What to Make of Multicore Processors for Reliable Real-Time Systems?

Technical Report TR-100401

Theodore P. Baker*
Dept. of Computer Science
Florida State University
Tallahassee FL 32306-4530, USA
e-mail: baker@cs.fsu.edu

4 April 2010

Abstract

Now that multicore microprocessors have become a commodity, it is natural to think about employing them in all kinds of computing, including high-reliability embedded real-time systems. Appealing aspects of this development include the ability to process more instructions per second and more instructions per watt. However, not all problems are amenable to parallel decomposition, and for those that are, designing a correct scalable solution can be difficult. If there are deadlines or other hard timing constraints the difficulty becomes much greater.

This paper reviews some of what is known about multiprocessor scheduling of task systems with deadlines, including recent advances in the analysis of arbitrary sporadic task systems under fixed-priority and earliest-deadline first scheduling polices. It also examines critically the foundations of these theoretical results, including assumptions about task independence and worst-case execution time estimates, with a view toward their practical applicability.

1 Introduction

Over the past decade, the microprocessor industry has been moving increasingly toward symmetric multicore architectures. The introduction of the AMD dual-core Opteron, in 2004, was followed closely by the Intel dual-core Pentium D and the IBM dual-core Power5 processor. Increases in feature *density* predicted by Moore's Law appear to be sustainable for at least a few more generations, but increases in processing *speed* do not. Pipelining and speculative execution appear to have reached a point of diminishing returns. Clock rates also seem to have reached a limit, as power consumption increases at approximately the cube of clock frequency. Manufacturers have decided that the way to extract improvements in performance from further advances in miniaturization is through the use of multiple processors of moderate power, executing in parallel [38]. Two- and four-core processors are already a commodity, eight-core processors have been delivered, and "terra-scale" computing is predicted [30].

What will we make of these new processors? This question should be on the minds of developers of every kind of software. Appealing prospects include the ability to process more instructions per second and more instructions per watt. However, not all problems are amenable to parallel decomposition, and for those that are, designing a correct scalable solution can be difficult. If there are deadlines or other hard timing constraints the difficulty becomes much greater.

This paper seeks to convey an appreciation of the gap that exists between what is known in theory about the problem of scheduling ideal multiprocessors to meet deadlines, and the behaviors of actual multicore processors. It begins with a sampling of theoretical research that appears as if it should be applicable to

*This report is based upon work supported in part by the National Science Foundation under Grant No. EHS-0509131.

this problem. It then questions the foundations of these theoretical results, including assumptions about processor architecture, task independence and worst-case execution time estimates, with respect to their applicability to current and future generations of multicore processors in high-reliability real-time systems. It finally expands the focus to consider the directions that programming languages and software design methodologies may need to go in order to better exploit multicore technology.

2 Scheduling Theory Foundations

Scheduling theory is based on abstract models of workloads and processors. A great many different models have been studied. This paper focuses on just one type of workload model and one type of processor model.

Task workload models. In real-time scheduling theory, a *task* is an abstraction for a source of a potentially infinite sequence of *jobs*. A job is a schedulable computation with a finite *execution time*. Other parameters of each job include a *release time* and a *deadline*. Each task has a set of *scheduling parameters* that constrain the execution times, arrival times, and deadlines of the sequences of jobs that it may generate. The jobs of each task are required to be executed serially, using only one processor at a time.

A *task* in scheduling theory may be used to model an execution of a sequential thread of control, such as an Ada task[1], in software. Jobs correspond to bursts of computation between wait states. The execution time of a job is the amount of processor time used during one such burst of computation, the release time is the arrival time of the event that triggers the transition from waiting to competing for execution, and the completion time is the time of the next transition to a wait state.

This paper focuses primarily on periodic and sporadic task systems. A *sporadic task system* is a set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of tasks, each characterized by a triple (p_i, e_i, d_i) where: p_i is the minimum separation between release times of the jobs of the task, also known as the *period* of the task; e_i is an upper bound on the execution time for each job of the task, also known as the *worst-case execution time* (WCET); and d_i is the *relative deadline*, which is the length of the scheduling window of each job. A periodic task system is said to have *implicit deadlines* if $d_i = p_i$ for every task, *constrained deadlines* if $d_i \leq p_i$, and *arbitrary deadlines* if there is no such constraint. A *periodic task system* is like a sporadic task system except that the separation between release times of τ_i must be equal to p_i. Additional parameters are sometimes specified, such as the release times of the first job of each task, and an upper bound on release time jitter.

