MULTITASKING PARALLEL METHOD FOR HIGH-END EMBEDDED APPLIANCES

Embedded appliances such as high-end cell phones require not only high performance but also a performance guarantee. The authors demonstrate a performance guarantee framework using an asymmetric multiprocessing approach. They implemented the proposed method on a multicore processor using Linux. Evaluation results show that the method improves the performance guarantee while maintaining software compatibility.

Junji Sakai
Inoue Hiroaki
Sunao Torii
Masato Edahiro
NEC Corporation

Cell phones come with increasingly rich functions, comparable to those PCs had several years ago. You can use a cell phone to send e-mail with pictures, browse the Internet, and run Java applications with 3D GUIs. Several methods have attempted to improve embedded appliances’ general performance, such as raising the processor clock, extending the processor instruction set, adding hardware accelerators, and introducing multicore technology. With these hardware enhancements, embedded appliances’ total throughput would fit their performance needs.

Although the trend is for embedded appliances to follow PCs, they perform differently from PCs. PCs are general-purpose and essentially best-effort tools. That is, they can run various applications, but the applications’ performance isn’t guaranteed. For example, when you start a Web browser while a movie player is running, you might experience dropped frames in movie playback or awkward page scrolls of the browser screen. In contrast, cell phone applications are almost fixed, and we expect each application to perform correctly in any combination that conforms to the cell phone’s specifications. Moreover, even when an application hits a severe bug, the whole system shouldn’t freeze. This issue relates to robustness, but here we include it within the concept of the performance guarantee.

In general, performance guarantee is one of the most essential requirements for embedded appliances, and will only become more so as these appliances become more complex. Because embedded appliances require ever-improving processing performance, we need an architecture to fulfill the performance guarantee requirement. A multicore approach would let us achieve the performance guarantee by assigning specific jobs to each processor core. Low power dissipation and easy software migration are the other requirements for embedded appliances. The fact that parallel processing generally leads to lower power consumption also points to a multicore approach.

Performance guarantee and multicore

Symmetric multiprocessing is perhaps the best-known technology used in multicore approaches. In fact, many multiprocessor
servers and PCs use SMP. Researchers can use CPU affinity to separate real-time from non-real-time tasks, so the real-time performance should be guaranteed. Because the SMP operating system kernel is monolithic, however, there is still some interference among CPUs. Moreover, if a virus or bug causes a CPU to break down, other CPUs governed by the same SMP operating system instance might stop together. Other work reports that the cache coherency mechanism in an SMP chip consumes too much power and die area, making these chips not suitable for all embedded systems.

To further reduce interference among CPUs, you can run a separate operating system kernel instance on each CPU core in a multicore processor, where each CPU core is totally separate at the hardware level. This parallel model is called asymmetric multiprocessing. In AMP, a problem in one kernel on one CPU core won’t affect its neighbor, and the system will continue to operate even when some of the cores break down. From a hardware viewpoint, we can build an AMP chip using traditional single CPU cores, which are small in size and considered mature in quality.

To apply the AMP model to embedded appliances, we must resolve three problems:

- **Software compatibility.** Programs in an AMP system must use special communication APIs different from those in traditional single-CPU operating systems. Obviously, forcing programmers to modify applications before porting them to a multicore environment is undesirable.

- **Hardware resource management.** Because each operating system on each CPU core runs independently, it can erroneously grant simultaneous access to the same hardware resource, which might cause hardware inconsistencies and bring incorrect results.

- **Kernel memory consumption.** We might need a large memory area to run as many independent operating system instances as the number of cores. This is, of course, also undesirable.

Our multitasking parallel method resolves these issues.

**Multitasking parallel method**

Our multitasking parallel method is an AMP with some enhancements. As Figure 1 shows, in our method, an instance of a single CPU operating system runs on each processing element (PE), and a set of tasks for a single-CPU operating system runs on each operating system instance. We use an OS wrapper to resolve the software compatibility issue. A client-server method and execute-in-place technique handle the hardware resource and kernel memory issues.

**OS wrapper**

Because a traditional, single-CPU operating system runs on each PE in our multitasking parallel method, the software environment is naturally compatible with traditional operating systems, except for communication among PEs. The OS wrapper is a software layer laid on top of operating system kernels that performs interprocess communication (IPC) among PEs. It’s responsible for keeping IPC APIs compatible with traditional APIs. In other words, applications on the OS wrapper can use the same set of APIs for both intra-PE and inter-PE communications. The OS wrapper catches the API calls and passes the information to the target if needed—that is, the OS wrapper “wraps” the original operating system kernel.

