CMP
United Business Media

Embedded Systems Design

feature

Multicore architectures can provide the performance boost you're looking for, but the software is certainly more complicated.

Is symmetric multiprocessing for you?

BY DAVID N. KLEIDERMACHER

For the past thirty years, computing has enjoyed continual boosts in performance, primarily due to increases in clock speed, pipelining efficiency, and cache size. Recently, however, traditional microprocessor optimization has hit the proverbial wall. Although tweaks such as further cache size increases can continue to nudge system performance, it's clear that Moore's gains are behind us. Meanwhile, embedded systems continue to grow in software complexity, with consumers expecting that all the bells and whistles will continue to come in ever shrinking cost, size, weight, and power footprints.

Microprocessor designers have concluded that the best path toward meeting the growing demand for performance with controlled footprint is to employ multicore architectures, in which the main premise is to partition the software and parallelize or offload execution across multiple processing elements. *Symmetric multiprocessing (SMP)* is one such architecture, consisting of homogenous cores that are tightly coupled with a common memory subsystem, as shown in Figure 1. SMP is a *de facto* standard on the desktop, but adoption in embedded applications has been slow, with recent surveys showing only a small percentage of designs using single-chip SMP-capable devices.

So if your design is in need of some extra horsepower, how can you determine whether SMP is a sensible choice? Several key requirements enable you to realize the promise of SMP. First, the software must be partitioned and parallelized to take advantage of the hardware concurrency. Second, operating systems must provide the load-balancing services required to enable distribution of software onto the multiple processing

An example of a symmetric multicore system is shown.

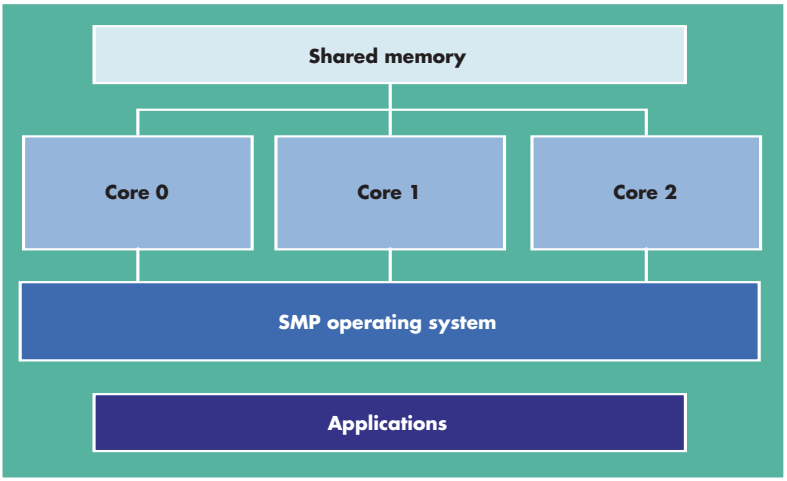

Figure 1

elements. And finally, you will need to learn and use development tools specifically tailored to the difficult task of multicore system debugging so you can find concurrency problems quickly and avoid time-to-market delays.

PROGRAMMING FOR CONCURRENCY

If your software has no potential for application-level parallelism (for example, a simple control system), then SMP is not for you. If software has the potential for parallelism but isn't currently multithreaded, then SMP could still be a good fit.

There are two ways to partition and parallelize software to take advantage of multicore concurrency: manual and automatic parallelization. *Manual parallelization* requires the programmer to deduce which parts of the application can be parallelized and write the code such that this parallelism is explicit. For example, the developer can place code into threads that will then be scheduled by an SMP operating system to run concurrently.

Automatic parallelization involves using a tool to discover a program's "parallelizability" and convert the code into an explicitly parallelized program. Some forms of parallelization focus specifically on loops. This approach is sensible: loops tend to be execution bottlenecks and sometimes can be converted into parallelizable iterations. However, many loops aren't parallelizable (even with a very smart compiler), and many applications simply don't benefit from this approach.

If software has the potential for parallelism but isn't currently multithreaded, then SMP could still be a good fit.

Parallelizing compilers do exist, but the embedded software community hasn't found automatic parallelization (autoparallelization, for short) technology to be of general use due to the compilers' focus on data-level parallelism. Certainly, a developer wouldn't take a legacy embedded control application running on a unicore platform and expect a parallelizing compiler to convert the application into something that runs optimally on an SMP. Autoparallelization may indeed boost performance in places, especially when the user can add some hints and directions to aid the compiler (known as semi-automatic parallelization), but a systemwide approach is required in general. Future innovations in autoparallelization could be more effective.