Real systems often do not fit the periodic or sporadic model, or have any other sufficient constraints on the patterns of arrivals and execution times of jobs to support schedulability analysis. One way of accommodating such workloads is to queue their jobs to be executed by a *server task*. The server task is scheduled according to an algorithm that limits the amount of processor time that the server can consume within any given time interval, by imposing a *budget*. In effect, the natural job boundaries of the aperiodic workload are redrawn by the server scheduling mechanism, which ends the current "job" of the server whenever it reaches the limit of its budget, and releases a new "job" whenever the server's budget is extended. Sporadic Server [44] and Constant Bandwidth Server [1] are examples of such *bandwidth-limiting algorithms*, which allow the worst-case behavior of a server to be modeled by a sporadic task in most contexts.

Identical multiprocessor models. This paper focuses on multiprocessor platforms with a set of identical processors and shared memory with uniform access speed. Every job is presumed to be executable on any processor, with no difference in worst-case execution time between processors.

The above models leave out many ugly details of real systems, including release time jitter and overheads such as context-switches and migrations of tasks between processors, in order to make analysis more tractable. When applying the theory, one must add sufficient margins to the task parameter values to account for such differences between reality and the model[2].

[1] This overloading of the word "task" can lead to erroneous reasoning, since not all Ada tasks can be modeled as scheduling theory tasks. In this paper "task" always means a scheduling theory task, except in the phrase "Ada task". "Thread" is used for Ada tasks and other similar threads of control.

[2] Narrowing these margins would appear to require accounting explicitly for circular dependences between actual job execution times and decisions made by on-line schedulers. A critical concern in the application of any such analysis would be potential instability under execution time variations caused by factors other than the scheduler.

3 Scheduling Algorithms and Tests

A *schedule* is an assignment of jobs to processors, varying over time. A schedule is *feasible* if it satisfies all the constraints of the workload, including job deadlines. A collection of jobs is said to be feasible if there exists a feasible schedule for it. For practical purposes schedules must be computable by a *scheduling algorithm*. A particular workload is said be *schedulable* by a given algorithm if the algorithm always finds a feasible schedule.

Scheduling algorithms can be dichotomized as static *vs.* dynamic, off-line *vs.* on-line, and preemptive *vs.* nonpreemptive. Priority-based on-line schedulers are classified according to how priorities are allowed to vary. Rate Monotonic (RM) scheduling, in which tasks with shorter periods are assigned higher priorities, is an example of a *fixed-task-priority* (FTP) scheduling algorithm. Earliest-deadline-first scheduling is an example of a *fixed-job-priority* (FJP) algorithm. PD^2[2] is an example of a *dynamic priority* algorithm. *Within this paper, all on-line scheduling algorithms are assumed to be applied in preemptive mode.*

A *schedulability test* tells whether a particular workload is schedulable by a given algorithm. A schedulability test is *sufficient* for a given class of workload and processor models if passing the test guarantees the workload is schedulable. It is *exact* if it also only fails when there is some sequence of jobs consistent with the workload model that is not schedulable.

One of the subtleties of multiprocessors is that certain intuitively appealing scheduling algorithms are subject to "*anomalies*", in which a schedulable system becomes unschedulable because of some apparently harmless change, such as earlier-than-expected completion of a job. A good scheduling algorithm should continue to schedule a task system satisfactorily if actual tasks behave better than the specifications under which the system was validated, or if the specifications of tasks are changed in a direction that reduces the overall workload. This property, named "*sustainability*" in [14], has been shown to hold for some multiprocessor scheduling algorithms and tests, but not for others (*e.g.* [9]).

Anyone making a choice of scheduling algorithm for an application should consider several factors, including: (1) sufficient generality to cover the variety of workload types expected in the application; (2) flexibility to handle the changes in task specifications that are expected over the lifetime of the application; (3) compatibility with the operating system that will be used; (4) run-time determinism or repeatability, as an adjunct of system testability and reliability; (5) sustainability of the algorithm and the available schedulability tests for it; (6) effectiveness of the scheduling algorithm in scheduling jobs to complete within deadlines; (7) effectiveness of the available schedulability tests. Studies of real-time scheduling theory mostly focus on the last two issues, but for practical purposes it is important to consider all of them. It is also important to consider the last two as a pair, since *a scheduling algorithm can only be trusted to schedule a particular task system correctly if it has been verified to do so.*

There are several empirical ways to evaluate multiprocessor scheduling algorithms and tests. One is to compute the success ratio on large numbers of randomly generated task systems (*e.g.* [8]). This can provide useful information if the characteristics of the task sets considered are similar to those expected for a given class of applications. It may be the only way to compare some combinations of scheduling algorithms and schedulability tests. Such experiments can be especially informative if the task parameter values used in the tests are adjusted to take into account the measured overheads of an actual implementation (*e.g.* [22])[3].

