTHER LIES

Many network-processor vendors claim that abstracting code—that is, using C instead of microcode—saves a designer from having to understand the underlying hardware architecture. In almost every case, this claim is patently absurd.

Consider the issue of resource contention. In a multiprocessor or multithread architecture, two executing tasks often want to use the same resource, such as the memory, the coprocessor, or the on-chip accelerator. Resolving such contentions requires buffering and arbitration. One task gets priority over the others, and the others have to wait for the resource to free up. The processor can execute code that does not depend on the interaction with the resource during waiting, but most programming is linear, and nonlinear processing can go only so far. Chances are, you can't avoid a stall.

Deterministic environments have a maximum time in which the processor can grant access to a resource. The larger the maximum time, however, the less processing that can occur at this stage. Only a designer with full knowledge of the architecture can determine this maximum time. Increase the complexity of the possible contention, and you also increase the complexity of determining the maximum latency.

No clear answer exists for how multithread/multiprocessor architectures can efficiently resolve resource contention. Several packets in a pipeline, while at different stages of processing, may each require access to a look-up engine. Perhaps an arbitrator buffers and resolves such conflicts. However, without understanding how these stages impact each other, a designer may inadvertently design a system that calls upon a resource in bursts as opposed to spread out over time, thus creating an

unnecessary system bottleneck.

There's also the problem of nondeterministic resources such as encryption engines. Loads to an encryption blade are difficult to predict, and a thread could be seriously delayed based on traffic in another part of the system altogether. The worst-case for encryption can be so bad that you may find it better to throw off such packets as exceptions.

A compiler or a hardware arbitrator can resolve resource-contention issues, but how efficiently can they do so? There's no simple benchmark; efficiency depends on the designers' code and coding style.

Another common marketing untruth is the quality of compiled code. If you press them, most vendors admit that you will want to hand-optimize compiled code because the compiler often does not understand critical system issues that affect optimization. Remember, too, that the vendor may be giving the compiler an advantage in that it can easily perform certain tricks, such as loop unrolling, performing inline instructions, executing code out of order, and collapsing contiguous memory accesses into a one access, or collocation, of data. The handwritten code they compare the compiled code against may not have taken advantage of any such optimizations. Also, compiled code sometimes takes advantage of previously hand-coded blocks of code for specific functions. The vendor demo code will follow form enough for the compile to recognize such blocks and substitute the optimized code. However, will the code you write accomplish this?

Several vendors claim that you can write the code for IPv4 (Internet Protocol Version 4) routing in 10 minutes. This claim makes no sense. It takes more than 10 minutes to figure out

what you want to design, more than 10 minutes to load the software, more than 10 minutes to learn to use the tools. These claims suggest that the vendor doesn't understand the complexity of these designs. Few designers would use such code anyway; you don't have much of a design if you could do it all in a few minutes or a few days. It pays to carefully probe such vendors' software claims and understand clearly what it will take for you to alter such software so that it can be useful to you.

Another marketing claim that is hard to verify unless you've committed to undertaking a serious evaluation of the hardware is the quality of multiprocessor/multithread compilers. The DSP industry, which has for years used multiprocessor/multithreaded designs, has yet to develop excellent tools for handling debugging, profiling, and optimizing such designs. The network-processing industry and its tools are relatively new. These problems are difficult to solve. Even the control path is starting to go multiprocessor as data rates increase and single processors provide insufficient control management.

Various multiprocessor/multithreaded network processors take different approaches to partitioning processing. Two common methods are dedicating processors/threads to specific pieces of packet processing and having each processor/thread handle the entire processing of a packet. In the first case, breaking processing into pieces adds the complexity of having to stitch the pieces back together and manage the data flow between processors/threads. The internal communication architecture is like a tiny network in itself, with buffers and flow control. Such an architecture also needs to be able to change

dynamically, based on actual traffic patterns. How do you account for such factors during compilation?

In the second case, the processors/threads act independently and create resource contention, depending on how different types and sizes of packets happen to fall. Packet processing is an asynchronous operation, and various processors/threads quickly fall out of synch with each other. A compiler may be able to handle contention issues for well-understood operations, but high-speed operations working with a variety of protocols with different levels of servicing and processing become difficult to define as predictable and, hence, understandable. It is a fundamental truth of network traffic that it is neither deterministic nor predictable.

Using multiple chips adds complexity, meaning that you need to share flow-control/backpressure information and resolve resource contention. A network processor and coprocessor using a shared memory can significantly reduce the effective memory bandwidth. Using multiple chips or even multiple dies in a multichip package adds additional potential system bottlenecks. Cache coherency becomes an issue for chips that share cache: When one component is using a part of memory, another chip may be unable to access that part until the part is free, causing latencies.

Nevertheless, many of the tools vendors offer are leading-edge products. However, they may not necessarily be exactly what you need. Optimization still requires a human perspective that can consider all the factors you can't describe to a compiler. As good as the tools are, this still isn't a task for inexperienced programmers.



generation chip, the role of the accumulator changed internally, and the position reversed: The direct code worked faster.

Getting too involved with the underlying architecture presents a temptation to many engineers to build code optimized for that architecture. After all, every engineer knows that a human coder is more efficient than a compiler. But you have to approach this situation from a system-level perspective. Is reducing a core code loop more efficient if it takes a programmer a week to do so? A week is more time than what many coprocessor vendors claim is necessary to construct a *shim* layer, which adapts the abstracted code to the available hardware without modifying the majority of the abstracted code. The trade-off takes a wider perspective than merely how fast code executes. Which is more important: running a block of code a few instructions faster or evaluating the performance of an additional coprocessor?

