Single Chip Coherent Multiprocessing

The Next Big Step in Performance for Embedded Applications

April 2008

MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043
(650) 567-5000

One Processor versus Many

Driving the performance of an individual processor to the limits of the possible in a given implementation technology is never easy or efficient. The tried and true methods of faster clocks, deeper pipelines, and bigger caches all have silicon area and power dissipation costs that get well into diminishing returns to get that last 10% of performance. There are times when there is no alternative but to turn up the clock and upgrade the power and cooling subsystems; but when a workload can be split across multiple processors, the limits to maximum total performance are pushed back, and the design of the processing elements themselves can be made simpler and more efficient.

While you see this approach today in PCs, servers and workstations, this is just as true in embedded SoC designs. In fact, many embedded SoC designs today make use of multiple processors, but do so in an application-specific or “loosely coupled” manner. Until recently, SoC design options for software-friendly multiprocessing were severely limited. However, with the advent of SoC design components such as the MIPS32® 1004K™ Coherent Processing System (CPS), on-chip Symmetric Multiprocessing (SMP) under a single operating system has become a real design option, and system architects need to understand its promises and limitations.

Many Paths to Parallelism

Exploiting parallel processors requires parallel software, and “parallel programming” is a model that creates some apprehension with software engineers because not all existing code was written for a parallel processing platform. But there are several paradigms for parallel software, some of which are already very familiar to software designers, even if they don’t necessarily think of them as such.

Data-Parallel Algorithms

Data-parallel algorithms are ways of attacking a single basic computational problem by carving up the data set to make use of more than one processor, ideally up to a large number of CPUs. The textbook case of a large data set is a large input file or data array, but in embedded systems, it can mean high I/O and event service bandwidth. In some SoC architectures, multiple sources of input data, such as network interface ports, each of which needs to be handled the same way, can be statically assigned to multiple processors running the same driver/router code to make for natural data-parallelism.

When the power of multiple processors must be brought to bear on a single data array or single input stream, data-parallel algorithms that “divide and conquer” the data are often used, such as the simple example shown in figure 1. Such algorithms are generally sub-optimal on a single processor, but make up for their inefficiency with scalability to exploit more computational bandwidth. They “make up for it in volume.” These algorithms have been shown to be the most scalable approach to parallel computing, but converting a working sequential program to a data-parallel algorithm may be trivial, difficult, or impossible, depending on factors such as the dependency characteristics of the program.

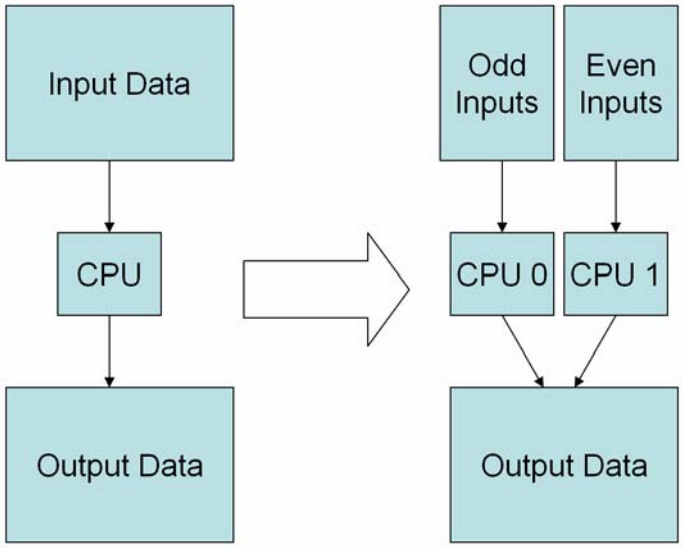

Fig 1 – *Data-Parallel* Programming Model

A system designer looking for higher performance for an existing application would most likely look to explicitly implement data-parallel algorithms if the vast bulk of computational work in the application is done in a relatively small number of long runs of regular computational loops.

