Introduction

At one time, most embedded systems had modest software requirements — typically, a few thousand source lines of code. Today, however, an embedded system may contain hundreds of thousands or even millions of source lines, and use a large number of software components that interact in complex ways. In fact, research suggests that the code base for the average embedded project is doubling in size about every 10 months.

Typically, the many subsystems, processes, and threads that make up a modern embedded system are developed in parallel with one another. The design is divided among multiple development groups, each with its own performance goals, task-prioritization schemes, and approaches to runtime optimization. Once these subsystems are integrated into a common runtime environment, all parts of the system must provide adequate response under all operating scenarios, including normal system loading, peak periods, and failure conditions.

Given the parallel development paths, performance issues invariably arise when integrating the subsystems from each development team. In fact, many of these issues emerge only during integration and verification testing, when the cost of software redesign and recoding is at its highest. Unfortunately, few designers or architects are capable of diagnosing and solving these problems at a system level. Designers must juggle task priorities, possibly change thread behavior across the system, and then retest and refine their modifications. The entire process can easily take several calendar weeks, resulting in increased costs and delayed product.

Balancing security and upgradability

To complicate matters, many embedded systems today are dynamic, network-connected devices that can (or must) support new software functionality throughout their mission life. This upgradability offers numerous advantages, but it also poses a multitude of design challenges. How, for instance, can a device download and run new software components, without compromising the behavior and realtime performance of existing components? And how can the device perform these upgrades safely, while
maintaining continuous availability and a high level of security? In such cases, it is rarely feasible to add more hardware or processing power to handle the new features. Systems designers must therefore employ advanced techniques to guarantee that every software component, both old and new, can always access the computing resources it needs, including CPU time.

**Partitioning as a Solution**

Recently, the concept of partitioning has gained mindshare as a way to manage system complexity and to ensure higher system availability. Briefly stated, this approach allows design teams to compartmentalize software into separate partitions, where each partition is allocated a guaranteed portion (or budget) of system resources. Each partition provides a stable, known runtime environment that development teams can build and verify individually. This partitioning enforces resource budgets, either through hardware or software, to prevent processes in any partition from monopolizing resources needed by processes in other partitions.

Systems designers can choose between hardware or software partitioning. Time, cost, and performance considerations will heavily influence which approach they select.

**Hardware partitioning**

The concept of hardware partitioning isn’t new. In fact, most large systems provide some level of hardware separation for different functions. For example, in the networking industry, each card (line card, switch card, control plane processor) within a shelf may have its own dedicated processors. Even with this division of responsibilities, specific cards, such as supervisory processors and control plane processors, are often stretched thin by the sheer volume and complexity of the software that they must host.

Typically, an embedded system has a central processing complex that provides the overall brains for the system. As software complexity increases, it becomes a challenge to integrate all software subsystems into this centralized complex. A hardware partitioning approach employs dedicated hardware to run these various subsystems, with the goal of minimizing resource contention between them. (For a description of such an approach, see “Extreme Partitioning,” *Embedded Systems Programming*, www.embedded.com/showArticle.jhtml?articleID=171203272.)

Hardware partitioning uses added hardware to reduce the risk of software development and to minimize overall project-cycle time. It thus offers a feasible alternative for systems that ship in limited volumes and where the deployed system cost is small in comparison to the development cost. For products where price is important, software partitioning is more appropriate.

**OS-controlled partitioning**

An embedded operating system (OS) is a good candidate to enforce partition budgets since the OS already controls access to underlying computing resources. For instance, some realtime OSs (RTOSs) provide memory protection to prevent coding errors in one process from damaging memory used by other processes or by the OS kernel.

A partitioning OS can also help control CPU time. To accomplish this, the OS implements a partitioning scheduler that guarantees CPU cycles for each partition while still providing the deterministic, realtime
response required by embedded systems. Depending on the OS, the scheduler will either provide fixed partition budgets or offer a more dynamic approach that allows partitions under heavy load to access CPU cycles unused by other partitions.