Analytic ways to evaluate scheduling algorithms and tests include "speedup factors" (*e.g.* [12]), which are out of scope for the present paper, and *utilization bounds*. The *utilization* u_i of a task τ_i is the ratio e_i/p_i of its execution time to its period. A *utilization bound* for a scheduling policy is a function β such that each task system τ is guaranteed to be schedulable if $u_{\text{sum}}(\tau) <= \beta(\tau)$, where u_{sum} is the sum of the utilizations of all the tasks in the system. The bound is *tight* if there are unschedulable task systems with $u_{\text{sum}}(\tau)$ only infinitesimally larger than $\beta(\tau)$.

A *density bound* is an extension of the notion of utilization bound, for task systems that have other than implicit deadlines ($d_i \neq p_i$). The definition of density bound is the same as utilization bound, except that the utilization u_i is replaced by the *density*, defined as $\delta_i \stackrel{\text{def}}{=} e_i / \min(d_i, p_i)$.

This paper uses utilization bounds and density bounds to provide insight into the relative effectiveness of various methods of multiprocessor scheduling, not because they tell the whole story, but because they can

[3]Running schedulability tests on random task sets is not the same as simulating actual task system executions. The latter is not a valid way to verify schedulability for multiprocessor systems since the "critical instant" property (*c.f.* [34]) no longer holds.

be stated concisely, can be compared easily, and there are examples of sufficient schedulability tests.

Care must be taken when interpreting such bounds, since they can be misleading. Both pertain only to schedulability of worst-case (pathological) examples[4]. Empirical studies have shown that much higher utilization levels can be achieved for most task systems.

4 Static Scheduling

Historically, the analysis of scheduling for multiprocessors focused first on static scheduling techniques, in which a finite schedule is computed off-line. Static scheduling has been studied extensively in the literature of operations research and discrete mathematics. Although optimal static scheduling is known to be NP-hard for all but a few simple classes of problems [46], it is still practicable for real-time task systems of moderate size. Moreover, for practical purposes optimal scheduling is not required; any schedule that satisfies the application constraints will do.

There is a wealth of theory and published practical experience on static scheduling of multiprocessors. One representative example is [48], which reports success in optimal multiprocessor scheduling for periodic systems with deadlines, precedence, and exclusion constraints. A more recent one is [31], which shows how to approximate optimal scheduling using linear programming.

Static schedules have several virtues. They can handle complex forms of constraints and achieve a high degree of optimization. Optimizations can be applied to a variety of criteria, including output jitter, power, and memory. They can eliminate the need for software locking, by scheduling access to shared resources during non-overlapping time intervals. They can also minimize buffering and dataflow blocking, such as between producers and consumers. Though static schedules are intrinsically periodic, they can be adapted to handle sporadic and aperiodic workloads by means of a periodic server. Since the schedule repeats, behavior observed during a few periods of testing is a valid predictor of behavior over longer periods of time. Static scheduling may also prove to be a practical necessity for accurate analysis of the effects of task-dependent interference between processors on WCET.

Static scheduling does have shortcomings, including a tendency to over-allocate processor time, and fragility with respect to run-time variations in workload and longer-term changes in system specifications. Static scheduling may be impracticable for systems in which the release times and execution times of jobs are highly variable or cannot be predicted with confidence. In the single-processor domain, the limitations of static scheduling have led to widespread adoption of priority-based on-line scheduling algorithms.

However, where the workload is sufficiently predictable, static scheduling may still be the best way to achieve reliable high performance from a multicore processor. Moreover, with multiple cores one can apply static scheduling to an appropriate subset of the system functionality, on a subset of the processors, and apply dynamic scheduling to the rest.