The emergence of multi-core “x86” chips for PC, workstation and server processors has generated research and investment in a new wave of libraries and toolkits to enable, and more easily exploit, parallel algorithms on modest numbers of processors. Many of these are open-sourced and portable to embedded architectures such as MIPS. OpenMP extensions to *gcc* for data-parallel C/C++ as well as FORTRAN are becoming a part of the standard GNU compiler collection.

Control-Parallel Programming

Another paradigm, which will be referred to here as *control-parallel* programming, is to split the work of a program by task, rather than by input. If an automobile factory where 100 workers are each given a car to build can be seen as a metaphor for a 100-way data parallel algorithm, the analogous metaphor for a control-parallel program would be a factory with a single assembly line consisting of 100 stations, each staffed by a worker performing a different task that is 1/100 of the assembly work. A simple example of a two station implementation is shown in figure 2. The assembly line approach is generally more efficient, but there is a limit to how far one can divide up the work of assembling a single car. This limitation is significant for scientific codes that one would like to scale to thousands of processors, but not generally an issue with modestly parallel SoC architectures for consumer applications.

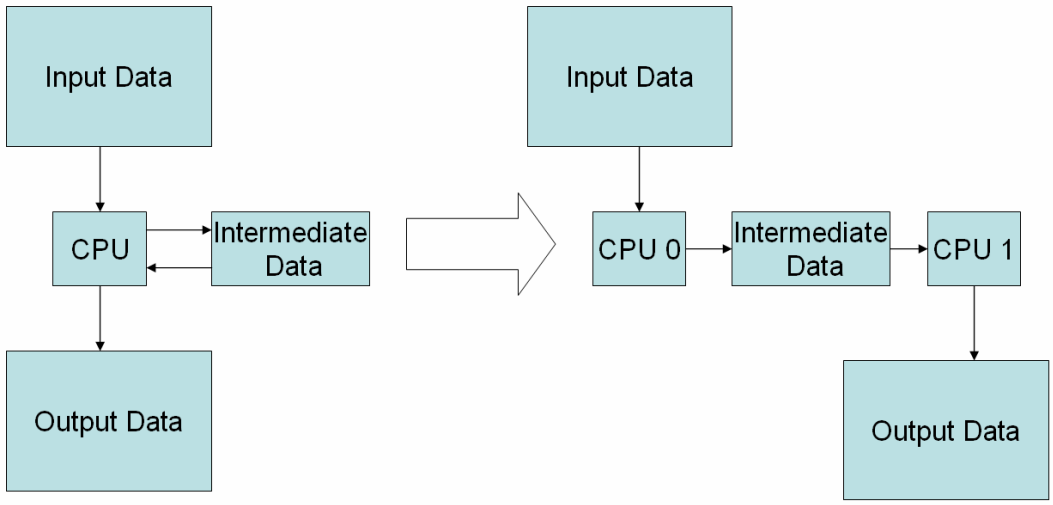

Fig 2 – *Control-Parallel* Programming Model

Even without taking parallel processing into consideration, software engineers will often break programs up into phases. It makes for easier coding, debugging, and maintenance by teams of programmers, and it reduces pressure on instruction memory and caches. In many cases, the control-parallel decomposition of a problem has already been taken to the level of OS-visible tasks. The single command “cc” on a UNIX-like system invokes, sequentially, a C language pre-processor, a compiler, an assembler and a linker. On an SMP multiprocessor, several of these can be run simultaneously, with each successive program using the output of the previous phase as its input, using files or, better still, the software “pipes” that have long been a feature of UNIX-like operating systems, including Linux.

When decomposition into independently run tasks hasn’t already been done, some software engineering must be done to make the phases of an application visible to the operating system and the underlying hardware, and to explicitly pass data from one task to another when its “ownership” passes from one phase to another. But there should be no need to rethink or rework the algorithms of the constituent phases, as is generally required for a data-parallel decomposition. Coarse-grain task decomposition can be done in terms of processes communicating via files, sockets, or pipes. For finer-grained control, the POSIX thread API, *pthreads*, is widely used, and is supported by a broad range of operating systems, including Linux, Microsoft Windows, and many real-time operating systems.