Using a higher level language that shields you from the details lets you more quickly sketch out an architecture. Again, using C sometimes results in significant inefficiencies. But these inefficiencies have a lower cost if you still have available overhead in the network processor. The trade-off becomes the ability to create code that leaves room for future products or testing several divergence architectures and profiling system-level inefficiencies to evaluate which overall approach will serve best in the long term (see sidebar “Migration patterns” on the Web version of this article at www.ednmag.com). As long as the network-processor vendor has an available assembler, you can optimize key sections of code when you have time to take care of details. Regarding overhead, optimizing core code between generations of products frees up some headroom. Also, don’t underestimate the value of next-generation devices (see sidebar “Silent but deadly” on the Web version of this article at www.ednmag.com). In the time it takes you to optimize your code, the vendor may have released a chip that offers enough of a performance gain to obviate the optimization. Of course, you have to have faith in a vendor’s road map to adopt this course (see sidebar “How well do you know your vendor?” on the Web version of this article at www.ednmag.com). The goal to keep in mind is to create a finished product. Great code in a product that isn’t finished before the

R&D dollars run out is worth nothing (see sidebar “What’s the real cost?” on the Web).

ABSTRACTING APPLES AND ORANGES AS FRUIT

Two years ago, many network-processor vendors loudly proclaimed that programming in their network processors’ microcode was easy. Today, they are just as loudly proclaiming today vendors that expect you to code in microcode are making things difficult for you. All this marketing hype hides a truth: The single biggest step vendors can take to accelerate software development is to simplify the programming model (see sidebar “Some questions about software” on the Web version of this article at www.ednmag.com).

One popular method of simplifying programming is to create software abstracts of functions that map to various hardware elements. Thus, the programmer focuses on the application and avoids the perplexing details of hardware implementation. The general idea is that you can reduce any application to some significantly smaller number of core functions, much in the same way that the core instructions of a programming language form the basis of complex applications. For example, Internet Protocol Version 4 routing can be reduced to approximately 30 fundamental functions. One such function might modify a table entry in a forwarding table. Above this subset, the software need not understand how you modified a table—only that you can modify it. To port this application, you need to supply shim code, the code necessary to implement the core functions on the hardware platform. Some network processors supply code or libraries, and APIs access their functions. In such cases, shim code would bridge the two APIs, and you would have to write shim code for both the control- and the data-plane processors. Add a coprocessor, and you also add a shim. You may also need to build a shim for the operating system (see sidebar “Open-source operating systems on the Web version of this article at www.ednmag.com). This approach may leave you with several shim layers to create.

The admitted complexity of shim layers varies from vendor to vendor. Some claim that you can port to their APIs in a week. Others suggest allocating three to six months for the process. The longer figures are probably more accurate in

that they consider such issues as the complexity of the other API you are creating the shim for; the learning curve for the tools and software; how much code in kilobytes versus megabytes you have to port; how long it takes to sort through the code and understand all the relevant dependencies, including both the data- and the control-plane shim to the operating system; and testing and validating the shim to the point that you consider it not only working, but also robust. If the vendor still claims that you can write the shim in a week, ask the marketers why the company’s engineers haven’t yet written it, or make writing the shim part of your purchase deal (see sidebar “Anatomy of a third-party network-processing-software vendor” on the Web version of this article at www.ednmag.com).

Some vendors offer “device-independent” APIs, meaning that you can use such software regardless of the hardware implementation you choose. This software to some degree shifts the onus of compatibility from software to hardware. Traditionally, software developers have written software to match the hardware. Now, you have to make the hardware match the software. To interface the control and data planes, this matching may mean that you may have to write a shim not on the control plane but on the data plane, where processing is most costly. If the shim is relatively thin (meaning not complex or process-intensive), this extra cost is negligible.

Device independence also challenges the premise that network processing is a system-level problem. Network processors and coprocessors as a rule offer features that differentiate them from other devices. To abstract functions means to limit the visibility of application code to capitalize on these unique features. Some functions, such as “search table,” remain relatively constant across implementations. “Relatively” is the key word, however. The special features of each coprocessor can make all the difference in table look-up and management, often key differentiators among products. Some vendors address this issue by writing blocks of modular code and providing the various network processor- and coprocessor-specific shims as well as a wider API that can expose such features in the modular code. Such generalization, however, may add layers of inefficiency unless you can configure the mod-



ular code for this purpose. Also, look at the internal partitioning within the software to see how its designers tried to solve the problem; the paradigms they selected to some degree define the limits of your system-level performance.

On the network-processor side, many vendors have tried to simplify the programming model in various ways (see sidebar “The Network Processor Forum” on the Web version of this article at www.ednmag.com). Several of the multi-processor/multithreaded network processors have single-threaded programming models. In other words, you write your code as if your design had only one thread and one processor. The compiler and network processor take care of everything else for you. What sacrifice in performance do you make for this simplicity? There are many internal constraints, including access to memory, to coprocessors, and to internal buses, within a network processor. If the programmer has no idea what these constraints are, the code could challenge a compiler to optimally allocate these resources (see sidebar “Marketing untruths and other lies”).

At the onset of a design, it may be unclear which functions should run on which piece of hardware. Given the variety of coprocessors and kind and amount of processing each does, partitioning functions is difficult and often an after-the-fact experiment. An onboard coprocessor can process security functions, for example, inline. In this approach, all packets pass through an encryption engine before hitting the network processor. Alternatively, the coprocessor could pass these functions to a dedicated encryption “blade,” or card. It’s also unclear how much of say, SSL (Secure Sockets Layer) processing should take place in the coprocessor, the control processor, and the network processor. Unfortunately, few tools exist for making system-level evaluations without actually requiring you to design the system. Magnitudes-of-order difference can result from system-level structural changes, so evaluating architectures becomes an intensive process.

