Debugging Real-Time Multiprocessor Systems

Class #264, Embedded Systems Conference, Silicon Valley 2006

Jakob Engblom
Virtutech
Norrtullsgatan 15
SE-113 27 Stockholm
Sweden
e-mail: `jakob.engblom@virtutech.com`

Technology trends are causing a paradigm shift in how computer systems are designed: Instead of steadily getting faster single processors, the entire semiconductor industry is turning to multiprocessor designs to improve performance. Future real-time and embedded systems will be using multiple-processor hardware, and the software is expected to adapt to the situation. Writing parallel software programs is known to be very difficult and fraught with unexpected problems, and now parallel programming is expected to go mainstream.

This class will discuss how to debug parallel software running on top of multiprocessor hardware, and the types of errors that occur due to parallelism. A range of techniques and tools are covered and the goal is to prime real-time and embedded software developers for multiprocessor integration and debugging.

1 Introduction

Since the early 1980s, embedded software developers have been taking advantage of the steady progress of processor performance. You could count on new processors being made available that would increase the performance of the system without disturbing the software structure overly much. If you used a 500 MHz processor last year, you could buy a 1000 MHz processor next year. You would get a doubling in your application performance without any other work than designing in a new, software-compatible chip.

This performance increase was often factored into application and system design: The software designers planned and implemented application sets that were too heavy for the currently available processors, counting on a new, faster processor to come on the market and solve their performance problems before the product got to market. Such a strategy was necessary to stay competitive.

This comfortable state was driven by the steady progress of the semiconductor industry that kept packing more transistors into smaller packages at higher clock frequencies, enabling performance to increase by both architectural innovations like more parallel pipelines, by adding resources like on-chip caches and memory controllers, and by increasing the clock frequency.

The 32-bit PowerPC family offers a typical case study on this progression. Starting with a 603 processor, users have been able to upgrade with full application-software compatibility to the “G3”/750 processor series (rising from 166 to 1100 MHz over time). Next came the “G4”/7400 series, and then the 64-bit “G5”/970 series, continuing to provide software engineers a performance increase in a single processor.

However, in 2004, it became clear that the progress in single-processor performance began slowing considerably. Due to a number of chip design and manufacturing issues, the clock-frequency increase slowed down, and we could not expect to get twice-as-fast processing from clock speed increases. Instead, the semiconductor industry turned to parallelism to increase performance. Using multiple processor cores (known as a multicore) on the same chip, the theoretical performance per chip can increase dramatically, even if the performance per core is only improving slowly. Also, the power efficiency of a multicore implementation is much better than traditional single-core implementations, which is a factor as important as absolute performance [1].

Consequently, every high-performance processor family is moving to multiprocessor designs. Freescale has announced the PowerPC 8641D, with two G4-class cores on one chip. IBM has the PowerPC 970MP, with two G5 cores. Intel is pitching dual-core Pentium-M machines to the embedded market, and PMC-Sierra has been selling dual-core RM9200 MIPS64 processors for some time. Cavium has announced up to 16-way parallel MIPS64-based processors. Parallel machines are here in force.

The move to parallel hardware, however, creates problems for software developers. Applications that have traditionally used single processors will now have to be parallelized over multiple processors in order to take advantage of the multiple cores. Programs that used to run on single processors will have to run on multiprocessors. Apart from the act of creating parallel software, debugging is greatly complicated by the combination of parallel execution and tighter packaging [2][8].

This will be the greatest challenge to the embedded software industry for a very long time. Both tools and thinking need to change. The level of conceptual change is comparable to the introduction of multitasking operating systems, virtual memory and object-oriented programming.

1.1 Multi-what?

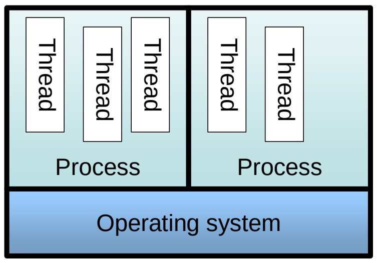
Multiprocessing systems can be built in a number of ways, and there are several terms referring to various forms of packaging and implementing multiprocessor systems.

- **Multitasking** means that a single processor can run several different software tasks, scheduled by a (real-time) operating system.
- A **multiprocessor** is any computer system using more than one processor to perform its work.
- A **shared-memory or symmetric multiprocessor** (SMP) is a design where the processors in a multiprocessor share the same memory, and all can access the same data and run the same tasks. In contrast, an **asymmetric multiprocessor** (AMP) uses several processors with their own local memories, often using different architectures.
- A **multicore** processor is a single processor chip containing multiple processor cores, which can be identical (as on the Freescale MPC8641D) or of different types (as on a Texas Instruments OMAP). Essentially, it is a multiprocessor on a single chip. There is some debate on precisely what deserves to be called multicore [4], but in this paper we suppose that anything with more than one processing core on a single chip qualifies.
- **Hardware/symmetric multithreading** (S/MT) is a technique where a single processor core is used to execute several parallel threads of computation. This provides less performance than a true multicore design, but also has a much lower implementation cost. Examples include the IBM Power5 and Intel Pentium4 HyperThreading.
- A **chip multiprocessor** (CMP) is a processor employing multithreading and/or multicore techniques to create a parallel computer on a single chip. An example is the Sun "Niagara" UltraSparc T1. CMP designs are typically SMPs, as they mainly come from the mainstream PC/server market.

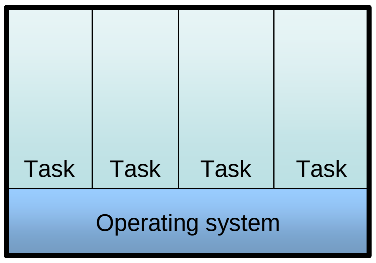
In the rest of this paper, we will use the term "multiprocessor" to denote any computer system containing more than one thread of control in hardware.