POSIX

POSIX is a collection of open standard APIs specified by the IEEE for operating system services. *POSIX threads*, or *Pthreads*, is the part of the standard that deals with multithreading. The Pthread APIs provide interfaces for run control of threads, synchronization primitives, and interprocess communication mechanisms. While other multithreading standards exist, Pthreads is the most generic, widely applicable standard. Pthreads are supported by a wide range of embedded operating systems such as INTEGRITY, LynxOS, and QNX.

Due to POSIX's ubiquity, a large base of application code exists that can be reused for embedded SMP designs. Another strong advantage of POSIX is its independent conformance validation. The list of POSIX implementations that have been certified conformant to the latest POSIX specification can be found at *http://get.posixcertified.ieee.org/cert_prodlist.tpl?CALLER=index.tpl*. By programming to the POSIX API, developers can write multithreaded applications that can be ported to any multicore platform running a POSIX conformant operating system.

In embedded systems, add-on software components can often be easily mapped to individual threads. For example, a TCP/IP network stack can execute within the context of one POSIX thread; same for a file system server, audio application, and so forth. Because of this, many embedded software systems can take advantage of SMP to improve performance without significant application modifications.

LANGUAGE-LEVEL CONCURRENCY

Because threads are an integral part of the Java and Ada languages, designing multithreaded software in these languages is relatively natural. Java and Ada programs using language-level threading can map nicely to SMP. Yet C

and C++ remain the most popular languages for embedded systems. Surveys in recent years have shown C and C++ (which lack native thread support) accounting for about 80% of embedded software, with no significant downward trend.

If your software base is hopelessly dependent on a real-time operating system (RTOS) that doesn't support SMP, then SMP may not be for you. If you have the freedom to select a new operating system, your best bet at future portability is to select one that supports both POSIX and SMP. An SMP operating system will simply schedule concurrent threads to run on the extra cores in the system. This automatic load balancing is the primary advantage of SMP: adding cores will increase performance, often dramatically, without requiring software modifications.

There's one important exception to the automatic reusability of multithreaded applications on an SMP system. Most SMP operating systems will allow threads at varying priority levels to execute concurrently on the multiple cores. Most real-time embedded software is written for a strictly priority-based preemptive scheduler. Trouble will ensue if the software is using priority as a means of synchronization. For example, software may manually raise a thread's priority to preempt another thread. On an SMP system, this preemption won't occur if the two threads are the highest priority runnable threads on a dual-core system. Embedded designers must analyze their systems to ensure that the SMP scheduling algorithms won't pose a problem.

CORE BINDING

If your embedded system has tight real-time deadlines, than SMP may pose a problem: context switches can be delayed due to the overhead of interprocessor interrupts (IPIs) and cache inefficiency. For example, when an interrupt service routine executes on one core and signals a thread to run, the SMP scheduler may decide to run the thread on a different core, requiring an IPI. If the thread didn't last run on that same core, there will be additional overhead to rewarm the cache with the thread's code and data. SMP operating systems tend to migrate threads, making it difficult to predict whether this overhead will be incurred.

The good news is that most SMP operating systems provide the ability to map interrupts and bind threads to specific cores to specific cores. Thus, real-time performance can be accommodated while other software is optimized across the multiple cores as deemed appropriate by the RTOS. The bottom line: real-time systems can take advantage of SMP, but designers should be prepared to spend time tweaking the system's scheduling parameters.

The high-speed interconnect is the centerpiece of the NUMA system.