By leveraging OS-controlled partitioning, systems designers can address many of the issues associated with complex design and system integration. For example, the designer can divide the system into partitions that are constructed and tested according to their partition budget. This approach removes complex CPU contention issues and thereby simplifies system integration. The net effect is improved time to market — provided that the partitioning scheduler can be dropped into the existing design. If developers must redesign their applications to fit with the OS partitioning model, then the costs of using OS-controlled partitioning could easily offset the savings.

Scheduler efficiency is also an issue. An inefficient partitioning scheduler will require additional (or faster, more expensive) hardware to deliver the same level of responsiveness as a traditional priority-based preemptive scheduler. Thus, designers must carefully consider scheduler efficiency when selecting a partitioning-scheduler approach.

**Application-level partitioning**

If developers are using an OS incapable of enforcing partition budgets, they can still implement application-level partitioning. For instance, the developer could develop a monitor application that continually examines the CPU usage of every thread to detect when a thread consumes too much CPU. To succeed, this approach requires techniques to notify the thread and to have the thread "back off" or yield in some way.

<table>
<thead>
<tr>
<th>Partitioning Approach</th>
<th>Product Cost</th>
<th>Software Development Cost</th>
<th>Time To Market</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Partitioning</td>
<td>- Redundant hardware cost passed onto customers; results in less-competitive pricing</td>
<td>- Less software complexity; requires less development effort</td>
<td>- Favorable time to market, but higher hardware costs</td>
</tr>
<tr>
<td>OS-controlled Partitioning</td>
<td>- Minimal processing overhead</td>
<td>- Minimal software complexity to provide CPU guarantees</td>
<td>- Favorable time to market, assuming that existing code base can be easily used</td>
</tr>
<tr>
<td>Application-level CPU Control</td>
<td>- Processing overhead consumed by application may require incremental hardware</td>
<td>- Complex design required to manage CPU allocation</td>
<td>- Complicated, multi-party design &amp; implementation reduces productivity and slows time to market</td>
</tr>
</tbody>
</table>

**Figure 1** — Comparison of partitioning approaches.
This approach uses a complex design that provides only a coarse level of CPU control. It also relies on the correct behavior of all participating threads. Without considerable refinements, this approach can make it hard to differentiate between a runaway error condition and a thread that is simply busy performing its intended function.

Besides spending time and effort to build the monitor application, the developer must also implement behaviors in each participating thread to monitor the thread’s CPU use. The cost of such development quickly multiplies as more threads are added. An application-level approach to controlling CPU utilization can also be difficult to maintain and scale. As a result, developers spend valuable design resources on this solution instead of developing core product features.

Figure 1 summarizes the merits of each partitioning approach. The case for a partitioning OS becomes even more compelling as we consider the additional benefits it can offer.

**Process Starvation and the Case for OS-controlled Partitioning**

A key aspect of OS-controlled partitioning is the ability to guarantee CPU time. To date, OSs have used multithreading and priority-based scheduling to help control which tasks consume CPU cycles. Systems designers can assign a priority to each thread and then schedule the threads for execution by using a

![Diagram of Priority Scheduling](image)

**Figure 2** — Priority scheduling ensures that the most critical tasks gain access to the CPU, but it can also cause problems when a high-priority task inadvertently or maliciously consumes all available CPU cycles. For instance, Task A prevents all other tasks from accessing the CPU after T4.
scheduling policy such as round robin or first in, first out (FIFO). Priorities dictate which queue of ready threads will execute first and the scheduling policy dictates how threads of equal priority share the CPU.

Priority-based scheduling provides an easy method to define the scheduling priority of every task. It does pose a problem, however: If a given task is even one priority level higher than another task, the higher-priority task has the power to completely starve the less-critical task of CPU time. For instance, let’s say you have two processes, process A and process B, where A has a slightly higher priority than B. If process A becomes swamped with work or becomes the target of a denial of service attack, it will lock out process B (as well as any other lower-priority process) from accessing the CPU.