The more there is to do, the more there is to do concurrently.

Complex, modular, multitasking embedded software systems will often exhibit “serendipitous” concurrency, such as that illustrated in figure 3, even if it was not a design objective. The overall mission of the system may involve the operation of multiple tasks, each of which has a distinct responsibility, responding to a distinct set of inputs. Without a time-sharing operating system, these tasks would each have to run on a separate processor. On a time-sharing uniprocessor, they run in alternating time-slices. On a multiprocessor with an SMP operating system, they can run concurrently across as many processors as are available.

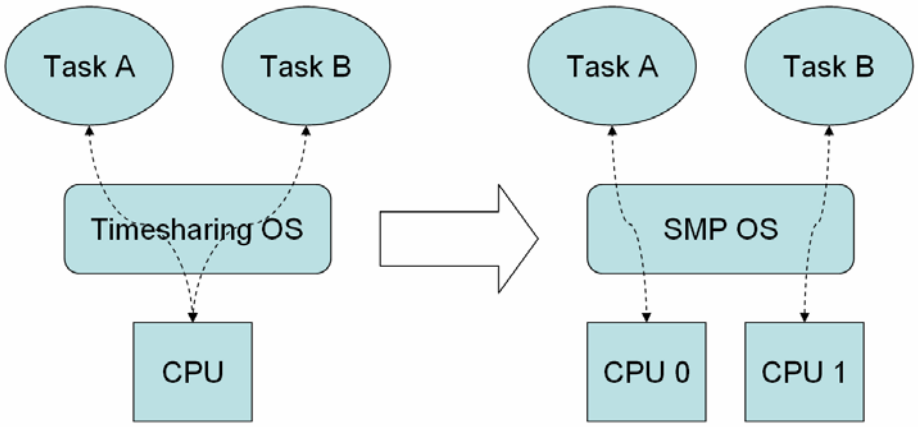

Fig 3 – Concurrent Multitasking

Distributed Processing

Another form of parallel processing that has become so commonplace that it is sometimes not even thought of as “parallel” is distributed computing, of which network client/server models are by far the most common paradigm. Client-server programming is basically a form of control-flow decomposition. Rather than performing all of a computation itself, a program task connects and sends work requests to one or more specialized tasks in a system which are designated to perform specific jobs. While client/server programming is most commonly done across LANs and WANs, communications between tasks within an SMP SoC follow the same paradigm. One can use unmodified client/server binaries communicating by TCP/IP via on-chip or null “loopback” network interfaces, or more efficiently by using local communications protocols that pass data buffers in memory.

In practice, any of the above techniques may be used alone, or in combination, to leverage the power of an SMP-based platform for a given application. One could even construct a data-parallel array of distributed SMP servers, each of which implements a control-flow pipeline. But for such a scheme to be efficient, there would need to be a very large workload and data set.

System Software Support is Critical

In SoC systems where parallelism by static physical decomposition of tasks onto processors is possible (e.g. one processor core per input port), the assignment of parallel tasks to processors can be done in hardware. This reduces software overhead and footprint, but provides no flexibility.

Similarly, if an embedded application can be statically decomposed into clients and servers communicating across an on-chip interconnect, the only system software required to tie the system together would be message-passing code that implements a common protocol between processors. The message passing protocol provides some level of abstraction that can enable configurations with more or fewer processors to run a common base of application code, but for any given configuration, the load balancing between processors is as static as the hardware partitioning. For more flexible parallel system programming, software distribution of tasks across a multiprocessor system with shared resources is needed.