5 Partitioned Dynamic Scheduling

Until recently, partitioned scheduling was widely held to be the only way to use multiprocessors for embedded real-time systems. It is a very natural and convenient bridge from single-processor dynamic scheduling to a multiprocessor platform. By assigning each task to one processor, statically, and then applying a dynamic local scheduling algorithm and schedulability test, one achieves some of the predictability of a static schedule and some of the flexibility of a dynamic scheduler, without having to develop any new scheduling algorithms or schedulability tests. Finding a partition that is feasible locally on each processor may be accepted as a sufficient system-level schedulability test, if the tasks are sufficiently independent. Sustainability of the schedulability analysis carries over from the single-processor local scheduling algorithm.

Optimal assignment of tasks to processors is a form of bin-packing problem, which is NP-hard, but optimality is not required. The First-Fit-Decreasing-Utilization partitioning algorithm has been cited as

[4]Care must also be taken when applying a utilization or density bound outside the constraints of the model for which it has been proven. Some of the original published proofs only mention periodic task systems. There is a pattern of these analyses remaining valid for sporadic tasks, because good scheduling algorithms are sustainable under later job release times that do not violate the sporadic minimum separation constraint. Another pattern is of utilization bounds extending to density bounds. One should not assume any such extension is valid without checking that it has been proven.

being very effective in some publications. However, in practice one would want to use an ordering heuristic that takes into account addition considerations, including localization of data sharing, balancing data flow rates between cores, and distributing excess processor capacity among the processors.

Since the allocation of processing resources is forced to be done fixed-size chunks, partitioned scheduling is generally not *work conserving*; that is, a processor may be idled while tasks eligible for execution on other processors are not able to execute. This can result in partitioning failures at low utilization levels, and even when partitioning is successful the idling of processors can result in longer average-case response times and lower total system throughput than with global scheduling.

For partitioned preemptive Earliest-Deadline-First (EDF) scheduling, a utilization bound of $\frac{m\lfloor 1/u_{\max}(\tau)\rfloor+1}{\lfloor 1/u_{\max}(\tau)\rfloor+1}$ for m processors was derived in [36]. For the unrestricted case where $u_{\max}(\tau) = 1$ this reduces to $(m+1)/2$. The proof is for periodic task systems only. This worst-case utilization bound is tight in the sense that a scheduling algorithm that does not vary priority within a single job cannot achieve a utilization bound higher than $(m+1)/2$ on m processors, whether scheduling is partitioned or global. That is easy to see by considering a set of $m+1$ tasks, each with processor utilization infinitesimally larger than 50% [4].

For partitioned RM, a utilization bound of $(n-1)(2^{1/2}-1)+(m-n+1)(2^{1/(m-n+1)}-1)$ for m processors and n tasks was derived in [35]. For large n, this reduces to $(m+1)(2^{1/2}-1)$. This result was proven to be tight for "reasonable" partitioning schemes based on the single-processor RM utilization bound as a test of schedulability, and the proof has been claimed to extend to all partitioning schemes. The proof is for periodic task systems only.

Partitioned scheduling can handle periodic and sporadic task sets, and can also handle aperiodic workloads scheduled under a bandwidth-limiting server mechanism. Several authors have studied partitioning algorithms for these more general workload models, but a description of those results does not fit within the scope of the present paper.

An advantage of partitioned scheduling is that efficient single-processor locking mechanisms such as SRP [5] can be applied for resources that are only shared by tasks on the same processor, and higher-overhead global locking mechanisms can be reserved for resources shared across processors. Space limitations here permit only the barest summary of research on protocols for global locks, which is still inconclusive. The most appropriate locking protocol for global resources will depend on (1) whether task scheduling is partitioned or global, (2) the priority model, and (3) the length of critical sections. However, a central consideration in all cases is to minimize idle processor time caused by tasks awaiting global locks. One approach, used by the Distributed Priority Ceiling Protocol (D-PCP) [40], is to bind each globally shared resource to a single processor and execute sections on it via remote procedure calls to a high priority server on that processor. An alternative approach, used by the Multiprocessor Priority Ceiling Protocol (M-PCP) [40], Multiprocessor Stack Resource Protocol (M-SRP)[25], and Flexible Multiprocessor Locking Protocol (FMLP) [20], is to raise tasks holding global locks to nonpreemptible local priority. It was shown in [37] that using the same priorities for local scheduling and service order on global locks could be harmful for overall system schedulability, as compared to service based on tasks' tolerance for blocking, or even FIFO. The M-SRP and FMLP adopt FIFO service. The FMLP additionally optimizes short-wait locks by using bare spinlocks. The performances of some of these algorithms in actual implementations on multiprocessors are reported in [21, 41, 26].