This problem occurs in all fields of embedded software. In the automobile, process A might be the navigation display and process B the MP3 player — if the navigation system consumes too many CPU cycles when performing a route calculation, it can starve the MP3 player and cause MP3s to skip. In a network element, process A might be the TCP/IP stack and routing protocols and process B the SNMP agent. In an industrial control system, process A could be the robot-arm control loop and process B the human machine interface (HMI).

No matter what market an embedded system is targeted for, task or process starvation represents a serious concern. Services provided by lower-priority threads — including diagnostic services that protect the system from software faults or denial-of-service attacks — can be starved of CPU cycles for unbounded periods of time, thereby compromising system availability. The inability to provide guaranteed

![Fixed Partitioning](image)

Figure 3 — While fixed partition scheduling will prevent a high-priority task (for instance, task A) from consuming all CPU cycles, it can also sacrifice performance by wasting unused CPU cycles in a given partition (for instance, partition 3).
CPU budgets for lower-priority tasks can also make it difficult to integrate subsystems from multiple development teams, since it allows tasks in one subsystem to starve tasks in other subsystems — a problem that may not become obvious until integration and verification testing. These issues become more acute as system complexity (and the number of threads) increases.

To address this problem, some RTOSs offer a fixed partition scheduler. Using this scheduler, the system designer can divide tasks into groups, or partitions, and allocate a percentage of CPU time to each partition. With this approach, no task in any given partition can consume more than the partition’s statically defined percentage of CPU time. For instance, let’s say a partition is allocated 30% of the CPU. If a process in that partition subsequently becomes the target of a denial of service attack, it will consume no more than 30% of CPU time.

Fixed partition schedulers have their drawbacks, however. First, each partition must be defined at system build time and requires code modification to implement and change. Also, because the scheduling algorithm is fixed, a partition can never use CPU cycles allocated to other partitions, even if those partitions haven’t used their allotted cycles. This approach squanders valuable CPU cycles and prevents the system from handling peak demands. Because of this wasted time, fixed cyclical schedulers can achieve only 70% CPU utilization. Manufacturers must, as a result, use more expensive processors, tolerate a slower system, or limit the amount of functionality that the system can support.

**Adaptive Partitioning**

Another approach, called adaptive partitioning, addresses these drawbacks by providing a more dynamic scheduling algorithm. Like static partitioning, adaptive partitioning allows the system designer to reserve CPU cycles for a process or group of processes. The designer can thus guarantee that the load on one subsystem or partition won’t affect the availability of other subsystems. Unlike static approaches, however, adaptive partitioning uses standard priority-based scheduling when the system isn’t under full CPU load or attack. As a result, threads in one partition can access any spare CPU cycles unused by threads in any other partition. This approach, which was pioneered by QNX Software Systems, offers the best of both worlds: it can enforce CPU guarantees when the system runs out of excess cycles (for maximum security and guaranteed availability of lower-priority services) and can dispense free CPU cycles when they become available (for maximum performance).

Adaptive partitioning offers several advantages, including the ability to:

- provide CPU time guarantees when the system is heavily loaded — this ensures that all partitions receive their fair budget of CPU time
- use realtime, priority-based scheduling when the system is lightly loaded — this allows systems to use the same scheduling behavior that they do today
- make use of free CPU time from partitions that aren’t completely busy — this gives other partitions the extra processing time they need to handle peak demands and permits 100% processor utilization
• overlay the adaptive partitioning scheduler onto existing systems without code changes — applications and system services can simply be launched in a partition, and the scheduler will ensure that partitions receive their allocated budget

• dynamically add and configure partitions at runtime, enabling the system to adjust processor consumption in response to fault conditions or other scenarios

• guarantee that recovery operations have the CPU cycles they need to repair system faults, thereby improving mean time to repair for high availability systems

• stop malicious code from stealing all the CPU time through a denial of service attack

Partition inheritance

To further enhance performance, adaptive partitioning can provide partition inheritance. When a server process (for instance, a device driver or file system) executes requests on behalf of a client application, the client is billed for the server’s time. As a result, server processes can run with minimal CPU budget — only the budget required to perform background tasks. Moreover, systems designers don't have to reengineer the server budget when more clients are added.