Flexibility and Adaptability of SMP Systems

As the name implies, SMP operating systems have a “symmetric” view of the system. All processors see the same memory, the same I/O devices and the same global operating system state. This makes migration of programs from one processor to another extremely simple and efficient, as shown in the simple example in figure 4, and makes load balancing easy. With no additional programming or system administration, a set of programs that multi-tasks on a single CPU using time-slicing will run concurrently on the available CPUs of an SMP system. An SMP scheduler, such as that of Linux, will switch programs on and off of processors so that all make progress in a fair manner.

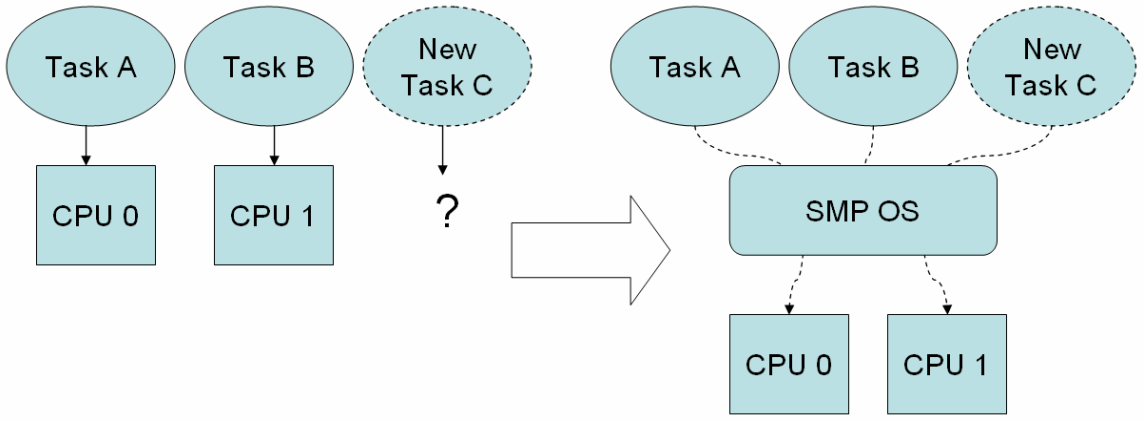

Fig 4 – SMP Task Distribution Across Multiprocessor Resources

A Linux application that runs as multiple processes needs no modification to take advantage of SMP parallelism. In most cases, no recompilation is required; the exception being binaries that were statically linked with non-thread-safe libraries.

An SMP Linux environment provides a number of tools that allow a system designer to tune the way tasks share the available processors. Tasks can have their priorities raised and lowered, and can be restricted to run on arbitrary subsets of processors. With appropriate kernel support, they can request the use of different real-time scheduling regimes.

UNIX-like operating systems have always allowed applications to have some control over the relative scheduling priority of tasks, even in uniprocessor time-sharing systems. The traditional *nice* shell command and system call have been augmented in Linux with more elaborate mechanisms to manipulate the priority of tasks, groups of tasks, or specific users of a system, should it be necessary to second-guess the OS.

Additionally, in multiprocessor configurations, every Linux task has a parameter that specifies what set of processors may schedule the task. By default, that parameter is the full set of processors in the system, but, like priority, this CPU affinity can be controlled either by the *taskset* shell command, or by explicit system calls to manipulate the “CPU affinity” of tasks.

Enabling SMP

An SMP system paradigm requires that all processors see all of memory at the same addresses. For simple, low performance processors, this isn't too difficult to accomplish. One simply puts the instruction fetch and load/store traffic of all processors on a common memory and I/O bus. This simplistic model breaks down pretty quickly with increasing numbers of processors however, as the bus quickly becomes a performance bottleneck. And even in uniprocessor systems, the bandwidth requirements for instructions and data of high-performance embedded cores dictate that cache memories be used between main memory and the processor.

A system with independent per-processor caches is no longer naturally SMP. When one processor's cache contains the only copy of the most recent value of a location in memory, there's a basic—and dangerous—asymmetry. Cache coherence protocols must be added to the system to restore that symmetry. In very simple systems, where all processors are connected to a common bus, it is sufficient for all cache controllers to monitor the bus to see which cache owns the latest version of a given memory location. In more advanced systems, such as the MIPS32 1004K CPS, processors are connected to memory using point-to-point connections to a switching fabric rather than a bus. Cache coherence thus requires more sophisticated support. The 1004K coherence manager imposes a global order on memory transactions and generates the necessary intervention signals to maintain cache coherence among multiple 1004K processor cores.