Unfortunately, critical sections are not the only kind of blocking that can occur in a real system. Dataflow blocking can occur, in which one task waits for data or buffer space to be produced by another. This kind of blocking does not require idling the processor in a single-processor system or with global scheduling, because whenever one task has to wait for another to perform an action the scheduler can give the processor of the waiting task to the awaited task. In contrast, if the waiting task and the awaited task are on different processors, the waiting task's processor may become idle. This idle time can reduce system throughput in the short term, which can cause congestion later, resulting in missed deadlines. If care is not taken in assigning deadlines or priorities across processors, a third task may preempt the awaited task on its processor, resulting in potentially unbounded blocking. Appropriate buffering can reduce data flow blocking, but the partitioning algorithm still needs to pay attention to matching average data flow rates across processors.

A partitioning of tasks among processors may be fragile. If the execution time of one task increases, there is no limit to the range of the potential side effects. Repartitioning and lock protocol changes may be necessary. The effects may cascade across every processor. Fragility may be reduced by distributing excess capacity evenly across processors.

While it may seem wasteful to only use 50% of of the theoretical processing capacity of a system, thinking

in terms of the worst-case utilization bound may not be so bad. Pushing for high utilization of cores on multicore processors may be just as bad for performance as pushing disk and memory utilization to high levels on single-processor systems. For reasons explained in Section 8, trying to schedule all processors at 100% capacity is likely to be counter-productive, as job execution times can grow with greater intercore contention for data paths and memory access. Moreover, idle cores need not necessarily translate to wasted power.

Partitioned scheduling is easy to implement. The partitioning is done off-line. The on-line component is low-overhead because of the per-processor task ready-queues, which incur less contention for concurrent access and can be protected by a lighter-weight locking mechanism than is required to maintain a global ready-queue.

Test coverage with partitioned scheduling will be better than with a globally scheduled system, because of the reduction in the number of possible combinations of concurrent task activities on different processors. Execution times should also be more predictable, for the preceding reason and also because of the absence of migration events.

6 Task Splitting

An inherent limitation of all partitioned scheduling schemes is that each task must execute entirely on one processor. This constraint makes optimal partitioning NP-hard, and prevents a pure partitioned scheduling algorithm from achieving a utilization bound greater than $(m + 1)/2$.

Recent research has attacked this limitation using *task splitting* techniques. These algorithms deviate from the strict partitioned model by allowing limited planned migration of a few tasks, typically up to $m - 1$ tasks for m processors. The partitioning algorithm assigns tasks to processors in some heuristic order. When it reaches a point where adding another task to a given processor would cause a local deadline miss it splits that task in two, assigning a fraction to the given processor that will fit without causing any missed deadlines, and migrating the remainder to another processor. This approach is able to get past the $(m + 1)/2$ limit on utilization bounds for partitioned fixed-job-priority scheduling.

Several task splitting techniques have been proposed. One noteworthy recent example is [29], which demonstrates that a partitioned fixed-task-priority (FTP) scheduling algorithm can achieve the utilization bound $mn(2^{1/n} - 1)$ on m processors. This is of special interest because it equals the worst-case RM utilization bound for a single processor. Another recent example is [32], which is based on preemptive EDF local scheduling and obtains a utilization bound of $m(4\sqrt{2} - 5)$ on m processors, or about 65% of the total processing capacity.

So far, it does not seem that any of these techniques has been tested in an implementation, so there remain questions about their practicability.

7 Global Scheduling

As an alternative to partitioned scheduling, a dynamic scheduling algorithm may be applied in global mode, where all tasks compete for execution on all processors. Global dynamic scheduling can be work-conserving, which can result in better scheduling performance than partitioned scheduling. In fact, there is a very simple global dynamic scheduling algorithm that can meet all deadlines at utilizations up to 100%.