1.2 Software Structure

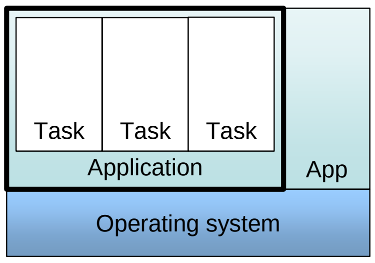
On the software side, we prefer to use the generic term "task" to mean a single thread of computation. We want to avoid confusion between "process" and "processor." A number of tasks can share memory in order to implement an application, and this is really what is meant by a parallel program: A number of tasks cooperate to perform the function of a particular application.



Desktop/Server model: each process in its own memory space, several threads in each process with access to the same memory



Simple RTOS model: OS and all tasks share the same memory space, all memory accessible to all



Generic model: a number of tasks share some memory in order to implement an application

Many operating systems use the term “process” to mean a single memory space in which one or more “threads” can run. Between processes, memory is not shared. Commonly, there is only one thread of computation inside a process, and people tend to use “process” to mean a single thread of execution.

2 Programming Parallel Machines

There are several ways in which to write code to run on a multiprocessor machine. The most important distinction is between *message passing* and *shared memory* styles. In message-passing, communication is explicitly programmed, while shared-memory makes communication implicit in the reading and writing of variables. Note that message-passing programming can make sense even on a shared-memory machine, where messages are implemented using shared memory.

Classic supercomputing (weather prediction, astronomy simulations, airplane design) typically uses sequential languages like C or FORTRAN with a library (pthreads) or compiler directives (OpenMP, MPI) to divide a computation task up into parallel tasks. Shared memory and message passing are both in use, depending on the nature of the machines used. This programming tradition usually creates programs that use a number of tasks in parallel to solve a small part of a single computation problem. Typically, parallelism is applied to single kernel loops. This model works well for the target domain, and supercomputing computations have been successfully parallelized to use 10,000s of processors in parallel.

In the desktop and server market, commercial multitasking and multiprocessing workloads are programmed using sequential languages (C, C++, Ada, Java, COBOL), using libraries, OS APIs or language support. Synchronization and communication can be provided as part of the language (Java and Ada), the operating system API, a library like pthreads or compiler directives like OpenMP and MPI. To contrast with supercomputing, parallel tasks are typically not homogeneous, but rather perform quite different computations based on a common data set. The most successful class of parallel programs in this space is servers, where each task handles an individual independent session with clients. Another successful example is using a task to run a user interface in parallel to the core application logic.

OpenMP is proving to be a popular way to provide parallel programming support for new machines. For example, the ARM MPcore has an OpenMP implementation available, and Microsoft provides it for their Xbox 360 game console [6][16]. Another trend is that virtual machine-based languages like Java and C# have started to provide support for managing parallelism as part of the virtual machine definition.

There are also parallel languages which go beyond the simplistic approach in Ada, Java and C# to make parallelism an integrated feature of the language. OCCAM is designed for supercomputing, where specific computation parts can be specified to execute in parallel (using constructs such as parallel loops). Erlang is an interesting language designed for control-plane applications, using hundreds or thousands of tasks each with their own local storage and communicating using message-passing. Tasks are used as the primary means of program structuring — rather like objects are used on object-oriented programming.

Writing parallel programs seems to be easier where large data sets are being manipulated, like supercomputing and databases. Parallelizing programs with smaller data sets is harder by experience. However, there is one mitigating factor in the current move to multicore implementations: Onboard a multicore chip, communication is much faster than in a traditional multi-chip multiprocessor. This helps programmers write efficient programs, and should allow beneficial parallelization of more types of programs.

In this paper, however, we will mainly consider the case of using C or C++ with shared memory, as that is the model with the most subtle debugging problems, and the one that most embedded programmers will encounter when multicore processors replace old single-core processors at the core of their products.

Overall, the move to multiprocessing constitutes a break with the mainstream (and embedded) software-development tradition. Most programmers have been taught how to program single-threaded programs. Few real-time operating systems support SMP processing. Debuggers (with a few exceptions) operate under the assumption that we are debugging a program that is serial, repeatable and self-contained.

3 Embedded Multiprocessing

Parallel processing per se is nothing new to embedded systems. Many embedded applications are using parallel processing with great success. For example, signal-processing in telecommunications makes use of arrays of digital signal processors (DSPs) to handle wireless and wired signals. Routers use multiple input and output ports with multiple traffic engines to scale to extreme traffic volumes. Automotive applications use large numbers of small processors spread throughout a vehicle. Control applications are distributed across multiple processor cards. Mobile phones contain a multitude of processors in parallel to handle signal processing, applications, Bluetooth ports, etc. The motivation for multiprocessor designs have mainly been to achieve required absolute performance, as well as cost and power efficiency of the overall solution.

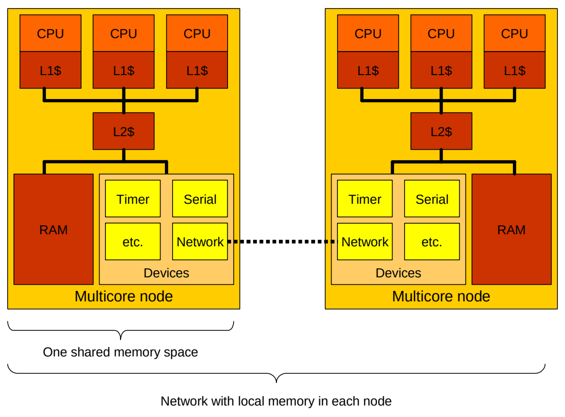
However, most such applications have used asymmetric multiprocessing (AMP), where each processor has its own local memory and performs work that has been statically assigned to it. Each processor has its own private set of tasks running (or tasks that can run), and running tasks do not migrate between processors. The computations to be performed naturally divide into independent tasks and are "embarrassingly parallel" [12][13]. For example, in a mobile phone the user interface can run in parallel to the radio interface, and a mobile phone base station has one thread (or more) per active mobile terminal. Such parallel computation is easier to understand and design than general shared-memory processing [3].

