

Optimizing heterogeneous architectures

A SYSTEMS APPROACH TO MULTICORE-DSP ARCHITECTURES EXTRACTS THE HIGH PERFORMANCE THAT TODAY'S APPLICATIONS REQUIRE.

The myriad DSP (digital-signal-processing) applications requiring the performance and integration of SOCs (systems on chips)—from consumer communications to multimedia products—have created new challenges for developers. Many of these challenges stem from the inherent complexity of the integration, development, testing, and verification of heterogeneous architectures comprising multiple processors, coprocessors, accelerators, and peripherals. A typical heterogeneous SOC comprises many subsystems and processors. For example, a multimedia processor may have a GPP (general-purpose processor); multiple DSPs; audio/video interfaces; codecs; multiple accelerators; wireless interfaces, such as WiFi; flash memory; high-speed communication links; sensors; and display controllers. With a comprehensive set of multicore-aware tools that address shared-resource contention, performance optimization based on actual usage scenarios, and power management across multiple cores, developers can confidently unlock the power of SOC devices and achieve fast market response. Additionally, proper adjustment of these subsystems enables optimization across features, power usage, and performance without requiring developers to completely redesign the product, as they might have to with a less integrated architecture.

Traditionally, a design team working with a heterogeneous architecture, such as a GPP and a DSP, usually divides itself into at least three groups. One group writes code for the GPP and primarily handles control processing. A second group writes code for the DSP and on-chip coprocessors, taking care of most of the data processing. A third group is responsible for system integration, bringing together both designs from the other two teams as well as dynamically repartitioning the system as new constraints and limitations manifest themselves (Figure 1). Most programmers worry about code density, memory usage, power consumption, and cache efficiency in a block of code. But you cannot accurately and individually measure these characteristics apart from the complete system. Contention for processing cycles, memory, and peripherals among multiple blocks of code determines the efficiency with which code runs.

The system-level-design group can see trade-offs that the programming teams cannot. For example, the complexity of programming the function on a processor usually determines the decision to program a function to the GPP or the DSP.

The group usually partitions control and user-interface functions to the GPP, which is better suited to them, and data-processing functions to the DSP. Sometimes, however, the obvious partitioning is less efficient.

Consider the autofocus function of a digital camera, an ideal task for the DSP to handle. It is easier for programmers to write the autofocus function for the DSP rather than for the GPP, and the code is more compact and runs faster. Given that the DSP also consumes less power than the GPP, this decision appears sound.

However, when taking into account typical camera-usage patterns, the choice is more complex. Often, a user turns on and focuses a camera but either is not looking at the screen or is waiting for the perfect moment to shoot a picture, meaning that the DSP is not in use. If the designer implements the autofocus function on the DSP, then, during this time, both the GPP and the DSP must be fully powered. If the designer implements the autofocus function on the GPP, assuming that enough overhead is available, the DSP could be in a powered-down state. When the user takes the picture, the DSP quickly returns to an active state and captures the image with minimal latency.

The trade-off is that the autofocus function may be less efficient to implement on the GPP, requiring more CPU cycles and

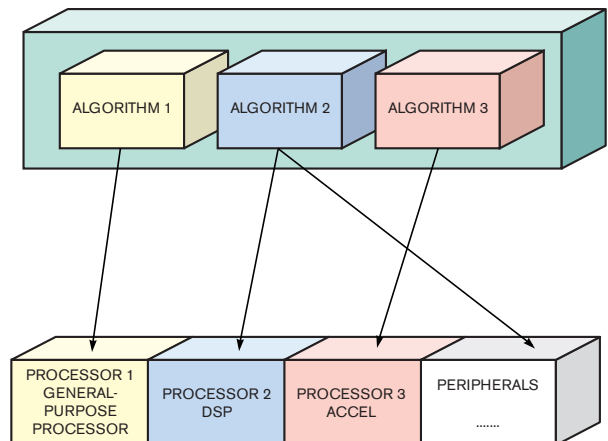


Figure 1 Traditional fixed partitioning yields design teams that usually divide into system- and processor-level development groups.

slower response; however, the significant power conservation that results from being able to power down the DSP, increasing overall battery life, more than offsets this inefficiency (see sidebar “Slave to efficiency”).