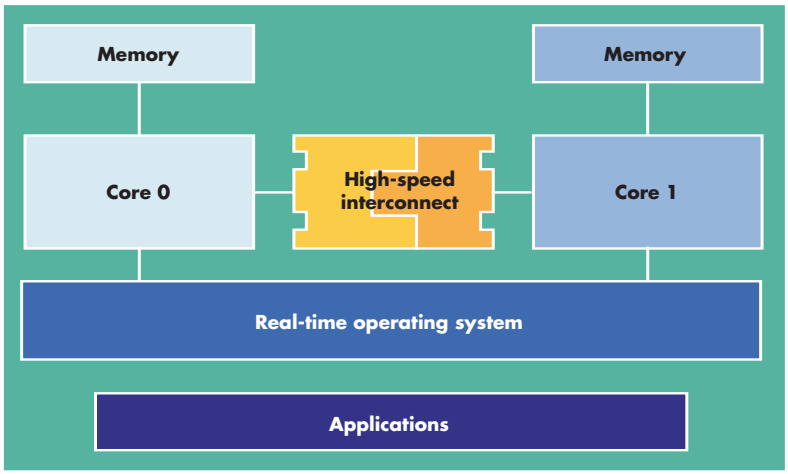

Figure 2

NUMA FOR EMBEDDED

SMP's single memory-bus architecture may be a poor fit for memory- and I/O-bound applications, relative to compute-intensive systems. The only way to be sure of the payoff is to run the software on an SMP. However, engineers sitting on the SMP fence may be excited about the prospect of NUMA (non-uniform memory access) systems. NUMA is similar to SMP except that the system contains more than one memory source, where the time to access each memory source varies. This architecture is depicted in Figure 2.

NUMA represents a compromise in which code can still be shared and automatically load-balanced in the manner of an SMP. Yet you can optimize memory access times by running threads on the core for which the thread's memory references are local. One way to do this is simply to take advantage of the aforementioned binding capabilities of the SMP operating system. You can locate thread-required memory to a core's local memory bank and bind the thread to the same core. The NUMA-aware operating system may automate this optimization of memory and thread binding. Although NUMA isn't available in mainstream embedded devices, there are rumors about future parts that could provide an intriguing alternative to SMP in the future.

When moving to an SMP platform for the first time, developers must be prepared to use tools required in the multicore development, debugging, and optimization process. Tightly coupled multicore processors often provide a single on-chip debug port (such as JTAG) that enables a host debugger, connected with a hardware probe device, to debug multiple cores simultaneously. With this capability, developers can perform low-level, synchronized run control of the multiple cores. Board bring-up and device-driver development are two common uses of this type of solution.

The development tool lets developers visualize all the system's cores and choose any combination to debug, each optionally in its own window. At the same time, the tool provides controls for synchronized running and halting of the debugged cores.

RUN-MODE MULTICORE DEBUGGING

Run-mode debugging is also useful for SMP systems, as the cores are never stopped. Rather, the debugger controls application threads using a communications channel (usually Ethernet) between the host PC and a target-resident debug agent.

The SMP operating system typically provides an integrated debug agent (and the associated communications device drivers) that's operating-system–aware and provides flexible options for interrogating the system. For example, one operating system comes with a powerful debug agent that communicates with the debugger, providing the ability to debug any combination of user threads on any core. The user can set specialized breakpoints that enable user-defined groups of threads to be halted when another thread hits the breakpoint. Some classes of bugs require this fine-grained level of control.

By collecting a system's execution history and making it available for playback within debugging tools, even the most difficult multicore bugs become easy to find and fix. If you're new to SMP, choosing a processor with on-chip trace capabilities may be desirable.

Multicore trace capability is just starting to arrive on multicore processors. A major technical challenge that has kept this hardware feature from becoming a reality involves finding a way to keep up with trace data emitted simultaneously from multiple cores. An emerging solution is high-speed serial trace (HSST). HSST replaces the current generation of parallel trace ports by taking advantage of high-speed serial bus technology, which enables higher data throughput with a lower pin count. HSST has been proposed to the Nexus standards committee. In addition, ARM has adopted HSST as part of its CoreSight trace solution.

SMP is a promising technology for improved performance in an attractive cost and power footprint. However, SMP is not a panacea. The application must have the potential for concurrency, and designers may need to manually refactor software to unlock this concurrency. Furthermore, SMP systems are more difficult to manage and debug than unicore designs. This in turn may require switching operating systems and tooling to acquire the load balancing and multicore debugging capabilities that go hand in hand with SMP. ■

SMP's single memory-bus architecture may be a poor fit for memory- and I/O-bound applications, relative to compute-intensive systems. The only way to be sure of the payoff is to run the software on an SMP.

David Kleidermacher is chief technology officer at Green Hills Software where he has been designing compilers, software development environments, and real-time operating systems for the past 16 years. David frequently publishes articles in trade journals and presents papers at conferences on topics relating to embedded systems. He holds a BS in computer science from Cornell University, and can be reached at davek@ghs.com.