However, debugging an asymmetric parallel system is still harder than debugging a single-processor system. Fundamentally, humans are poor at thinking about parallel systems; we seem to be wired to handle a single flow of events better than multiple simultaneous flows.

The problem now is that the embedded systems software industry has to figure out how to employ shared-memory symmetric multiprocessing to handle applications that have been traditionally implemented on single processors as single tasks. Processor designers have assumed that delivering multicore as a way to avoid the power implications of simply increasing the clock rate is acceptable and that software engineers will have no difficulty making use of processor power delivered in this way. While true for servers, it is not true for most applications of interest, and this creates the current paradigm shift in the industry.

Seymour Cray says, "If you were plowing a field, which would you rather use: two strong oxen or 1,024 chickens?" It seems that we are forced to start harnessing these chickens.

In a shared-memory system, a multiprocessor runs a single instance of an operating system, and tasks can execute on any processor (as decided by the operating system scheduler). Shared-memory systems have many advantages, particularly in balancing loads and migrating tasks between processors. It is more power-efficient to use several processors running at a low clock frequency than a single processor at a high frequency. For example, the ARM11 MPCore multiprocessor varies both the number of active cores and their clock frequency to process a workload with maximum efficiency [5].



As illustrated above, we can expect future embedded systems to contain several shared-memory multiprocessor nodes (on individual boards or blades or cards), connected using a network. Each individual processor core in the system has a local level 1 cache, and shares level 2 cache (if present) and the main memory and device set with the other cores on the same node. Within each SMP node, the caches are kept coherent so that the node presents a single unified memory from the perspective of the software, despite the memory being spread out in several processor-local caches.

4 The Software Breaks

Ignoring the issues of creating efficient parallel programs for the moment, even getting programs to function correctly in an SMP environment is harder than for a single processor. Existing software that has been developed on a single processor might not work correctly when transitioned onto a multiprocessor. That an application works correctly in a multitasking environment does not imply that it works correctly in a multiprocessing environment; serious new issues occur with true concurrency.

True parallel execution (or concurrency) makes it hard to establish the precise order of events in different concurrent tasks. The propagation time of information between processors in shared-memory multiprocessors is not zero (even if it is fairly short, maybe a few hundred clock cycles at most), and this is sufficient to create subtle random variations in the execution, which can snowball. A multiprocessor by nature exhibits chaotic behavior where a small change in initial parameters gives rise to large differences in system state over time. The system is actually inherently unpredictable, at least in terms of timing, and correct function can only be achieved by allowing for this and designing code, which works even in what seems like bizarre circumstances.

The following catalogue of problems attempts to highlight the many new and interesting ways in which software can break on a multiprocessor.

4.1 Latent Concurrency Problems

There can be latent problems in an existing, proven, multitasking workload that runs just fine on a single processor. The presence of true concurrency makes mistakes in protecting against concurrent accesses much more likely to trigger and cause program errors. As the timing of tasks becomes more variable, and they run in parallel for longer periods of time, the task set is simply subjected to more stress. This effect is similar to optimizing a program in a C compiler: optimizations might expose bugs in the program that were previously hidden. The error was always there; it just didn't manifest itself.

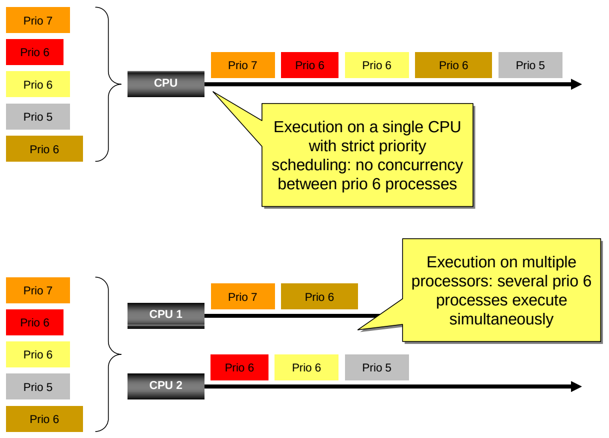
4.2 Missing Reentrancy

To make efficient use of a multiprocessor, all code that is shared between multiple tasks has to support reentrant execution. This means using locks to protect shared data and to allocate local data for each time a function is called. Shared state between invocations of the same function has to be avoided. In a multiprocessor environment, actual occurrences of multiple tasks using the same shared function simultaneously will occur much more frequently (and thus trigger bugs according to Section 4.1).

This effect is especially important for operating systems and shared libraries, as such code will be used heavily by multiple tasks.

4.3 Priorities do not Provide Mutual Exclusion

In application code written for a traditional single-processor, strictly priority-scheduled RTOS, a common design pattern to protect shared data is to make all tasks that access the same data run at the same priority level. With a strict priority-driven scheduler, each process will run to completion before the next process can run, giving exclusive access without any locking overhead. This will fail when true concurrency is introduced in a multiprocessor; the picture below illustrates a typical case:


Prio 7
Prio 6
Prio 6
Prio 5
Prio 6
CPU
Prio 7
Prio 6
Prio 6
Prio 6
Prio 5
Execution on a single CPU with strict priority scheduling: no concurrency between prio 6 processes
Prio 7
Prio 6
Prio 6
Prio 5
Prio 6
CPU 1
Prio 7
Prio 6
Execution on multiple processors: several prio 6 processes execute simultaneously
CPU 2
Prio 6
Prio 6
Prio 5

This code is multitasking-safe on a single processor, but will break on a multiprocessor. This means that even existing proven code will have to be reviewed and tested before it can be assumed to run correctly on a multiprocessor. Explicit locking has to be introduced in order to handle the access to shared data.