The key to understanding how to get 100% processor utilization and still meet deadlines with the workload and processor models described here is to recognize the importance of keeping all processors busy. It is easy to keep a single processor busy, by never idling the processor if there is work to be done; this can be achieved without constraining the order in which jobs are executed, and so there is no conflict with executing jobs in deadline order, and no problem achieving all deadlines up to 100% processor utilization. With more than one processor, the situation changes. The job execution order can affect the ability to keep all processors busy. For example, if a dual-processor scheduler has two short jobs and one long job ready, and decides to execute the short jobs first, it may end up later with one processor idle. The problem is that a single job cannot use additional processors[5]. One way to work around this limitation is to split up the job's execution

[5]Unless, of course, one breaks out of the task model and codes jobs for parallel execution. This is one of the weaknesses of the task model, discussed further in Section 9.

and interleave it serially with other tasks. Suppose the processor time can be split up into small enough units that each task τ_i can be assigned a fraction u_i of a processor between its release time and its deadline. Such a scheduler is optimal in the sense of never missing a deadline for a feasible task system, and can achieve 100% utilization without missing deadlines on implicit-deadline periodic task systems.

While this ideal processor sharing model is impracticable to implement, [15] showed that the same utilization bound can be achieved by a quantum-based time-slicing approximation. A number of variants of this proportional fair-share (Pfair) scheduling concept have been explored. One variant, called PD^2, is work-conserving and has been shown to be optimal for the scheduling of independent asynchronous implicit-deadline periodic tasks [2].

A criticism of the Pfair approach is that by slicing time finely it incurs a large scheduling overhead. Brandenburg *et al.* [22] performed experiments using an implementation of the PD^2 algorithm on a variant of the Linux operating system kernel using an 8-core Sun "Niagara" processor. They measured the scheduling overheads on a few examples, and then applied schedulability tests to large numbers of randomly generated task systems with shared resources and critical sections. Potential dataflow blocking was not taken into account. The task execution times were adjusted to allow for scheduling overheads, and schedulability tests were modified to take into account blocking times due to critical sections. The experiments showed that that PD^2 had serious problems with high preemption and migration costs until the scheduler was modified to stagger the time-slicing points of the processors across the cores, so that quantum expirations no longer occurred synchronously on all processors. Even with this improvement, PD^2 did not seem to perform as well as partitioned EDF, except for systems with tasks near the 50% utilization region (which is known to be pathological for partitioning algorithms), and without a high degree of global resource sharing. However, it is difficult to say whether such experiments will accurately predict performance in any particular application.

Global applications of EDF and RM were neglected for decades after Dhall [24] showed that the utilization bound on m processors is 1 (as if the system had only one processor). The proof is a pathological case involving m low-utilization (*light*) periodic tasks and one high-utilization (*heavy*) task with slightly longer period. Fortunately, closer study of this phenomenon over the past seven years revealed ways of obtaining much better performance.

A global EDF utilization bound of $m-(m-1)u_{\max}(\tau)$ was derived in [45, 27]. This shows that worst-case behavior only occurs for large values of the maximum individual task utilization $u_{\max}$. The bound can be shown to be tight by generalization of the Dhall example. However, it is not an exact schedulablity test, and other forms of tests have done much better in experiments with random task systems [8]. The global EDF utilization bound extends to a density bound for sporadic task systems with arbitrary deadlines. In this more general form it has been called the "density test" for global EDF schedulability.

For global RM, a utilization bound of $\frac{m}{2}-(\frac{m}{2}-1)u_{\max}(\tau)$ was derived [17]. It also extends to a density bound.

Variants of EDF have been designed with higher utilization bounds for systems with large $u_{\max}$. An algorithm called EDF-US[ζ] gives top priority to jobs of tasks with utilizations above threshold ζ and schedules jobs of the remaining tasks according to their deadlines [45]. It achieves a utilization bound of $(m+1)/2$, which is tight.

Similar variants of RM have also been proposed. RM-US[ζ] gives higher priority to tasks with utilizations above ζ [4, 7]. A RM-US utilization bound of $\frac{m+1}{3}$ is proven in [17]. A tight bound is not known, though [39] argued that the optimum value of ζ is approximately 0.3748225282.

Another recently discovered FTP priority assignment algorithm, known as SM-US[ζ] and based on ordering tasks by slack, has been shown to have a utilization bound of $m(2/(3+\sqrt{3}))$ on m processors, or approximately 38% of the total processing capacity [3].

Despite the low worst-case utilization bounds, fixed-job-priority algorithms have several advantages. They perform fewer context switches than the Pfair methods, and hence have lower scheduling overhead. They can achieve more precise control over timing, by using interval timers that are finer grained than the overhead of quantum-based scheduling methods would permit. In the case of fixed-task-priority, support already exists in most operating systems, including Linux.