By abstracting code, you retain some freedom in how to partition functions, especially if the API has several layers that you can peel back to the level you wish to work at (see sidebar “Customizing code: a serious prospect”). Thus, you

have the choice of moving some control code to the data plane. This task is especially important depending on how you implement the data plane and what coprocessors and features each of these offers. In other words, if you spec a device that accelerates a function, does the API become a logical stranglehold preventing you from partitioning that function to the device that should handle it, rather than the device that the software architects decided should handle it?

DEVELOPMENT TOOLS

Few vendors still offer solely an assembler/debugger combo. Most vendors offer a development environment with a compiler, a debugger, a simulator, a profiler, and a traffic generator with a variety of reference designs from which to launch designs (see sidebars, “Simulator tools,” “Profiler tools,” “Traffic-generator tools,” and “Some reference designs are more equal than others” on the Web version of this article at www.ednmag.com). Those chips that aren’t programmable come with the appropriate “configuration” tools to set the device up for an application. A few vendors offer frameworks, encompassing the traditional design environment. Finally, a short list of vendors offers tools that abstract the entire design cycle away from the hardware, topped with code-generation tools.

Tools at this highest level of abstraction map abstractions of functions to hardware or software. During initial modeling, you have the option of trying different mappings to test the efficiency of different hardware and software partitioning, as well as hardware devices. You can develop generic functions and later define the abstraction down to lower levels, such as modeling scheduling across multiple processors, threads, or both; shared-memory resources; and synchronization of elements, to name a few. A back-end compiler/assembler generates code for network processors and coprocessors, building a custom implementation from a generic description. How useful this code is depends upon the resolution with which you define your mapping and how cooperative the processor and coprocessor vendors are in supporting the mapping tool; that is, someone has to write the code or back-end compiler/assembler.

The difference between a framework and a development environment is that

the framework is targeted for a specific application. Thus, the framework “understands”, to some degree, that you are not just developing a network processing system, but that you’re developing, say, an IPv4 system. Common aspects and concepts of the application are blended into the environment. Frameworks often provide a skeleton system, which directs or forms the basis for development, guiding the developer along a known course.

Even if a vendor offers several devices and a development environment that “seamlessly” integrates them, you may want only one part from the vendor and the ability to mix and match it with devices from other vendors. You should be able to use only those parts of the tool suite that apply to the device you choose. For the tools to be most useful, you want to be able to link the tools for the other devices without having to fight with scripts. It’s worth checking to see whether can you add extensions to an environment. For example, you may want to analyze data in a manner that the tool doesn’t currently support. Finally, let the vendor show you some of the extra features that make its tools a cut above the competitors’ (see sidebar “Bells and whistles” on the Web version of this article at www.ednmag.com).

Of course, the most important characteristic of a tool is how it helps you develop your design. High-level abstraction tools may not be useful to you if you’ve already decided upon your hardware architecture and have a code base to carry over. It could be more work figuring out how to abstract all these elements so that you can map them back to themselves than to simply develop more code.

Developing code for network processors today differs greatly from the process it was even two years ago. Since then, the IC vendors have realized that simply having amazing hardware is too little to go to market with. Difficult-to-use tools and the inefficiencies of poor abstraction threatened to make writing software the bane of network processors. Has much changed?

The answer is yes, on many levels. The tools and software have become surprisingly better, given the short time vendors have had to develop them. Is the change enough? That answer depends on your application and how much performance you need to squeeze from a grain of sand. You can’t evaluate a network proces-



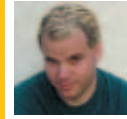
sor/coprocessor and its tool set by watching a demo. The only way to find out is to get your hands dirty with the tools.

Will the network-processing market fade away if the existing tools aren't enough? Too many companies have spent too much money to accept this

possibility. If sufficient tools don't yet exist, then ASICs will continue to dominate the market for another year or so until they do arrive. The ability to spin new product lines without spinning new ASICs is simply too appealing a proposition. □

(EDN is updating its Network Processing Web resource, available online at www.ednmag.com under Technical Resources.)

You can reach Technical Editor Nicholas Cravotta at 1-510-558-8906, fax 1-510-558-8914, e-mail ednick@pacbell.net.



FOR MORE INFORMATION...

For more information on products such as those discussed in this article, go to www.ednmag.com/info and enter the reader service number. When you contact any of the following manufacturers directly, please let them know you read about their products in *EDN*.

Agere Systems

1-800-372-2447
www.agere.com
Enter No. 352

Arc Cores Inc

1-408-437-3400
www.arc.com
Enter No. 353

Azanda Network Devices

1-408-720-3100
www.azanda.com
Enter No. 354

Bay Microsystems

1-408-653-2181
www.baymicrosystems.com
Enter No. 355

Broadcom Corp (SiByte)

1-949-450-8700
www.broadcom.com
Enter No. 356

ClearSpeed Technology (Pixelfusion)

+44 0117 317 2000
www.clearspeed.com
Enter No. 357

Clearwater Networks (XStream Logic)

1-408-376-1500
www.xstreamlogic.com
Enter No. 358

Cognigine

1-510-743-4900
www.cognigine.com
Enter No. 359

Consystant

1-425-739-9927
www.consystant.com
Enter No. 360

Cypress Semiconductor (Lara Networks)