We took measures to implement the OS wrapper technique at three points.

First, we allocated a memory region outside the operating system kernels to hold IPC-related information, and used userland modules to manage the information. This lets us retain information from permanent IPC objects, such as shared memory objects, beyond a single IPC transaction, and manage information shared among PEs within mutually independent operating system instances.

Next, because we schedule tasks related to the IPC operations in userland code, we built a mechanism that controls waiting tasks’ restart using a special proxy task.

Finally, to improve performance, we built the OS wrapper making the best use
of our multicore chip architecture, which includes the inter-PE interrupt mechanism and fast mutual exclusion control using the on-chip shared memory.

We implemented our multitasking parallel method on the Linux operating system, which developers are adapting more and more frequently to embedded systems such as cell phones. We designed the Linux version of the OS wrapper to provide the functions of System V IPCs (message queues, semaphores, and shared memory objects) and Unix domain sockets that are compatible with those of traditional, single-CPU Linux. Although processes on different PEs can use Internet domain sockets to communicate with each other, the Linux OS wrapper is useful because it lets us port most traditional Linux applications, which typically use Unix domain sockets or System V IPCs rather than slower Internet domain sockets, to a multicore environment without modification.

Figure 2 is a block diagram of the Linux implementation of our OS wrapper. It consists of application libraries, device drivers, and proxy processes. The figure illustrates an example situation in which application A on PE0 is sending a message to application B on PE1. When application A calls the API `msgsnd()` with data to be sent, it’s hooked by an application library (`ipc.lib`). The library sends the data to PE1 through an inter-PE interrupt device driver (`ipi dd`), which performs the most primitive signal transmission between PEs. On the PE1 side, application B calls the API `msgrcv()` and is blocked. When an inter-PE interrupt to PE1 signals the proxy process on PE1 that data has arrived, the proxy process uses a Linux signal to resume the blocked application B on behalf of the Linux kernel and finally transmits the incoming data to application B. Thus, the OS wrapper realizes IPCs across PE boundaries.

As we mentioned, we designed and implemented most of the OS wrapper within the userland space. For Linux versions of the OS wrapper, this design also makes it easier to keep up with future kernel version changes. So far, we’ve often seen nontrivial changes of internal structures of the Linux kernel with minor and major version changes, but have rarely seen changes in the kernel interfaces, which are open to device drivers and applications. The userland implementation will also help us realize communications between different types of operating systems in a multicore chip, such as RTOS and Linux. Our current implementation is mainly composed of userland libraries and processes, with only a few device drivers in the kernel space. The current implementation is about 4,000 lines of codes in total, with no modification of the Linux kernel itself. Our OS wrapper mechanism is rather portable.

Hardware access server
To address the hardware resource-management issue, we could improve the kernel so that all kernel instances cooperatively manage hardware resources. However, this would be difficult and would spoil the advantage of having little interference...
between PEs. Instead, we introduced another mechanism, the **hardware access server**.

This mechanism basically relies on the client-server model. A hardware access server task makes the actual access to a hardware resource and forbids other tasks from accessing the hardware directly. Other tasks, which we regard as clients, request that the server access the hardware by proxy and send the results back to the clients. Because clients and servers should communicate using the IPCs covered by our OS wrapper, any application on any PE can safely and correctly use hardware resources. We also prepare client libraries, which act as a bridge between a client-side communication module and the application body.

We can implement this mechanism in Linux without much difficulty using existing techniques, such as the X Window System to manage input devices and display output. Any X-based GUI application can receive input key events and draw output results on a screen, no matter which PE they’re on. Similarly, we can use a network file system server for storage devices such as Flash ROM, and network address translation services for external network access.

The hardware access server mechanism introduces two types of possible overhead: overhead due to inter-PE communication, and overhead due to the detour when accessing hardware via the server. For the first type, we consider the performance decline permissible for general applications. For example, our first evaluation using a set of test programs, including GUI and file operations, normally shows a 5 percent (at most 10 percent) performance degradation when run on a PE other than the hardware access servers. One reason for the small degradation is the high speed of the on-chip, inter-PE communication. The second type of overhead is a necessary expense in maintaining the hardware access order in a multitasking environment. Generally, the hardware access server mechanism is beneficial, considering the multitasking parallel method’s advantage, including load balancing of main and I/O processing.

Still, in some cases the hardware access server mechanism’s overhead can’t be ignored. For example, some multimedia processing uses hardware resources such as a digital signal processor (DSP) or other hardware accelerator. Multimedia applications must transmit and receive considerable quantities of data to hardware resources without pause, so require both low latency and high throughput. For such cases, we introduce a **direct access** method that lets the device driver on each PE access the hardware resources directly.