One solution proposed for maintaining the semantics of single-processor priority scheduling on an SMP is to only run the task(s) with highest priority. Thus, if there is only a single highest-priority task in a system, only one processor will be used and the rest left idle. This ensures that high-priority tasks do not need to worry about simultaneous execution of lower-priority tasks, but does not solve the problem for tasks with the same priority when priority is used (incorrectly) as a serialization device.

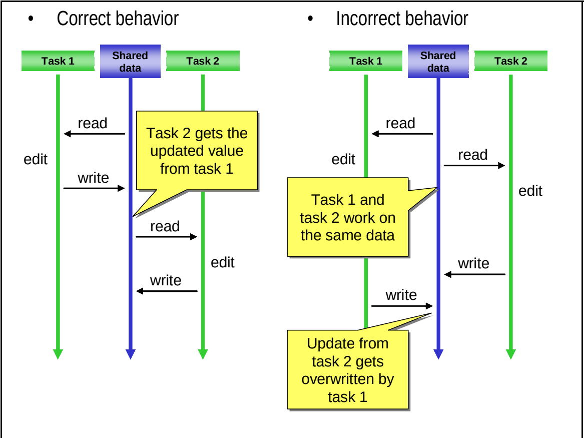
4.4 Interrupts are not Locks

In the operating system and device driver code, you can no longer simply assume that you get exclusive access to shared data by turning off all interrupts. Instead, SMP-safe locking mechanisms have to be used. Redesigning the locking mechanisms in an OS kernel or driver (or user application, in case it makes use of interrupt management) is a major undertaking in the change to multiprocessors, and getting an operating system to run efficiently on an SMP will take time [9][10].

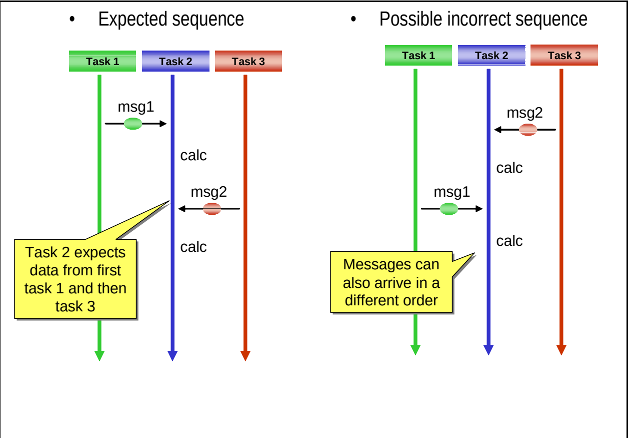
4.5 Race Conditions

Race conditions are situations where the outcome of a computation differs depending on which participating task gets to a certain point first. They typically trigger when some piece of code takes longer than expected to execute or timing is disturbed in some other way. Since race conditions are inherently timing-related, they are among the hardest bugs to find. The name comes from the fact that the tasks are racing forward in parallel, and the result depends on who gets to a certain point first.

The picture on the left below illustrates the classic data race, where a piece of data shared between two tasks and the tasks do not protect the common data with a lock. In this case, both tasks can be editing the data at the same time and will not correctly account for the updates from the other task. If task 1 was fast enough, it could finish its editing before task 2 begins, but there are no guarantees for this. Note that shared data is often a complex data structure, and the net result of a race is that different parts of the structure have been updated by different tasks, leading to an inconsistent data state.



Classic data race


Expected sequence
Task 1
Task 2
Task 3
msg1
calc
msg2
calc
Task 2 expects data from first task 1 and then task 3
Possible incorrect sequence
Task 1
Task 2
Task 3
msg2
calc
msg1
calc
Messages can also arrive in a different order

Message races

The picture on the right illustrates another type of race, the message race, where one task is expecting a series of messages from other tasks. Such messages can come in any order, as there is no synchronization between the tasks; if task 2 implicitly expects a certain order, it will get the wrong results if task 3 happens to send before task 1. Whenever the ordering of events is important, explicit synchronization has to be in place.

Note that races occur in multitasking single-processor systems too, but there they are less likely to trigger, as they require a task switch to occur at an unlucky time. In a true concurrent system, races will happen more often, as discussed in Section 4.1.

4.6 Deadlocks

When all shared data is protected by locks, you get into another situation where the locking itself can be the cause of errors. If two tasks are taking multiple locks in different order, they can get stuck, both waiting for the other task to release the other lock. A simple but fairly typical example is given below:

Task T1	Task T2

`main():`
`  lock(L1)`
`    /* work on V1 */`
`  lock(L2)`
`    /* work on V2 */`
`  unlock(L2)`
`    /* more work on V1 */`
`  unlock(L1)`
	`main():`
	`  lock(L2)`
	`    /* work on V2 */`
	`  foo()`
	`    /* more work on V2 */`
	`  unlock(L2)`
	`foo():`
	`  lock(L1)`
	`    /* work on V1 */`
	`  unlock(L1)`

Task T1 locks lock L1 first, which protects the variable V1. After awhile, T1 also needs to work on variable V2, protected by lock L2, and thus needs to lock L2 while still holding L1. This code in itself is sound, as all accesses to shared data is correctly protected by locks. In task T2, work is being performed on variable V2, and lock L2 is taken. When calling the function `foo()`, V1 is also accessed, and locking is in place.

Assume a scenario where T1 and T2 start at the same time, and manage to obtain one lock each. When T1 gets to the point `lock(L2)`, it stops and waits as T2 is holding that lock. Slightly later, T2 gets to the call to function `foo()`, and duly tries to lock L1. At this point, we are stuck, as T1 and T2 are mutually waiting for the other task to release a lock, which will never happen. We have a deadlock.

The example above illustrates a common cause of deadlocks: calling into functions like `foo(),` which do locking invisible to the caller. In a complex software system, it happens easily that locks get taken in a bad order if no measures are taken to ensure consistent locking orders.