1-408-943-2600
www.cypress.com
Enter No. 361

Effnet Inc

1-650-390-8700
www.effnet.com
Enter No. 362

EZchip Technologies

1-408-879-7355
www.ezchip.com
Enter No. 363

Fast-Chip

1-408-523-8050
www.fast-chip.com
Enter No. 364

GlobespanVirata

1-888-855-4562
www.globespanvirata.com
Enter No. 365

Hifn

1-408-399-3500
www.hifn.com
Enter No. 366

IBM

1-650-694-3007
www.chips.ibm.com/products/wired
Enter No. 367

Improv Systems

1-978-927-0555
www.improvsys.com
Enter No. 368

Intel Corp

1-408-765-8080
www.intel.com
Enter No. 369

Internet Machines

1-818-575-2100
www.internetmachines.com
Enter No. 370

IP Infusion

1-408-794-1500
www.ipinfusion.com
Enter No. 371

Kawasaki LSI

1-408-570-0555
www.ksli.com
Enter No. 372

Lexra Inc

1-408-573-1890
www.lexra.com
Enter No. 373

LSI Logic

1-866-574-5741
www.lsilogic.com
Enter No. 374

LVL7

1-919-865-2700
www.lvl7.com
Enter No. 375

Marvell

1-408-222-2500
www.marvell.com
Enter No. 376

Micron Technology

1-208-368-4400
www.micron.com/tcam
Enter No. 377

Mindspeed Technologies

1-949-579-3000
www.mindspeed.com
Enter No. 378

AMCC Networks

1-408-731-1600
www.mmcnetworks.com
Enter No. 379

Mosaid Technologies

1-519-599-9539
www.mosaid.com
Enter No. 380

Motorola

1-800-521-6274
www.motorola.com
Enter No. 381

NEC Electronics

1-408-588-6000
www.necel.com
Enter No. 382

NetLogic Microsystems Inc

1-650-961-6676
www.netlogicmicro.com
Enter No. 383

NetPlane Systems Inc

1-781-329-3200
www.netplane.com
Enter No. 384

Paxonet Communications

1-510-770-2277
www.paxonet.com
Enter No. 385

PMC-Sierra

1-604-415-6000
www.pmc-sierra.com
Enter No. 386

RadiSys Corp

1-503-615-1100
www.radisys.com
Enter No. 387

Radlan

1-408-996-2121
www.radlan.com
Enter No. 388

SiberCore Technologies

1-613-271-8100
www.sibercore.com
Enter No. 389

Silicon Access Networks

1-408-545-1100
www.siliconaccess.com
Enter No. 390

Solidum Systems Corp

1-613-724-6004
www.solidum.com
Enter No. 391

Teja Technologies

1-408-288-2560
www.teja.com
Enter No. 392

Tensilica

1-408-986-8000
www.tensilica.com
Enter No. 393

Terago

1-408-941-9664
www.terago.com
Enter No. 394

TranSwitch Corp

1-203-929-8810
www.transwitch.com
Enter No. 395

Vitesse Semiconductor Corp

1-800-848-3773
www.vitesse.com
Enter No. 396

Wind River

1-510-748-4100
www.windriver.com
Enter No. 397

Xelerated

+46 8 506 257 00
www.xelerated.com
Enter No. 398

Zettacom

1-408-869-7000
www.zettacom.com
Enter No. 399

RESOURCES

Network Processing Forum

www.npforum.org
EDN Network Processing
www.ednmag.com

SUPER INFO NUMBER

For more information on the products available from all of the vendors listed in this box, enter no. 400 at www.ednmag.com/info.

MIGRATION PATTERNS

An important consideration is how well a development environment handles migration of code and parallel variants. In effect, your first design may be a basic product from which you want to spin off variations for different markets. For example, you can modify port configurations, the number of cards allowed, and other features.

How well does the development environment support your need to manage a single base design and migrate changes across all the variations?

Several vendors are trying to target a range of markets from OC-12 to OC-192. You may be able to reuse much of the same code across such diverse prod-

uct lines if they have a common instruction set. In this case, C has a distinct advantage over assembly or microcode.

Maintaining development at an abstract level reduces the need to learn a new architecture and new tools and carries code to next-generation devices and across different markets.

In the upcoming generation of devices, some vendors are adopting new architectures to realize performance advantages. These architectures will result in the loss of the code base you are currently developing. Even if the architecture is much the same, new features will affect your design. For example, an

additional acceleration engine could replace some of your optimized code; perhaps it is worth focusing your optimization efforts on blocks of code that won't receive a hardware boost. Also, your optimization choices will change as the internal structure of the device changes—say, with additional interprocessor communication buses or increased memory access. You also need to keep your co-processor choices in mind. Forwarding and classification engines are taking on more tasks than just searching tables, freeing up some resources, and necessitating code rewrites. If the architecture will differ substantially, how does the vendor

plan to move your code to the new architecture?

In this way, you can be developing code that may actually run on a future generation of the device. In some cases, waiting for the next generation will give you greater performance gains than writing better code. One issue to consider, however, is that network processors tend to advance in fairly large quantum steps: from OC-12, to OC-48, to OC-192, and beyond. It probably makes little financial sense, for example, to use an OC-192 chip for a high-performance OC-48 application. Therefore, a vendor focusing on building an OC-192 part won't really help your OC-48 design.

SILENT BUT DEADLY

Engineering, at least from a marketing point of view, is a race of features and getting to market. As an engineer, you probably face the fact that someone else has determined a product's time to market and features. Not surprisingly, the timing that marketing usually requests is "now." Marketing often bases these decisions on what the competition is doing. However, in the network-pro-

cessing space, the competition revelations are not all that revealing.

Stories abound about companies that suddenly go bankrupt just as they're on the verge of a revolutionary product. On the other hand, some companies are foundering in the design area but are ready to pounce on the market with an amazing product. Say, for example, that your competition is working on

a product using first-generation silicon. Second- or third-generation silicon is becoming available, but no product is yet in sight. If you start with the third-generation silicon, you have the advantage of more robust tools, more integrated features, and more libraries, right? Maybe not. Several silicon vendors admit that they never intended to enter volume production with their first-generation parts. They

intended these parts to serve as training chips to get engineers moving on designs using tools and writing code that would carry over to the next-generation parts. However, although designers completed this work in 18 months that now you might be able to do in six, they're now done, and you still have six months to go simply to catch up.

WHAT'S THE REAL COST?