DYNAMIC PARTITIONING

When weighing partitioning advantages at a system level, programmers have to consider multiple usage scenarios. For example, the user is sometimes shooting numerous pictures in succession. If the GPP is in full use, the bottleneck in the system may be writing to the flash or the hard-disk drive, which limits both the maximum number of pictures that the camera can queue and the speed at which a user can take successive pictures. By offloading processing tasks from the GPP, the system can more quickly save images, increasing the performance of the camera. In this scenario, the system achieves optimal performance if the designer implements the autofocus function on the DSP.

Dynamic partitioning allows the programmer to implement a function on both processors, enabling optimized performance under both of these scenarios. When the picture queue is empty, the GPP handles autofocus, and the DSP powers down. When the user takes the first picture, the GPP wakes the DSP to capture the image. At the same time, it passes execution of the autofocus function to the DSP. While the DSP is active, it handles autofocus. When the DSP is no longer required—that is,

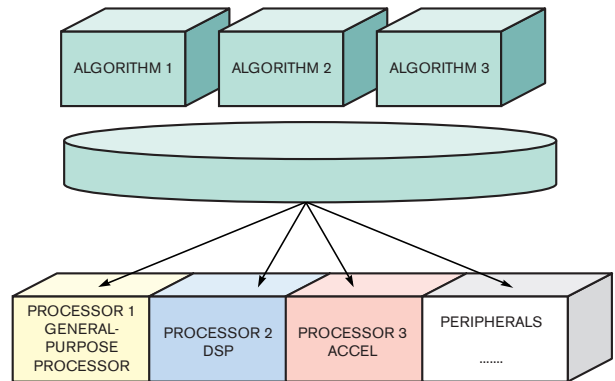


Figure 2 With tools and techniques such as dynamic partitioning, developers can optimize the performance of complex multicore architectures.

when the user is waiting to take another picture—the DSP alerts the GPP that it is ready to power down again. The GPP resumes executing the autofocus function and tells the DSP to power down. The programmer can seamlessly create an autofocus task for the GPP, the DSP, or both if the function can execute on either processor.

The chief concern when implementing dynamic partition-

SLAVE TO EFFICIENCY

One technique for conserving power in a heterogeneous system is to run the multiple processors in a master-slave configuration. The master processor manages system events and the user interface, and the slave processor powers down the system; both processors need not consume power during low-usage or idle periods. In general, the GPP (general-purpose processor) or control processor acts as the master, because designers usually implement system-management and user-interface functions on this processor. The system can handle simple events, such as updating a clock or responding to the keyboard, without waking the DSP. If the DSP were mas-

ter, then any event would trigger waking the GPP to handle it. For this primary reason, the GPP is the master, even though it typically consumes more power than the DSP.

When a processing event occurs, the master wakes the slave with the function it has to perform. For a multifunction device, overlays can complicate waking the DSP. To increase processing efficiency and power usage, you must load the proper algorithms into the DSP's fast program memory—which may be too small to hold all of the algorithms a device might need to support—before you awaken it. In most cases, you can store the program overlays in the extended memory of

the master processor. In this way, the master processor can overlay the appropriate code in the slave processor's fast program memory.

Although the system-level team manages overlays, overlays complicate code writing for the other two groups on the design team. They must write code without fixed address dependencies; the code must execute from anywhere in program memory and access variables from relative pointers. Given a limited amount of fast program memory, programmers shouldn't place infrequently used code in it. Programmers may also need to limit the use of global variables. Although using global variables may

simplify programming, they consume scarce memory resources whether or not those variables are in use at the time. If you are using an RTOS, you can configure the operating system or kernel to manage these concerns for you.

Note that you need not entirely power down the slave. In some cases, you may just want to slow down the slave to maintain a proper function and performance. For example, you can reduce the slave clock rate to a level just low enough to perform real-time autofocus. Programming the slave is like pressing the gas pedal on your car: You maintain full control of speed and power usage, and no power goes to waste.

ing is that the programmer must write the same code twice for different processors. In most cases, they write such code in C; therefore, porting this code between the processors requires minor adjustments to the optimized code that the appropriate compiler produces. Note that, when you have both sets of code at hand, you gain other advantages. For example, the autofocus function runs faster and consumes less power on the DSP than on the GPP. Today's development environments have advanced tools for profiling memory usage, including cache visualization; power-consumption monitoring; and bus-traffic analysis, enabling you to determine whether this information is true once you can profile execution on both the GPP and the DSP.