4.7 Partial Crashes

When a computation involves multiple tasks that compute different parts of an overall solution, it is possible that one (or more) of the tasks crash during the computation. This problem in a single task will then escalate to a parallel problem, as the other tasks wait forever for the crashed task to get back with its part of the result.

4.8 Silent Parallelization Errors

Parallel code written with explicit calls to threading libraries will naturally have to check whether thread creation and other operations succeeded, but when using indirect parallel programming by compiler directives like OpenMP, error handling is often suboptimal. The compiler will create extra code in the application to start tasks and handle synchronization, and this code does not have a way to report problems to the main user code. If a program fails to create the tasks it needs at run-time, it might just crash or hang or fail silently with no message to the user or programmer indicating a problem. Such behavior makes it unsuitable for high-reliability code where error recovery is necessary.

4.9 Bad Timing Assumptions

Missing synchronization is a common theme in parallel programming bugs. One particular variant to be wary of is the assumption that one task is going to finish its next work long before some other task needs the results of this work. In essence, as a task T1 is doing something short and simple, we expect that work to be completed before a task T2 finishes something long and complicated, and we know that they start out at the same time. A simple example is the following:

Task T1	Task T2

`create(T2)`
`write(V)`
	`initialize()`
	`read(V)`

Here, we expect T1 to finish writing V long before T2 is done with its initialization. However, it is possible that T2 in some situation completes its initialization before T1 can complete its write. This sort of bug typically triggers under heavy load or unusual situations in a shipping system.

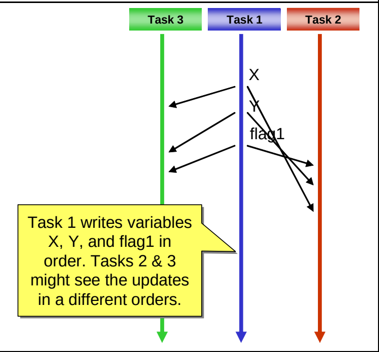
4.10 Relaxed Memory Ordering Wreaks Havoc

For performance reasons, a multiprocessor employs various forms of *relaxed memory orders* (also known as *weak memory consistency models*). Basically, the computer architecture specifically allows memory operations performed to be reordered in order to avoid stalling a processor when the memory system is slow. In most such models, memory operations from one processor may be *observed in a different order* from the viewpoint of another processor. This is necessary in order to get any efficiency out of a multiprocessor system. There are a number of relaxed memory orderings[1], differing in how operations can bypass each other [14]. Understanding relaxed memory orders is probably one of the hardest parts of understanding parallel computing [13], and unfortunately, they are visible to a programmer since they affect data-transfer operations between parallel threads. Many data exchange algorithms that are correct on multitasking single processor systems break when used on multiprocessors.

For example, reads can almost always be performed out of order with respect to the program, and sometimes writes might be seen in different orders on different processors, especially when they originate from different processors. In general, it is necessary to use special operations like *memory barriers* to guarantee that all processors in a system have seen a certain memory operation and that the state is thus globally consistent. Failing to deal with memory consistency will result in intermittent timing-sensitive bugs caused by processors observing the same execution in different orders.

One example is the classic Dekker locking algorithm shown below, which is perfectly valid on a single processor (assuming that each read of a flag variable is atomic), no matter how tasks are interleaved.

[1] Note that the potential operation reorderings allowed by a specific memory consistency model are precisely defined, but that even so, understanding the implications of a particular memory consistency model requires deep considerations. University students usually find memory consistency the hardest part of computer architecture.

Task T1	Task T2	Example sequence
Basic algorithm		 Task 3 Task 1 Task 2 X Y flag1 Task 1 writes variables X, Y, and flag1 in order. Tasks 2 & 3 might see the updates in a different orders.

`flag1 = 1`
`if(flag2 ==  0)`
`  critical section`
	`flag2 = 1`
	`if(flag1 == 0)`
	`  critical section`	

Problem example		

`x = 6`
`y = 5`
`flag1 = 1`
`…`
	`…`
	`while(flag1==0)`
	`loop;`
	`read x`
	`read y`	

Assuming that all variables are zero to begin with, it is quite possible to read old values for x and y in task T2 despite the use of a “lock,” since there is no guarantee that the writes to variables x and y are seen by task T2 before the write to flag1. With a relaxed memory ordering, you cannot assume that writes complete in the same order as they are specified in the program text. For example, if the write to flag1 hit the local cache, and x and y missed the cache, the net result could be that the write to flag1 takes a longer time to propagate. The execution is illustrated on the right in the picture above (in which we assume that each task runs on its own processor). Note that an additional task 3 can see the writes in the “expected” order – which is the common case. The rare case is the problem case. The programming solution is to put in barriers, forcing memory writes and reads to complete before the tasks continue beyond the lock. The cost is in performance, as processors stall when memory operations run to completion.

Note that using the C keyword “volatile” has no effect on memory consistency; it does guarantee that the compiled code writes variable values back to memory, but provides no guarantees as to when other processors will see the change. This is a symptom of a general problem in that most language definitions do not consider the implications of multiprocessors.

A more complex example that fails for the same reason is the following scheduling code from the SPLASH program:

Task T1	**Task T2**	**…**	**Task Tn**

`while(there are workunits)`
`{`
`  allocate new workunit`
`  fill in workunit data`
`  insert workunit into linked list`
`}`
`Head = first workunit`
	`while(MyWorkunit == NULL)`
	`{`
	`  lock critical section`
	`  if(Head!=NULL)`
	`  {`
	`     MyWorkunit = Head`
	`     Head = Head -> next`
	`  }`
	`  release lock for critical section`
	`}`
	`read data from MyWorkunit`		

Here, the worker tasks T2…Tn wait for the linked lists of work units to become available from the master thread T1, and then grab work units off of it protected by a lock. However, considering the possibility of write reordering, the work unit data read by a worker task need not be complete just because the linked list has been updated. And if the locking is not multiprocessor-aware, the whole task queue can be destroyed by processors having a different idea of the value of Head.