Establishing prices for software is often a complex negotiation. You may pay in several ways, so to understand the software cost, you must first know about all fees. Possible fees include those for evaluation and development tools, source code, additional code modules, updates, cus-

tomization, and continuing support. Some vendors charge \$50,000 or more for evaluation platforms. Such charges are typically a filter rather than a way to generate revenue. The high initial buy-in discourages less serious companies and lets the vendor concentrate on serving com-

panies that are willing to commit.

These fees may be one-time costs or perpetual yearly fees. Vendors may charge on a per-vendor, per-seat, or per-application basis. In addition, you may have to pay a royalty. Vendors are reluctant to reveal royalty

fees because they base them on the application, which means the projected volume for the design. Another problem with royalties is that the vendor then wants to know how many boxes you've shipped, a figure few companies are willing to reveal.

HOW WELL DO YOU KNOW YOUR VENDOR?

Choosing to use a device in your design means developing a close relationship with that device's vendor. It pays to understand a vendor's agenda and its impact on this relationship. Before committing yourself, ask the following questions:

- After spending a lot of money on creating a fantastic product, does the vendor have any money left over to support you through your design phases?

- Where do you fall on the list of priorities of a vendor that has 100 design wins?

- If your vendor is a start-up, how long will its venture-capital funding hold up?

- Did the vendor decide whether you would be one of its design wins, or did you decide whether the vendor would get your design win?

- How experienced is the vendor with network processing?

- How well do you trust your vendor with your intellectual property?

Remember that the vendor prequalifies you, just as you prequalify it. Getting a design win involves more than just a promise to purchase a product. The

vendor must support you and has enough resources to support only so many customers. Also, the vendor's reputation is on the line when it accepts a design win. If the prospect of working with you seems too risky, the vendor may back off.

Many vendors are glad to tell you how many design wins they have lined up. If possible, find out how many customers the software vendor has actually shipped its product to. In other words, is the vendor counting only those customers who have committed to purchasing the product or including customers who are still evaluating the software before committing? A world of difference exists between these two types of customers.

Looking at a road map of new parts is great, but consider the status of existing devices. Ask for details about the road map, including the full timeline—from preliminary data sheets to simulation models and from reference designs to samples. See whether the vendor has met previous deadlines and promises.

Some vendors let you purchase protocols and extensions that they promise to deliver later. You may want to make this investment because you could get a lower price now than later, and this early purchase gets you into the loop early enough to drive the development.

Find out what market your vendor is playing in and whether the vendor will treat you as a fringe customer or part of what the vendor considers a key market. Some vendors count on making substantial additional revenue from offering professional design services. If you are not interested in purchasing these services, does the vendor provide the development tools and support you could get from a vendor that would prefer to let you do all your own design?

Also, find out whether the vendor is willing to license back from you code that you have ported or added extensions to.

You should also try to figure out your options if a vendor goes belly-up and whether your design investment is protected. Many small vendors want a large company to purchase them.

These sales can have far-reaching effects, such as a blockage in the information flow as the larger company tightens its policies, the loss of attention a single-product company can offer its customers, and the severing of relationships with now competitive partners.

Further, what kind of program has the vendor developed for software development, including for third parties? Look at the program to see what kind of support you can hope to receive from these vendors based on the type of support they receive from the hardware vendor. Also, which third parties are part of the program, and how committed are they? And when a vendor has a relationship with third parties, does the vendor or the third party take responsibility to fix bugs?

Remember, vendors do want you to be successful. After all, they start making money only when your product enters volume production. The question is: How much faith does the vendor have in you to be one of its design wins to show a profit?

TRAFFIC-GENERATOR TOOLS

Traffic-generation tools should support a variety of protocols under a variety of conditions. Ideally, you should be able to generate traffic not defined by a protocol; you might want to create packets with errors, imple-

ment an emerging protocol the tool doesn't yet support, or create flows that stress the buffering mechanisms in your system. You should also be able to generate traffic of your own or inject captured real-world traffic so you

can configure the system in different ways.

You should be able to compare the actual versus the expected results. Finally, you may know what the results should look like, but generating

those results and putting them into a format that can be compared with actual results may be time-consuming if the traffic generator cannot create expected results for you.

THE NETWORK PROCESSOR FORUM

The data plane of network processing includes all wire-speed processing; that is, all the necessary processing to send a packet in the handful of nanoseconds the processor allots each packet. The control plane includes that processing that does not immediately need to send a packet from a port. Such processing includes forwarding table updates and processing exception packets, which are those packets that are too "unique" to pass through the data-plane engine.

The NPF (Network Processor Forum) is now trying to develop standard hardware and software interfaces to abstract processing elements. The software APIs it is developing attempt to abstract network-processor elements or functions to a control processor. One purpose of the APIs is to let you create application and middleware code that will allow control processors and network processors to interface with each other. The API hides the hardware particulars of a task. For example, the NPF will in the second quarter vote on the IPv4 (Internet Protocol Version 4) API. This version defines such functions as "add route" or "delete route." For those applications that need to implement beyond plain-vanilla IPv4, designers can drill to a lower level to such functions as "modify table entry." These still-abstract functions provide access to specialized and unique features of various network processors. An example of an application-based API is classification; that is, the application understands just that certain functions are available, not whether the processor implements classification in software, an acceleration engine, or a coprocessor.

One promised result of this API work is to ease the challenge third-party vendors face in designing application software. Currently, if a vendor wants to

write an IPv6 (Internet Protocol Version 6) stack, it has to write code to an API that is unique to the network processor in use. Thus, a vendor's resources for porting the software to other network processors limits the ability of the vendor to leverage software to a larger market. With the NPF API, the vendor should have to write the code only once to the API. To run the code on a network processor, the vendor needs to write a "shim" layer that interfaces the API to the architecture and the API of the network processor. This approach promises to open the market for third parties, making it worthwhile for them to write APIs because they'll be able to sell enough of them to make it worthwhile and to enhance the software offering of any network processor with a shim for the API.