Although the fixed-job-priority (FJP) algorithms do not have any practicable exact schedulability tests[6], there is a steadily growing collection of practicable *sufficient-only* tests for FJP scheduling policies, including

[6]The only known exact tests are based on exhaustive exploration of the scheduling state space, are limited to integer time values, and have unacceptable growth in time and space [10, 28].

the utilization bounds cited above and a number of more accurate tests that cannot be described within the space limitations here. Examples for EDF include [6, 16, 11, 18] and examples for FTP include [7, 17, 13]. These tests are difficult to compare. They generally cannot be strictly ranked in scheduling effectiveness, in the sense that there are examples of schedulable task systems that are recognized as schedulable by each test that are not recognized as schedulable by others. Some definitely appear to be more effective than others, on the average, for randomly chosen task systems. Some have good speed-up factors but do not perform well on the average. Some have been shown to be sustainable, some have been shown to be unsustainable, and the sustainability of others remains unknown. Other important properties, including the effects of blocking due to critical sections have not been studied well. So, it is likely to take several more years of study for any consensus to emerge on which of these tests are most useful in practice.

Global scheduling is more difficult to implement than partitioned scheduling. The shared dispatching queue is a bottleneck which becomes more serious as the number of cores grows[7]. The interprocessor interrupts needed to trigger scheduling on other processors are also costly. However, experiments with implementations of several global scheduling algorithms, including PD^2 and global EDF, in a variant of the Linux kernel have been reported, and the performance looks good [22]. Adding support for the RM-US and EDF-US hybrids would be extremely simple, and would increase scheduling effectiveness without adding any overhead.

Testing is likely to be more difficult for globally scheduled systems, unless explicit interprocessor synchronization points are inserted to reduce timing variations. That is, introduction of task migrations on top of preemptions increases hugely the number of different combinations of potential parallel interactions that may occur between tasks.

Another current area of weakness of global scheduling is the handling of critical sections. Intrinsically, global scheduling will incur greater overhead for lock and unlock operations than partitioned scheduling, because using light-weight local locking protocols for some critical sections is not an option. Moreover, the state of knowledge regarding suitable protocols for globally scheduled systems is behind that for partitioned systems. This author is unaware of any experiments with global scheduling analogous to those cited for partitioned scheduling in [26, 21, 41], but conjectures that when such experiments are done they will reveal that allocating global locks in scheduling priority order (as in [43, 42]) out-performs FIFO service if scheduling priorities or deadlines are applied globally. Likewise, global application of scheduling priorities should also result in reduced blocking if the SRP's highest-locker priority is assigned to global lock holders, rather than the M-SRP's total nonpreemptibility.

8 Cracks in the Foundations

Despite progress in theoretical understanding of real-time scheduling on multiprocessors, there is cause for concern about the validity of the theory for actual systems. The problem is that real multicore processors do not fit the assumptions of the models on which the theory is based.

One false assumption is that jobs running on different processors have independent execution times. Concurrent jobs can already interfere on a single processor, but the interferences occur around context switch points, which are not very frequent and can be bounded. On a multiprocessor, cache and memory bus conflicts between jobs can occur throughout the jobs' executions. Even predicting context switching costs is potentially more complicated on a multiprocessor, as the cost of a switch will depend on whether it includes migration between processors, whether the migration crosses cache domains, and what other tasks do to the cache between context switches. For example, [22] reports cases, on an 8-core processor with shared L2 cache, where the cost of migration was less than the cost of preemption. They attributed this effect to migration allowing a preempted task to resume executing sooner than if it had been forced to wait for its previous processor to become available again, and so incurring less loss of cache affinity. Besides concurrent cache contention and task migrations, other dynamic factors that can affect execution times include prefetching-hardware contention, memory-controller contention, and memory-bus contention. These are reported to account for variations in execution time of "60%, 80% and sometimes 100%" between executions of the same program [19].

[7] The standard Linux kernel maintains per-core task dispatching queues, and only performs migrations at longer intervals, for this reason.

A second false assumption is that the execution time of a job will be independent of which core executes it and when. One example is reported in [47], where an undocumented asymmetry with respect to memory bandwidth between the cores on an Intel Xeon 5345 quad-core processor resulted in differences of up to 400% in completion times of identical jobs running in parallel on different cores. Another example is the "Turbo Boost" feature of the Intel^{TM} Core i7^{TM} processor, which varies the execution speed of each core in response to developing hot spots within the chip [23].