In the direct access method, we insert arbitration codes in the device driver layer or in a userland layer to guarantee the mutual exclusion of hardware access. Take, for example, the decoding of a movie by DSP middleware. On each PE we install a DSP driver that not only controls the DSP middleware’s decoding operation but also performs arbitration among the DSP drivers on all PEs. The arbitrators in all the DSP drivers cooperatively restrict the number of simultaneous DSP access to one, considering priority. We’ve adopted this direct access method for DSP and video overlay hardware with adequate performance.

**Kernel text sharing**

A basic feature of our multitasking parallel method is that it runs the same type of operating system on each PE. Because their text areas are read-only and are the same for all PEs, we can reduce kernel memory consumption by having the PEs share the kernel text areas. Embedded systems often deploy the execute-in-place (XIP) software technique. In XIP, the CPU fetches its instructions directly from ROM to reduce the consumption of valuable RAM area. We’ve made slight modifications

---

**Figure 2. The OS wrapper implemented in Linux.**

---

---

Authorized licensed use limited to: Politecnico di Milano. Downloaded on January 29, 2009 at 06:58 from IEEE Xplore. Restrictions apply.
to the XIP memory-mapping mechanism so each kernel runs on the same ROM area. Our Linux implementation shows that the total kernel memory consumed by three PEs is no larger than 1.3 times that consumed by a single PE. The ratio varies, of course, depending on the application, but we think that this feature is essential for our multitasking parallel method.

**Evaluation on MP211**

MP211 is a multicore application processor for embedded systems that supports the multitasking parallel method. As Figure 3 shows, MP211 is a system on chip (SoC) that integrates three ARM926 cores, one DSP core (NEC Electronics’ SPXK6), an SDRAM controller, on-chip SRAM, and several intellectual property cores such as a graphic accelerator, an image processor, and peripheral interfaces. We expect high performance using these multiple processor and other IP cores. We also expect low power consumption because of the design specification that the processor cores’ maximum operating frequency be as low as 192 MHz at the relatively low supply voltage of 1.2 volts. MP211 is fabricated with a 0.13-μm CMOS process.

The MP211 architecture’s most prominent feature is its categorized bus structure, which supports the performance guarantee in embedded systems. Generally, CPU cores require low latency for their main memory access. However, the DSP and other media-processing IP cores need high bandwidth. MP211 has a set of quadruple buses for external SDRAM access with different characteristics, and each IP core is connected to one of the buses according to its memory access needs. The smart SDRAM controller built into MP211 schedules the incoming memory access requests so that it can make the most of DDR SDRAM performance. This feature keeps the CPU memory access latency low, even when rich multimedia applications are running on the DSP.

Another feature of MP211 is its low-power design. A dynamic clock-frequency-control technique reduces the operating power consumption by automatically lowering clock frequency when it detects that the system is in idle status. Moreover, the whole logic of MP211 is split into two internal power domains that system software can turn on and off using the power management unit (PMU) built into the chip, and the on-chip SRAM is designed with a low-leakage, thick-gate oxide process. These features help reduce stand-by power consumption.

**Evaluation of performance guarantee**

We first evaluated the most interesting aspect of our multitasking parallel method: the degree of performance guarantee. We constructed two sets of applications that we believe will be used in high-end cell phones in the near future.

**Digital terrestrial broadcasting television (DTV) viewer and news reader.** In the first scenario, two large applications—a DTV viewer and a news reader—are running concurrently. The DTV viewer consists of DSP middleware for decoding H.264 video and MPEG2AAC audio data and a CPU application that demultiplexes the received stream and adjusts the playback timing. The news reader consists of a Java program that periodically collects articles in rich site summary (RSS) format from external servers according to the user’s interest and converts them into an HTML page, and an HTML browser that displays the output. This application mix includes two types of applications with different characteristics: the DTV viewer contains some threads requiring real-time performance, whereas the news reader’s job is CPU-bound.

To evaluate the performance guarantee, we observe the DTV viewer playback when we operate the news reader using two MP211 versions—the single-PE version (all tasks run on one PE) and the 3PE version (tasks are divided among three PEs). As Figure 4 shows, we separate tasks in the 3PE version as follows:

- PE0 takes basic functions in the whole application mix,
- PE1 takes the DTV application to isolate real-time threads from others, and
PE2 takes the heavy Java application so it can use the whole CPU resource exclusively. Each PE runs at 150 MHz.