With both sets of code, you can also begin to test more complex partitioning schemes that you would otherwise be unable to evaluate. The performance of the system is not equal to the sum of its parts. It depends highly on the usage scenario and other active code. For algorithms such as video that put a heavy load on the DSP, you could shift the autofocus function back to the GPP. You could test partitioning scenarios among several functions, determining which offers the best power consumption and performance. You could automate such testing within the development environment, running through various combinations to find the optimal configuration based on performance, power consumption, or both. Having two sets of code allows system integrators to experiment with new configurations without forcing unexpected changes back onto the pro-

gramming groups. However, you can't achieve any of these goals if you are locked into running the code on only one processor (see sidebar "Cutting core count").

DANCING THE DEBUGGER TWO-STEP

An important system element of heterogeneous architectures is debugging of emulation circuitry. Such internal circuitry in the processors reduces the amount of debugging instrumentation in your code, giving you a more accurate runtime profile of your system. You also have a better idea of how you're using memory and can better optimize the use of fast memory. For example, high-performance DSPs include advanced circuitry to support nonintrusive ICE (in-circuit emulation), manage JTAG scanning, monitor and control power, enhance trace capabilities, monitor bus activity, perform high-speed data exchange, enable system triggering, and support other functions critical to providing high system visibility during debugging sessions.

Another critical component is multiple-core awareness in the debugger and development environment. You need to be able to synchronously stop multiple processors to solve some debugging problems. Such tools speed development by keeping you aware of overlays updating variable addresses and watch points when code moves within an overlay or between processors. You might also need to occasionally disable the debugger. Perhaps you want to test the responsiveness of a core coming out of sleep mode; it can't be asleep if it's supplying trace information. Alternatively, you may need a communications stack to continue

CUTTING CORE COUNT

With the increasing capabilities of processors, designers can scale back some multicore designs to a more traditional, single-core design. For example, a typical DSP can also handle user and peripheral controls. On the other side, a powerful GPP (general-purpose processor) can handle a single channel of MP3 decoding without the help of a DSP. Designers can now implement multicore systems in a single-core chip: a single DSP, GPP, or FPGA.

It is equally important to understand that a multicore SOC (system on chip) may be a poor fit for a single-function device, such as a radio-controlled watch or a hearing aid. With these devices, it

would be uncommon to need user-interaction and reprogramming capabilities. Programmers can typically dedicate the core in these applications to do the job in predetermined conditions and use much less power. The flexibility and scalability of a multicore device add little value for these applications.

Additionally, in some products, power consumption is less of a concern. For these products, the designer typically increases the clock rate to get the processing power to do the various jobs a single core usually handles. A device operating at approximately 1 GHz is a typical choice. For infrastructure applications, the power consumption per channel is compet-

itive, even if the device power is 1 to 2W. You can scale its needed performance and power consumption by selecting clock rates of 600 MHz to 1 GHz and still maintain code compatibility.

Design engineers may also ask, "When do I decide how many cores I want to use for my design?" The answer to this typical question is to wait as long as necessary, but don't compromise the software or features. This approach provides the flexibility of a design that can evolve throughout the process. Product requirements change frequently in response to market situations during the development cycle. So many companies adopt a spiral

model for product development. Each spiral involves feedback, refining and redesigning, optimization, cost analysis, and other factors. More design flexibility means more spirals. An original multicore SOC design may end up as a simple single-core system, or it could go the other way. In most cases, it is easier to scale down a multicore SOC to a simpler, single-core design. Many companies have developed in-house or commercial-level tools to facilitate high-level design by mixing software and hardware modules. So, a more important determination is how flexible and capable the design team is when responding to rapidly changing market situations.

processing, even though the system has encountered a breakpoint, so that you don't lose information.

Remember that you can use communication stacks such as USB or IEEE 1394 to transfer useful debugging information when the application is not using them. With a stack on both cores, you can more easily debug multiple cores synchronously. You also gain control over how you can pass and format debugging data without needlessly impacting the overall system. For example, if you use printf only in debugging code, you could cause the linker to bring in large library elements that would otherwise not be part of your system.