Figure 4 — Adaptive partitioning prevents high-priority tasks from consuming more than their assigned CPU percentage unless unused CPU cycles become available in the system. For instance, tasks A and D can run in time allocated to partition 3 because tasks E and F don’t require the rest of their budgeted CPU cycles. With a fixed partitioning scheduler, this free time would be wasted.
Faster debugging and testing

Adaptive partitioning offers benefits throughout the product development cycle to speed design, implementation, and testing. Consider a typical design cycle that consists of a high-level design, a subsystem-level design, implementation and unit testing, and system integration and final verification.

At the high-level design stage, systems designers can use adaptive partitioning to define CPU budgets, allowing each development group to implement their own priority schemes and optimizations within their assigned budget. When subsequently testing their partition, a development group can load a simple program into other partitions to consume CPU. This technique effectively loads the system and causes the adaptive partitioning scheduler to enforce the assigned budgets. The development group can then test the operation and performance of code within their partition under a simulated worse-case load condition.

Adaptive partitioning can also simplify day-to-day testing and debugging. For example, in the unit testing phase, code defects will occasionally cause runaway conditions that bring debugging to a halt. In these situations, the system appears to be locked and the developer can recover only through a reset — thereby losing key debug and diagnostic information. But by creating a partition that guarantees CPU time for console login and remote debugging, developers can collect information critical to understanding and correcting these defects.

At system integration time, problems typically arise because of unforeseen runtime interaction and CPU contention. These problems occur largely because there is no effective way to "budget" CPU utilization across different subsystems developed by different design groups. Adaptive partitioning alleviates this problem at the design stage. Also, developers can configure the target system with a “debug tools” partition to ensure that system diagnostic information can be retrieved while the system is under test.

Increased system availability

When a hardware or software subsystem fails in a high availability system, automated recovery functions must return the system to a proper operating state. The faster these functions execute, the lower the mean time to repair (MTTR) and the greater the overall system availability. Adaptive partitioning helps by ensuring that CPU time is available for the fault detection and recovery functions.

In systems that typically run at very high CPU utilization, processes that monitor system health and report errors don’t get an opportunity to run in a timely manner. The CPU guarantees provided by adaptive partitioning address this problem and ensure that routine diagnostic functions run as intended. These functions can thus detect and report problems before the problems result in hard failures.

In the most severe cases, the user must intervene to revive a system. In these cases, the system must notify the user of the failure and provide some way of diagnosing the problem. Again, adaptive partitioning helps by ensuring the system has enough CPU cycles to alert the user and to provide guaranteed access to the user interface, be it a system console, remote terminal, or other method.

Minimum risk

Adaptive partitioning doesn’t require code changes, nor does it change the programming model or debugging techniques that designers are already familiar with. It thus offers immediate benefits with
little associated risk. Nonetheless, a fixed-cycle scheduler may be desirable in some situations. To address this requirement, a properly implemented adaptive partitioning scheduler will allow the system designer to configure a system with fixed partition budgets and no CPU time “borrowing.” System designers are thus free to choose the scheduling behavior that best meets their system requirements.

**Partitioning Plus Performance**

Embedded software is becoming so complex that, without some form of partitioning, system designers and software engineers will be hard-pressed to satisfy the conflicting demands for performance, security, time to market, innovative features, and system availability. An OS-controlled approach to partitioning goes a long way toward addressing these requirements by providing each subsystem with a guaranteed portion of CPU cycles, while still delivering the deterministic, realtime response that embedded systems require. Device manufacturers can, as a result, readily integrate subsystems from multiple software teams, allow new and upgraded components to run without compromising the behavior of existing components, streamline software debugging, and protect their systems from denial of service attacks and other network-based exploits. If the partitioning model also provides a flexible, efficient scheduler that allows partitions under load to borrow unused CPU time from other partitions, then manufacturers can realize these various benefits without having to incur the cost of faster, more expensive hardware.