5 Debugging Parallel Programs

Debugging parallel programs is generally acknowledged as a difficult problem. Most of the literature on parallel programming focuses on the constructive part of creating a parallel algorithm for a particular problem and how to implement it, and mostly ignores issues of debugging and troubleshooting [6]. Despite the fact that multiprocessing has been on the horizon for a long time, very little in terms of practical tool support for parallel programming and debugging has been produced [2][7][8][9][12][2].

Debugging a software problem on a multiprocessor problem involves three main steps:

- Provoking problems
- Reproducing problems that have been provoked
- Diagnosing and resolving the reproduced problems

What makes multitasking programming and debugging difficult is that provoking and reproducing problems is much harder than with single-threaded programs. In classic single-tasking programs, most bugs are deterministic and caused by particular variations of input. Such deterministic bugs will still occur in each task in a parallel system, and will be solved in a traditional manner.

The parallel program adds a new category of bugs caused by the interaction of multiple tasks, often depending on the precise timing of their execution and communications. Such bugs are the most difficult bugs to provoke and reproduce, not to mention understand and resolve. They have a tendency to be Heisenbugs, meaning bugs that disappear when you try to observe the program in order to discover them [7][8].

This section discusses techniques for debugging multiprocessor programs in spite of this. Some techniques focus on provoking and reproducing bugs and others on diagnosis of the problems.

5.1 Breakpoints on Single Tasks

A traditional single-task debugger can be applied to a parallel software system by debugging a single task in the traditional manner, setting breakpoints and watchpoints, and investigating state. This sometimes does work well, but carries some obvious risks in a parallel setting:

- Breakpoints disturb the timing. Activating a breakpoint or watchpoint and just observing that it was triggered (i.e. not stopping execution) will make the timing of the task different. This can very well hide a timing-related bug.
- A stopped task can be swamped with traffic. If a single task is stopped and the rest of the parallel program continues to run, the other tasks in the system might keep queuing up requests for work and communications with the stopped task. This might cause an avalanche of problems in the rest of the parallel program if queues fill up and other tasks get time-outs on replies.
- A stopped processor in a real-time system will quickly cause a system crash, as the computer system is expected to keep up with the sensors and actuation needs of the controlled system.

Many tools in the market today support running multiple debug sessions in parallel, originally in order to support asymmetric multiprocessors (one window to the DSP, one window to the control processor). Such solutions typically feature various forms of synchronizations between the debuggers, so that when one processor stops, the other stops. Such global stops have a certain skid time, since it takes a small but non-zero time for the debugger to catch one breakpoint and order for the other processors to stop. Also, doing a global stop might not be feasible in a system connected to the outside world. For example, the operating system will still need to run in order to handle interrupts. Stopping a processor does not mean stopping the world.

Such tools are better than a single sequential debugger in an SMP environment, but the hardware support is not really there to make them work as well as expected. Consider the issue of using hardware breakpoints: A hardware breakpoint set in the debug facilities of one particular processor will not trigger if the task is scheduled onto another processor before executing. Debuggers for SMPs must work around such issues, as the hardware itself does not provide the facilities to migrate breakpoints around with tasks.

5.2 Traces

Tracing is an indirect debug that gathers knowledge: If an error is provoked or reproduced when tracing is enabled, the trace will hopefully contain hints to the sequence of events leading up to the error. Using this information, you can then construct a hypothesis as to what went wrong.

[2] Part of the problem could be that parallel programming has been on the horizon for a very long time, always in line as the next big thing but never actually arriving. It's simply a case of "crying wolf."

Traces can be gathered in a number of ways:

- Printf: The most common trace tool (and probably debug tool overall) is to use printouts in a program to determine its execution path. The advantage of using printouts inside the application code is that the information is typically application-level and semantically rich; a debug printout generally states that a program has reached a certain point and shows the values of relevant data at that point. The obvious disadvantage is that adding printouts to a program disturbs its timing, potentially changing its behavior so that the bug disappears. There have been cases where the printouts had to be left inside shipping systems (albeit silenced), as the system did not work without them.
- Monitor code: A more structured way to trace is to use special monitor programs that monitor system execution, outside of the actual application code. Such monitors will have less timing impact on the overall system, but also provide less information compared with a printout approach. Some multiprocessing libraries provide hooks for monitors. Embedded operating systems typically offer tools that trace operating system-level events like task scheduling, pre-emption and messages, which can be very useful to understand the behavior of a system.
- Instrumented code: Some debug and software analysis tools instrument the source code or binary code of a program to trace and profile its behavior. Such instrumentation will change the program behavior, but a tool can use intelligent algorithms to minimize this effect. Instrumenting binaries makes it possible to specifically investigate shared memory communications, as each load and store instruction can be traced.
- Bus trace: Since basically all multiprocessor systems use caches, tracing system activity on the memory bus will only provide a partial trace. Also, some systems employ split buses or rings, where there is no single point of observation. Continued hardware integration is making bus analysis less and less feasible as time goes by.
- Hardware trace using core support: A hardware trace buffer drawing on processor core abilities like embedded trace features and JTAG interfaces is a standard embedded debug tool. Applying a hardware trace to one processor in a multiprocessor system will provide a good trace of its low-level behavior, and provide minimal timing disturbance (provided the trace can keep up with the processor). The information will have to be digested by a debugger to provide a good insight into the system behavior.
- Simulation: Simulation (as discussed in Section 5.6) offers a way to trace the behavior of a system at a low level without disturbing it. The overall functionality is very similar to hardware tracing, with the added ability to correlate traces from multiple processors running in parallel. Simulation avoids the correlation problem.