Developers familiar with only one side of the process—either GPP or DSP—have difficulty evaluating trade-off decisions early in the design process. Additionally, they can't repartition a function to optimize performance or enable dynamic partitioning, and the team that ends up porting the code cannot use many of the lessons they learned writing the code. With the right com-

AS DEVELOPERS TEAR DOWN THE WALLS BETWEEN SUBSYSTEMS, THEY CAN UNLOCK THE ADVANCED CAPABILITIES OF EVEN THE MOST COMPLEX SOC ARCHITECTURES AS SIMPLY AS THEY DO WITH SINGLE-CORE ARCHITECTURES.

bination of code and tools, however, they can start to view their application at a system level, instead of as separate processing units. They can work at a higher level of abstraction rather than think of programming the GPP or the DSP.

KNOW THY USER

It's important to spend time evaluating usage scenarios before you begin partitioning. It is a waste of resources to write code for both processors for a function that achieves no gains through dynamic partitioning. Begin by defining typical and worst-case usage scenarios. Then evaluate the scope of each function in the context of each scenario and its dependencies upon other functions.

Next, prioritize functions. You cannot compromise the processing of an image for storage, and it has the potential to become a system bottleneck. On the other hand, you can scale back the preview function, showing the picture a user is about to take, either in quality or in resolution if the system is overloaded. Users would rather have high-quality captured images than high-quality temporary images that they may never look at.

Developers need to package functions that they can dynamically partition so that the system integrator can easily use either version. Consider implementing a standardized set of coding conventions and APIs that abstract where a function runs and "wraps" the algorithm for system-ready use. Such guidelines are

MORE AT EDN.COM



+ We encourage your comments!
Go to www.edn.com/ms4162 and click on Feedback Loop to post a comment on this article.

readily available; for example, Texas Instruments provides the TMS320 DSP Algorithm Standard, also known as Xdais. You probably also need to create a transparent transfer function that passes all relevant state, global, and system variables between processors to ensure a smooth transi-

tion when you dynamically repartition a function; no glitches should be on the screen when handing off autofocusing. An embedded kernel or operating system, such as BIOS/Link, provides straightforward mechanisms for efficiently bridging master and slave cores without requiring developers to reinvent such mechanisms.

Consider which peripherals and computational units a function needs to access. Some processors enable you to turn off portions of a chip that you aren't using. For example, you can use DMA to increase performance of a function. However, if the system is not in full use, you might increase power efficiency by dynamically not using DMA and turning off the DMA circuitry.

You can scale down functions. For example, if you have a dual processing core, you can choose to simultaneously process two images or two halves of the same image. In the first case, you need two full image buffers; in the second case, you can get away with a single image buffer by working on the same image in two parts and halving memory needs. Alternatively, some devices have multiple accelerators and coprocessors that allow for a long processing pipe. As a result, you can achieve high-performance processing through optimized parallel computing; imagine 10 engines running in parallel to process a block of data. Again, you have full control on how to use the available engines.

SOC programming used to be a difficult undertaking. However, as SOC designs become hardware/software hybrids and developers further abstract designs, designers can evaluate trade-offs at the system level. With the advanced high-level tools available today, a developer can seamlessly design a heterogeneous GPP, DSP, or another subsystem, bringing a system perspective across the entire design team. In this way, the traditional three-group development team can evolve to a more efficient single team with common expertise. This team, working in a single unified design environment, can develop a base hardware and software design for different products; for example, they can use the same heterogeneous SOC design for DSC (digital-still-camera) and DV (digital-video) camcorder products.

As developers tear down the walls between subsystems, they can unlock the advanced capabilities of even the most complex SOC architectures as simply as they do with single-core architectures (**Figure 2**). With the right tools and techniques, such as dynamic partitioning, they can squeeze more performance and longer battery life from their designs and more quickly bring products to market. **EDN**

AUTHOR'S BIOGRAPHY

Bruce Lee is a senior engineer at Texas Instruments. He has a bachelor's degree in electrical engineering from McGill University (Montreal, PQ, Canada) and more than 10 years of R&D experience in communications, robotics, and machine visions.