Many third-party vendors have developed their own shims to interface their APIs with specific network-processor APIs. By doing so, they can use an API that exposes details of both the application and the network processor to optimize the interface between the two. General interfaces are typically less efficient and more cumbersome than specific interfaces are. Thus, some vendors may continue to use their own architectures and support the NPF APIs to maintain a competitive edge. For example, it makes no difference to the application how you implement classification. However, once the application is running, the implementation can have a tremendous impact on system-level performance, such as how fast it completes a table update.

The NPF APIs focus on the control plane. With the variety of network-processor architectures and the vast differences in coding the network-processor engines themselves, it's unclear just how useful third-party net-

work-processor software is. In most cases, less than a few thousand lines of data-plane code run on the network processor itself. How much a third party would have to charge for such code is unclear. Additionally, this area is still one in which network-processor vendors and designers add significant design value and customization.

Candidates for future work within the NPF include multicast IPv4, MPLS, IPv6, and other protocols, which the NPF membership will determine. The NPF is also working on streaming interfaces, in which all data passes through each coprocessor or component, and look-aside streaming, in which only some data passes to the coprocessor and back into the stream. Many niches exist for vendors to develop coprocessors for encryption, classification, and traffic management. The goal is to create common interfaces to solve the current problem that coprocessors work with only one or a few network processors without using an FPGA bridge. The creation of this approach would allow designers to more easily mix and match devices.

At press time, the NPF projected that it would ratify a streaming interface by March. The first part of the look-aside interface, LA1, was also due in March. Devices that logically look like memory, including memory, content-addressable memory, and look-up engines, drove the interface. LA2, which does not have a target introduction date, will serve for devices with a request/response architecture. With such interfaces, you will be able to interface a network processor with a coprocessor without using complex software or an FPGA bridge. However, APIs generalize problems as they abstract them, creating inefficiencies and blocking access to nonstandard

features/acceleration engines on various chips. Thus, vendors will continue to offer their own APIs for high-performance applications that cannot suffer the inefficiencies. Also note that defining the control and data planes becomes less critical for lower speeds, and, in some cases, one chip can handle the entire processing task, possibly eliminating layers of abstraction. In any case, vendors will probably continue to support proprietary APIs to preserve the code base of existing designs. However, note which API the software truly supports. In other words, did the vendor rebuild the software architecture for optimization or merely add an extra shim layer to bridge the proprietary API with the standard API, thus adding another hidden layer of inefficiency?

The NPF is also working on various benchmarking standards. The need for benchmarks is undeniable. Yet even benchmarking from a black-box perspective with set traffic streams fails to account for the specialized functions of each combination of network-processing components.

Many vendors believe in the design that the NPF proposes. However, even if the NPF cannot reach consensus on low-level design issues, these vendors say, much good for the industry will have come from it. Many others feel that the act of standardization is absurd in a market in which every architecture is unique. In any case, it may be some time before you see wide implementation of the work of the NPF. Many vendors have spent lots of money and need to see some return on that investment before they make another upgrade. Pockets are only so deep, and many startups are discovering that it's expensive to support a customer after the design win.

SOME QUESTIONS ABOUT SOFTWARE

Vendors claim that the software they supply you will yield dramatic gains in time to market. It's worth looking past the claims and asking a few hard questions of your own. For example, what kind of API does the software offer for extending functions? Is the platform really open or just a proprietary API with some published spec sheets? How easy is it to add or revise code to the platform? Does the code have an API at both sides so you or third parties can build on top of the code? Does the software comply with the Network Processor Forum's APIs?

The vendor claims you'll get efficient compilation of its code. However, the code targets several platforms. How does the vendor guarantee that you'll have efficient compilation for your platform?

Some vendors claim that you cannot choose just pieces of their software without experi-

encing significant pain. The code is too integrated with itself, they say. (They'll encourage you to purchase code that they themselves don't offer instead.) Third parties offering a best-of-class block of code will probably disagree. In some cases, the third-party code will run on the other vendor's code in a way that not only doesn't damage the integrity of the code, but also adds features the code doesn't otherwise offer.

You sometimes gain an advantage when you use one vendor's code. For example, if you want to support IPv4, IPv6 (Internet Protocol versions 4 and 6), and MPLS (Multiprotocol Label Switching) code from a vendor, you can share resources, such as code, tables, and memory. Using code from three vendors may require three sets of resources.

How do you pick a "standard" piece of software, such as an IPv4 stack? Check the scalability

of the modules to the speeds and applications you plan to migrate your design to. Consider which extensions of the protocol the vendor supports and the vendor's history of supporting new extensions and changes to specs. Also, check how quickly the vendor has corrected errors that crop up in code. Confirm vendors' claims of standards compliance and whether the code has passed independent compliance testing.

How accessible is the code for customization? Will it take you longer to figure out how to customize the code than it would to write your own? Although off-the-shelf code can speed time to market, you still have to consider whether the look and feel of the code is compatible with that of your other products. Many vendors offer tools for modifying the management interface of control code, allowing you to use

your own screens and logos.

If you port the software to a platform, consider licensing it back to the vendor. Also consider that code you are preparing to invest in writing may suddenly become available from the vendor through a license from one of your competitors. If your design cycle slips, then the sweet spot in the market you targeted six or nine months ago may have moved sufficiently to warrant altering your design specifications. Can you make such an adjustment without disrupting your outdated design?

Once you decide on a various network processors and coprocessors, you still might want to consider how portable your code is. If one of your vendors goes out of business, what options do you have? Do you have rights to the chip design, and if so, do you really want to build your own chip?

PROFILER TOOLS

Profilers monitor the performance of a system over time. At a basic level, a profiler shows you how close your design is to processing packets within wire-speed limitations based on different traffic conditions. You need to know how much and how efficiently you're using the overhead you have for additional processing and services. The profiler can also tell you where your system hits the most congestion. The challenge is realizing when you've reached a point at which you approach the maximum capacity of the hardware.