A general problem with tracing is that when tracing execution on a multiprocessor system, it can be hard to reconstruct a global precise ordering of events (in fact, such an order might not even theoretically exist). Thus, even if the event log claims that event A happened before event B based on local time stamps, the opposite might be true from a global perspective. In a distributed system where nodes communicate over a network, a consistent global state and event history is known to be very hard to accurately reconstruct.

A limitation of most forms of tracing is that they only capture certain events. No information is provided as to what happens between the events in the trace, or to parts of the system that are not being traced. If more fine-grained or different information is needed, the tracing has to be updated correspondingly and the program re-executed.

Detailed traces of the inputs and outputs of a particular processor, program, task or system can be used to replay execution, as discussed in Section 5.7 [8].

5.3 Bigger Locks

If the problem in a parallel program appears to be the corruption of shared data, a common debug technique is to make the locked sections bigger, reducing available parallelism, until the program works. Alternatively, the program is first implemented using very coarse locking to ensure correct function initially. This is typically what has happened in previous moves to multiprocessing in the computer industry, for example in the multiprocessor ports and optimization of Linux, SunOS [9] and MacOS X [10].

After locks have been made coarser, the scope of locking is carefully reduced in scope to enhance performance, testing after each change to make sure that execution is still correct. Finally, since fine-grained locking is usually used to increase performance, performance has to be analyzed. If the wrong locks are made finer-grained, no performance increase might result from an extensive effort. Too fine-grained locking might also create inefficiencies when execution time is spent managing locks instead of performing useful computation.

5.4 Apply a Heavy Load

Reliably provoking and reproducing errors is one of the major problems in debugging parallel systems. Since errors typically depend on timing, stretching the timing is a good way to provoke errors, and this can be done by increasing the load on a system. At close to 100 percent CPU load, a system is much more likely to exhibit errors than at light loads. The best way to achieve this is to create a test system that can repeatedly apply a very heavy load to a system by starting a large number of tasks. The tasks should preferably be a mix of different types of programs, in order to exercise all parts of the software system. This tactic has proven very successful in ensuring high reliability in systems like IBM mainframes [11].

5.5 Use a Different Machine

For most embedded systems, the machine a piece of code will run on is usually known. The hardware design and part selection is part of the system development. Thus, running a piece of code on a different machine might seem pointless: We want it to work on the target system. However, testing on different hardware is a good way to find timing bugs, as the execution timing and task scheduling is likely to be different. It also serves as some form of future-proofing of the code, as code written today will be used in future upgraded systems using faster processors and/or more processor cores.

A machine can be different in several relevant ways: Each processor might be faster or slower than the actual target. It might also have a different number of processors, more or fewer. Moving to different processor architecture or a different operating system is likely to take as much time under a tight schedule, but if it can be done, it is an excellent way of flushing out bad assumptions in the code. Portable code is in general more correct code.

Simulation, as discussed in Section 5.6, offers an interesting variant on using a different machine.

5.6 Simulate the System

The key problem with debugging a parallel system is lack of synchronization between parallel processors and determinism in execution. The inherent chaotic behavior makes cyclical debugging very hard. There is one technique that overcomes these problems: simulation of the target system. Traditionally, simulation has been used as a means to develop and run software before the real hardware was available. In the area of parallel computer system execution, simulation remains useful even after hardware becomes available – simulation of a parallel machine provides explicit control over the execution of instructions and propagation of information, which makes for a very powerful debugging tool.

A simulated system fundamentally provides determinism, as each simulation run repeats the same execution (unless nondeterministic inputs or timing variations are provided as external input). This enables cyclical debugging again. It also makes reproducing errors easier, as once a bug has been provoked in a simulation, the same bug scenario can be replayed over and over again in exactly the same manner.

Simulators will commonly be used as a backend to standard debuggers, in which a user simply connects to the simulator instead of to a JTAG probe, remote debug server or other debug port. The simulator makes it possible to single-step one task while time is virtually standing still for other parts of the system, which solves the problems with breakpoints discussed in Section 5.1. For a real-time system where a simulation of the external world is available, simulation also makes it possible to single-step code and to stop the system without the external world running ahead.

Full-system simulators capable of running the complete software stack (from firmware to application software) also provide the ability to inspect and debug the interaction of an application with the operating system, as well as low-level code such as operating-system kernels and device drivers. Fundamentally, a full-system simulator provides a controlling layer underneath the operating system, which provides capabilities which cannot be implemented in traditional debug tools.

A simulator will not be able to faithfully reproduce the detailed timing of a real system in all but the simplest cases. This is not really a problem for software development, as the goal is to find and eliminate software bugs: If they occur in the simulator, they could have happened in some real circumstance as well, and thus they are valid bugs that should be fixed.

Furthermore, a simulator offers an interesting variant of running on a different system, as discussed in Section 5.5. The simulator can be used to increase the number of processors present beyond that available on any real-world machine in order to stress the software, and checks its behavior on future machines with more cores. Simulation can make different processors run at different speeds in order to provoke errors. Performing such corner-case chasing in simulators is common in hardware design, and it has great potential as a tool for software developers as well.

Several vendors offer simulation solutions for embedded systems of varying scope. The investment to create a simulation model can be quite high. But for larger development projects, the benefits typically outweigh the costs, especially when considering the added value of reverse debugging, as discussed in Section 5.7.

5.7 Replay Execution

A recurring theme is the problem that re-executing a multitasking, multiprocessor program will result in a different execution, potentially failing to trigger bugs. If a program can be forced to execute in the same way multiple times, debugging will be much easier. Such techniques are known as record-replay techniques. Replay is different from tracing, as the amount of data needed to facilitate a deterministic replay can be made much smaller than a full trace. The trace recorded only needs to contain the non-deterministic events from the real system, like task switches, message arrivals and other communication between tasks [7][8].