The 1004K processors thus see a symmetric view of memory. An SMP operating system such as Linux can freely migrate tasks and dynamically balance processor loads.

In an embedded SoC, a substantial portion of overall computation can be spent in interrupt service. This implies that good load balancing and performance tuning requires control, not only of where program tasks are allowed to run, but also where interrupt service is to be performed. The Linux operating system has an "IRQ affinity" control interface that allows users and programs to specify which processors are to be used to service a given interrupt. To be usable, this interface requires that the underlying system hardware provide a means to selectively route interrupts to processors. The 1004K global interrupt controller provides this capability for the 1004K CPS.

Cache coherence infrastructure is useful, not only between processors for symmetric multiprocessing, but between processors and I/O DMA channels. While RISC architectures such as MIPS32 have features to support software-based I/O coherence, this requires that DMA buffers be processed by the CPU before or after each I/O DMA operation. This processing has a measurable performance impact on I/O-intensive applications. In the 1004K CPS, connecting I/O DMA to memory via an I/O coherence unit allows DMA traffic to be ordered and integrated with the coherent load/store flow, eliminating the software overhead.

Paying the Piper – And Getting a Rebate

In SoC design, as elsewhere in life, there's no such thing as a free lunch. The 1004K coherence manager imposes order and sanity on memory traffic between processors, I/O, and memory, but in doing so, it adds cycles to the memory access time experienced by the processor. Ordinarily, this would always result in additional lost processor cycles when the pipeline stalls, waiting for the cache to be filled with instructions or necessary data. But the 1004K platform implements the MIPS multi-threading architecture first pioneered in the MIPS32® 34K® core family, which allows a single core to execute multiple concurrent instruction streams.

Each individual core within the 1004K CPS includes support for two hardware threads via "VPEs," virtual processing elements that look just like a CPU to operating system software. The two virtual processors share the same cache and functional units, and interleave their execution on the pipeline. If one VPE is stalled waiting for a cache fill from memory, the other can execute to keep the pipeline busy. In effect, the 1004K processor's multi-threading capability allows it to take back cycles otherwise lost to the latency of the coherent memory subsystem.

Since the 1004K processor VPEs look like full-blown processors to software, up to and including having independent interrupt inputs, the same SMP operating system logic that manages multiple cores can be exploited to manage their constituent VPEs. At the highest level of system administration, a dual-core 1004K system with all VPEs active looks to be a 4-way SMP system. Software that has been written or configured to exploit SMP can naturally exploit multi-threading, and vice versa.

While the view of system resources remains symmetric, it is true that two threads competing for the use of a single processor pipeline will achieve lower performance than two threads running on independent cores. This situation has existed for years in server systems, where coherent clusters of multi-threaded CPUs are not unusual, and the SMP Linux kernel for the 1004K is equipped to do the necessary load-balancing optimizations. If optimizing for power consumption, the scheduler can load work onto the virtual processors of one core at a time, so that the others can remain in a low-power state. If optimizing for performance, it can spread work across distinct cores first, only loading up multiple VPEs per core once all cores have an active task to run.

Conclusions

On-chip multiprocessing can be exploited in a number of ways to achieve high SoC performance. Static decomposition of work, by input data or processing function, can be very efficient, but is also very inflexible. SMP platforms and software provide a very flexible high-performance computing platform that can deliver significant speedup relative to single processors, often with little or no modification to application code. Multi-threading is highly complementary to SMP parallelism and allows for the highest possible utilization of pipeline resources in each processor. The MIPS32 1004K Coherent Processing System brings together MIPS multi-threading and coherent SMP in a single IP block to provide scalable, high-density embedded computing power.