Profilers work by capturing data during simulation. When you or a breakpoint stop the

simulation, the profiler can then sort through all the information it has captured. You should be able to search this data to see when a register was within a range or pull up a graph comparing values over time.

Some profilers allow you to track the path of, changes in, and duration of operations on a packet as you worked on it. You can track the number of packets crossing an interface. You can track the time it takes to process individual packets over time, viewing the minimum and maximum loads on the system and viewing events or functions. You can also monitor table management and confirm that the system efficiently maintains the

entries. You can monitor all of these functions by picking the appropriate registers to track, but a more useful profiler tracks such measurements on a functional level by, for example, letting you click on the "track-packets" option.

For multiprocessor/multi-thread architectures, you should be able to track the idle time the system spent within each processor/thread so that you can determine how to get that processor/thread to do more. You can also determine worst-case values for pipeline architectures or for processor/thread loading. The ability to track the use of both internal and external resources, such as memory

or look-up engines, is also important. However, for this information to be most useful, you need to be able to easily reallocate processor/thread loading so you can quickly test new balances.

Profilers seriously impact simulation speed. A good profiler lets you define what data to capture and how often—that is, how many cycles—to capture it. By limiting the data you capture, you can significantly increase simulation performance. This increase reduces the simulation time it takes to ferret out the results of corner cases or exceptions or to see how fast sustained bursts bring the system to its knees.

BELLS AND WHISTLES

Optional features make some tools more helpful than others. Among these features are configurability, bounds checking, and the ability to single step through a multiprocessor or multithread design.

You may need to configure a host of tools to initialize each device you wish to use. Tools that let you select settings and then set up the chip for both simulation and production code can save much time and prevent many simple but frustrating errors. With a bounds checker, you can prevent memory leaks, which are a disaster for systems that users never turn off. The

environment in this case monitors memory resources to flag such losses. For multiprocessor/multithreaded systems, single-stepping can mean the ability to single-step a processor/thread either while others run or sit suspended. Also, the ability to have complex triggers across multiple processors/threads can aid in understanding resource contention issues.

Other optional features include templates, drag-and-drop ability, and the use of scripts. Templates provide predefined constants for fields and structures within well-known protocols, which are useful for

field and bit manipulation. Drag-and-drop features allow you to view packets without typing or using checkboxes. For example, you can create a look-up key by dragging the appropriate fields onto a blank key. Scripts are useful for creating milestones or regression testing, a rigorous set of tests that your code passes without error. As you continue to change the code, you occasionally run the milestone script to ensure that you haven't created errors in the code with the new changes.

Libraries and comment display are also useful. Libraries can code common transformations,

such as conversions between protocols and the building of common look-up keys. Some debuggers can display comments alongside source code, meaning you don't have to constantly switch between screens.

Finally, the Web can provide invaluable support. Web-meeting software allows a geographically diverse team to attend demos or review meetings where all attendees can see the same screen. You can also have service support and watch and participate as you demonstrate a bug or ask a question on how to use software.

SOME REFERENCE DESIGNS ARE MORE EQUAL THAN OTHERS

Reference designs are a checkbox item on every network-processor vendor's marketing sheet. When a vendor offers a reference design, it has built a board with a device, written some software, and can get the little LED on the corner of the board to blink on and off. Some reference designs do no more than prove that the silicon won't explode when you apply current to it. Most claim to show you how much headroom is left beyond a basic application. Others, the vendors claim, are swell enough to put a logo on and sell as-is. In any case, silicon vendors generally focus on selling hardware, not software.

Will a reference design help you ship your product? In other words, vendors often use the term "reference design" for

"demo" or "tutorial," meaning that you have work to do even if you use the reference design as a baseline or foundation. The question is: How much?

In some industries, a general application reference platform can go a long way. In network processing, however, the varied and myriad architectures prevent general reference designs from having as much value as they might otherwise. A general network-processor design, for example, lets you add a variety of coprocessors. However, such a design is probably too general to take full advantage of the unique capabilities of the individual coprocessors, meaning that you must customize the code. For example, a reference design probably has different port configurations from the

configuration you want for your product. However, you may be unable to scale the code in this fashion; after all, the vendor may have written it as a one-shot deal to prove the silicon. Also, the reference design may offer only basic functions without giving you enough pieces to make a shippable product. The claims that a vendor can supply 90% of the software don't tell you much.

How do vendors determine what reference designs to create and support? Many talk of a push-and-pull dynamic. They ask their customers what they are designing and watch market demand so that they can guess the location of the sweet spot. A promise of a purchase order also helps.

Look at how many reference

designs and for how many applications the vendor offers. If programming the network processor or coprocessor is as easy as the vendors claim, then why don't they offer more software? Examine the reference design to see how flexible the code is and how straightforward the process will be for you to carry their code over to a design of your own. Challenge the vendor to prove the extensibility of the code to you by asking for a simple change in the code. See what such a change entails. Also check what language the vendor wrote the reference code in. The vendor may tell you that you can write code in C but then write code for the reference design in assembly or micro-code.

SIMULATOR TOOLS

If you wait until silicon is ready before you start evaluating a chip, you may have waited too long. Network-processor vendors typically offer simulation models of their chips months before samples of silicon will be available. In some cases, vendors announce chips a year before projected sampling. As a result, you run the double-edged-sword relationship of an alpha partner. On the one hand, you can influence the vendor and offer feedback that will facilitate easier design for you later. On the other hand, the simulation models the vendor provides you with will most certainly change as the vendor works out bugs and architectural kinks. Think about asking to see the revision specs for the simulation models. These specs might reveal something about how many and what kind of major architectural changes or minor tweaks the models have gone through, as well as how the vendor facilitated working these revisions into the customer design cycle.