To enable replay, the system has to contain a monitor that can record relevant data as well as force the execution to replay the same behavior. Typically, this has to work on the operating-system level, but it can also be applied at the application level if a program has sufficiently well-defined interfaces to the outside world. (A state machine is a good example of this: The time and type of messages arriving will determine its precise behavior).

In a parallel program setting, replay can be done on a single task, on a set of tasks (an entire application), or even on all software running on a single computer node. Recording-based replay allows the debugger to isolate one part of the system, as all input values and asynchronous timings are known. The rest of the system does not need to be running in the debug session, simplifying the problem.

5.8 Reverse Debugging

One of the key problems in debugging parallel programs in general, and parallel programs running on multiprocessors in particular, is that re-executing a program to put a breakpoint to locate an error is likely not to reproduce an error. Instead, a programmer would like to be able to go back from the point at which an error occurs and investigate the events that immediately preceded the error. What is needed here is a tool that allows reverse debugging, i.e. the ability to back up in a program from the current state instead of (attempting to) reproducing the state by executing from the start [12].

Reverse debugging is particularly useful for bugs corrupting the state of the machine. If the stack is overwritten or a process has terminated, there is little material to use for post-mortem bug analysis. By backing up in time, the state prior to the bug can be examined.

Tools for reverse debugging do exist today in the market. They can be based on simulation (see Section 5.6), trace collection (see Section 5.2) or virtual machines [15]. Such tools offer the ability to back up the execution to examine the path that led to a tricky bug. Tools based on traces will by necessity have a limited window of observation (limited by the size of the trace buffer), while simulation- and virtual-machine-based tools can make use of the resources of a powerful workstation to allow much longer sequences of execution to be observed. Also, most tools do not offer multiprocessor support.

Compared to replay as discussed in Section 5.7, reverse debugging is typically faster in practical use, as backing up a short distance in the execution is faster than reloading and replaying a complete execution. One of the main advantages of reverse debugging is actually that a programmer can focus on a small context and go back and forth over it, applying different breakpoints and traces to understand the bug, without long turnaround times that break the flow.

5.9 Formal Methods

No overview of parallel debugging techniques is complete without mention of formal methods. Researchers have proposed many different algorithms and techniques for formally proving the absence of errors or automatically locating errors in parallel programs. Some of these methods have made it onto the market. One instance is Intel's ThreadChecker, a tool that attempts to check that all shared data accesses follow a correct locking regime. If the locking regime assumed by the tool is used in your program, it can offer very good help for locating that particular (and important) class of bugs.

In general, real-world formal method tools applicable to real programs will be analogous to the classic lint program: They will find potential problems and initially produce many false warnings. Not all problems flagged will be real problems, as the tools' understanding of the code will always be incomplete. They will also only find the types of bugs they were designed to find. For example, a wild pointer will not be found by a tool checking for deadlock in message passing.

6 Summary

This paper has discussed the software implications of the undergoing hardware paradigm shift to multiprocessor, shared-memory computers. This is a fundamental change in how computers are constructed, and will affect both existing code and the creation of new code. We have taken inventory of problems that occur when moving to multiprocessor environments, and discussed a number of debug techniques available today on the market. The future is parallel, and we really have no choice but to prepare our code and ourselves for handling that parallelism. Even if the current situation in programming tools for multiprocessors is fairly bleak, we should expect new tools to hit the market in coming years that will make multiprocessor programming and debugging easier.

Acknowledgements

Thanks to Professor Erik Hagersten, Lars Albertsson, Dan Wallin, Henrik Löf, and my colleagues at Virtutech for discussions and input on multiprocessing and software development.

More Reading

If you want to know more about this subject, here are some recommended articles and books that will provide insight into the issues of parallel programming:

[1] Luiz Barroso. *The Price of Performance*, ACM Queue, September 2005.

[2] Jeff Bier. *Back to the drawing board on multicores, multiprocessing*, Embedded.com (www.embedded.com), Nov 28, 2005.

[3] Michael Christofferson. *Using an asymmetric multiprocessor model to build hybrid multicore designs*, Embedded.com, Nov 5, 2005.

[4] Scott Gardner. *Multicore Everywhere: The x86 and Beyond*. Extremetech.com, Jan 9, 2006.

[5] Kevin Krewell. *ARM Opens Up to SMP – MPCore Supports One to Four CPUs*, Microprocessor Report, May 24, 2004.

[6] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing (2nd edition)*, Addison Wesley, 2003.

[7] Joel Huselius. *Debugging Parallel Systems: A State of the Art Report.* MRTC Report no. 63, (www.mrtc.mdh.se), September 2002.

[8] Charles E. McDowell and David P. Helmbold. *Debugging Concurrent Programs*, ACM Computing Surveys, December 1989.

[9] Mache Creeger. *Multicore CPUs for the Masses*, ACM Queue, September 2005.

[10] John Siracusa, *MacOS X 10.4 Tiger*, ArsTechnica (www.arstechnica.com), April 28, 2005.

[11] S. Loveland, G. Miller, R. Previtt, and M. Shannon: *Testing z/OS: The Premier Operating System for IBM’s zSeries Server*. IBM Systems Journal, Vol 41, No 1, 2002.

[12] Herb Sutter and James Larus. *Software and the Concurrency Revolution*, ACM Queue, September 2005.

[13] Herb Sutter. *The Free Lunch is Over*, Dr. Dobbs Journal, March 2005.

[14] John L Hennessy and David A Pattersson. *Computer Architecture: A Quantitative Approach, 3rd Edition*. Morgan Kaufmann, May 2002.

[15] Samuel King, George Dunlap, and Peter Chen. *Debugging operating systems with time-traveling virtual machines*, Proc. USENIX Annual Technical Conference, April 2005.

[16] Kang Su Gatlin and Pete Isensee. *Reap the Benefits of Multithreading without All the Work*, MSDN Magazine, October 2005.