The single-PE version exhibits obvious brief interruptions of audio and video playback about once every two seconds on average. The 3PE version, however, works smoothly and we can’t see or hear any

Figure 3. The MP211 multicore application processor.

Figure 4. Evaluating the multitasking parallel method on an MP211 with three processing elements.
interruptions. Figure 5 shows the jitter of audio thread intervals for the single-PE and 3PE versions. In the single-PE execution, the CPU-bound news reader job disturbs the audio thread periodicity and the thread interval sometimes becomes very large, which substantiates our observation of interruptions. In contrast, periodicity is fully sustained for 3PE execution.

Figure 6 shows the trace analysis output of kernel context switching for the single-PE execution. The news reader application sometimes seizes the PE’s computation resource for about 150 ms, preventing the DTV real-time threads from performing normal operations, which must run once every 20 or 40 ms.

**Binary download.** In this scenario, we intend to show the system robustness. As Figure 7 shows, we download a specially formatted document and the accompanying viewer software from an external server. We then install the downloaded viewer on one of the PEs and start displaying the document, but a critical bug in the viewer causes the system to shut down.

In single-PE execution, the entire system stops its normal operation when the system hits the viewer bug, and we must reboot the system. In 3PE execution, however, a dedicated PE separated from the system core-function PEs executes the downloaded application. When the downloader PE hits the bug and shuts down, one of the system’s core-function PEs detects the abnormal situation and safely reboots the downloader PE. The system’s core functions remain operational, and their performance is guaranteed. Results from a demonstration system that we built for this scenario prove that our multitasking parallel method can improve system robustness.

**AMP issues**

The fact that our evaluation applications work correctly on multiple PEs without...
any source code modifications suggests that our OS wrapper mechanism realizes the software compatibility as expected. We only had to choose appropriate conventional applications, assign processes to the three PEs, and make a few adjustments to the interprocess communication parameters, such as designating the X server location in the system. These were quite easy jobs.

As for access to the shared resources, the applications in our examples access the X server and the external hosts seamlessly. And finally, we suppressed the memory consumption of separate kernels by adopting XIP technology, so we can run the same application mix in the 3PE environment with the same amount of SDRAM as in single-PE environment.

From these evaluation results, we can say that our multitasking parallel mechanism achieves the performance guarantee needed for embedded appliances and makes the most of conventional hardware and software properties.

Advanced desktop computing systems already require multicore technology, but it’s also increasingly necessary for the tiny digital appliances that fit in our hands. However, conventional parallel processing techniques don’t meet various needs specific to embedded systems. The combination of our multitasking parallel method and our multicore technology realizes both the performance guarantee and practical reuse of hardware and software properties. We believe it will be one of the most suitable approaches for embedded appliances in the near future. We plan to apply the proposed method to real embedded products for more realistic evaluations and to develop new technology to enhance the robustness and reliability of embedded appliances through multicore architectures.

Acknowledgments
We thank Masao Fukuma, Masakazu Yamashina, and Naoki Nishi for their encouragement. We also thank all the MP98 project members for their helpful discussion and great support in implementing and evaluating our proposal. Finally, we’re also grateful to the MP211 development team members at NEC Electronics for their cooperation and support.

References

**Junji Sakai** is a principal researcher at the System IP Core Research Laboratories, NEC Corporation. He has been involved in the R&D of embedded multicore software for more than 10 years. He received an MS in information science from Kyoto University. He is an affiliate member of the IEEE Computer Society.

**Inoue Hiroaki** is an assistant manager at the System IP Core Research Laboratories, NEC Corporation. His research interests include hardware and software codesign for multiprocessor systems. He received a master’s degree in engineering from Keio University. He is a member of the IEEE and ACM.

**Sunao Torii** is a principal researcher at the System IP Core Research Laboratories, NEC Corporation. His research interests include multicore and low-power processor architecture and systems. He received an MS in computer sciences from Keio University.

**Masato Edahiro** is a research fellow at NEC Corporation and a visiting professor in the Department of Computer Science at the University of Tokyo. His research interests include graph/network algorithms, combinatorial optimization, and parallel algorithms for many cores. Masato received a PhD from Princeton University. He is a member of the IEEE, the Institute of Electronics, Information, and Communication Engineers, and the Information Processing Society of Japan.

Direct questions and comments about this article to Junji Sakai, 1753, Shimonumabe, Nakahara-ku, Kawasaki, Kanagawa 211-8666, Japan; jsakai@computer.org.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/csdl.