Your design cycle will cross the simulation/silicon line, so many vendors design the rest of the tool chain such that these tools can't tell the difference between simulated chip data and actual chip data. In other words, you can use the same code, debugger, and profiler that

you will when working with actual silicon. A configuration tool that handles all the details between simulation and emulation is especially nice. The only real difference you should see, then, between a simulation and an emulation using a chip on an evaluation or prototype board is the speed of analysis. Because emulation employs hardware, it executes magnitudes of order more instructions per second than simulation does. On the other hand, simulators give you much greater access to internal data, code, and registers. However, you don't have much choice but to use the simulator until vendors offer either a physical sample or Verilog that you can download into an FPGA. You don't have much choice but to use the simulator.

Simulation speeds vary, depending on how you simulate the system. Verilog simulation runs at hundreds of cycles per second. Modular Verilog, in which you simulate only subsystems of the system in Verilog, can run at thousands of cycles per second. Full C-model simulation runs at tens of thousands of cycles per second. The numbers vendors quote, however, are typically higher than those you'll see in reality. As soon as you simulate on a system level, say by adding a look-up

coprocessor or connecting to a simulation/emulation of the control plane, you significantly reduce performance.

An important question to ask about the simulator is what the vendor means by a "cycle." Some simulators are instruction-accurate, versus cycle/pipeline-accurate, meaning that the simulator may not simulate system bottlenecks, such as resource blocking, cache flushes, contentious memory access, and so on. Simulating bottlenecks gives you a more accurate picture of the system but takes longer. For example, simulating a 333-MHz network processor even at 10,000 cycles/sec requires 9.25 hours for one second of traffic. How long will it take for a corner case to occur or a buffer to overflow?

You can take many approaches to speeding simulation. For corner cases, create data and forwarding tables to test extremes. You can also reduce the size or prepack overflow buffers. Disable breakpoints and turn off those parts of the trace capture that you don't need to reduce non-code-simulation processing. Adjust the sample rate of the trace from every 10 to 100 to 1000 cycles. Find out whether the vendor has additional hardware to accelerate portions of a simulation. Also, run no other

applications, including sending or receiving e-mail, while running a simulation.

Many real-world stresses, such as long, sustained burst of packets, are difficult to profile in a simulation environment. These tasks require the use of a hardware emulation system. Simulation works well for testing basic functions and system efficiency, but you still need to test the extreme but expected cases to see whether the configuration will fall apart under real-world stress.

Another important aspect of simulators is their extensibility. Does the simulator support a model of the coprocessor you plan to use, and does it support multiple models or multiple coprocessors? You can limit simulation to a component level, but one of the significant challenges of designing systems with such components is getting them to work together efficiently. What will porting a model yourself entail? Also consider how monitoring the simulation environment takes place. It can be useful to have hooks into the simulator, allowing you or a third-party vendor to extend the simulation environment, by either adding analysis tools or mining simulation data that the vendor has not yet made available.

OPEN-SOURCE OPERATING SYSTEMS

Linux is a serious contender as the operating system of choice for the control plane. Proponents of Linux, often a vocal group, cite many reasons for the success of open-source operating systems such as Linux. For one thing, "open source" means that you can use a range of available software and tools. You can also add your own functions, a difficult process when you're trying to build on a proprietary operating system. The disadvantage with

such a strategy, however, is that the various open-source operating systems have strict rules about how much of the intellectual property you put into your improvements that you have to make public. For this reason, some vendors support BSD rather than Linux, because the BSD open-source agreement makes it easier to protect proprietary intellectual property.

Another advantage of open-source OSs is that other parties

can improve the tool chain without investments from a network processor vendor or a designer. For IC vendors, however, an open-source OS can require a larger investment than do proprietary operating systems because they mean that the vendor may have to work with more than one company to support that OS.

In addition, Linux offers 64-bit addressing and SMP (symmetric multiprocessing) for applications

running on several across processors. Further, open-source OSs may have no royalty structures, and they can operate on PCs and workstations, meaning that you can develop and test code using familiar tools without requiring hardware prototypes, unlike some proprietary operating systems.

ANATOMY OF A THIRD-PARTY NETWORK-PROCESSING-SOFTWARE VENDOR

Third-party network-processor software vendors face a challenge in developing software for a market in which almost every design has some "special" functions. Code that manages the data plane, usually running on the network processor, is often less than a few thousand lines, and developers ply it with unique implementations depending on the application, silicon, and design team to differentiate their products. Thus, no large market exists for third parties targeting niche applications on the data plane.

On the control plane, however, some software vendors claim to offer complete software for

complex applications. But are there enough design wins to keep these companies in business? With a modular approach, they can reuse large amounts of their code base across several niche applications. By also offering design services, they hope to customize code for you.

Other vendors recognize that each customer has an interpretation of its application that is unique to its architecture and target market. In other words, each customer has its own twist to add. These vendors offer prepackaged applications with the caveat that the code will get you, say, 60% of the way there. They avoid uncommon or non-

standard functions, because they expect you to make changes at these points to yield system-level optimizations.

Many software vendors have developed their own "standard" APIs for their code and are working toward industry adoption of these APIs. The question is: Should this adoption be a criterion? The API may become the standard for the industry, but the vendor may then pursue its now-proprietary API and shun the accepted standard.

Few vendors are willing to abandon their proprietary APIs as groups, such as the Network Processor Forum, introduce standards. Most will support both so

that their customers' code base remains intact. Regardless of whether a vendor runs its API on the standard API or runs the standard API on its own API, you end up with an extra "shim" of code that may cause a slight loss of performance or functions.

Third-party vendors want you to think about your added value and hope that it isn't in the code they have to offer you. To some degree, they are right: Off-the-shelf code is becoming a commodity. In this light, is such code a safe choice to focus on as "special"?