The central false assumption, which underlies both of those above, is that one can schedule processors to meet deadlines effectively while entirely ignoring the on-chip bus and memory network. The emerging picture is that with more cores the primary scheduling bottleneck shifts from the processors to the resources that move information between them. Scheduling the cores without accounting for this network is naïve and unlikely to produce satisfactory outcomes.

A few efforts have been made at static WCET analysis that predicts and accounts for cache misses on multicore processors. The methods appear to be intrinsically limited, by growth of the number of combinations of potentially interfering concurrent computations that must be considered, to very simple task and hardware models. None appear to have been tested against actual performance of a real system.

Perhaps the most ambitious WCET analysis attempt so far is [33]. It assumes the workload is specified as a finite set of jobs assigned statically to processors, which are related by precedence constraints. The system is represented graphically as a UML-like "message sequence chart", which shows how jobs trigger releases of other jobs across processors over time. The example studied is a two-core processor with private 2KB direct-mapped L1 cache and shared set-associated L2 cache with associativity ranging from 1 to 4 and size ranging from 1KB to 16KB. The analysis only considers instruction cache. The tasks are assumed to be scheduled by priority, but given the way the analysis process iteratively refines estimates of earliest start times and latest completion times along with the job WCET estimates, it could just as well produce a static schedule. So, this work supports consideration of static scheduling.

It is going to be very difficult to find WCET bounds for multicore machines that are simultaneously trustworthy and not absurdly high, and it is likely to become more difficult as one moves from static to partitioned to global scheduling. The lack of good WCET estimates is likely to severely limit practical applications of hard-real-time scheduling theory for multicore processors. Obtaining useful guarantees of hard deadlines at significant levels of system loading may just be impracticable.

9 Predictions

Supposing that the claims above about multicore processors are valid – including growing difficulties with execution time variability and bottlenecks within the on-chip – where are we headed? With the acknowledgment that this is a case of piling speculation upon speculation, here are some predictions.

The end of hard real time? There will be increasing pressure to design real-time systems that are "softer" in order to accommodate the increasing difficulty of obtaining reliable WCET estimates and the resultant decrease in precision and trustworthiness of schedulability analyses. Where hard deadlines cannot be avoided, designers will need verify them under conservative assumptions. Softer real-time tasks will need to get by with statistical performance estimates, derived from testing and static analysis based on observed execution times. The system scheduling policy will need to enforce bandwidth limitations on all tasks accurately enough support the requirements of the tasks with hard deadlines.

If it turns out that processors are no longer the system bottleneck, and if power can be saved while they are idle, achieving high processor utilization will be less important. Queuing theory and the utilization-bound results suggest that as the number of cores and the number of tasks go up, work flow will increasingly approximate a fluid – so long as individual tasks are small. Dedicating a core to a task that is too large to fit this model or has a very strict deadline may be wiser than trying to push processor utilizations to some limit that is believed to be safe based on an unrealistic theoretical model.

As suggested in Section 5, the time may be coming when the processor allocation problem resembles the memory and disk allocation problems. If so, it may be time to adopt similar management strategies, such as allocating time in equal quanta to better fit the fluid model, and allowing a reasonable amount of spare capacity to even out the flow.

Threads considered harmful? There will be pressures to re-think the thread-based concurrent programming paradigm. The concept of a thread as the basic unit of concurrent programming and the idea of threads sharing their entire address space seem to be firmly entrenched in programming languages, operating systems, and programming culture. However, if the rules of microprocessor architecture are changing, it may be time for these conventions to change also.

We should look closely at the current thread model with respect to demand for on-chip network bandwidth, ability to handle fine-grained concurrency, and inappropriate scheduling constraints.

Threads enforce more serialization than is required for some applications. Thinking in terms of threads has led researchers in scheduling theory to constrain the jobs of each task to be executed serially, and use only one processor at a time. If these constraints stand in the way of fully exploiting the parallel execution possibilities of a multiprocessor, pressure will build to relax them.

With enough processors, it becomes profitable to split jobs into smaller units that can run in parallel. The ability to split a job requires forethought in the software's design and coding, but the number of ways in which a job is split is a performance-tuning and scheduling decision, which depends on the hardware configuration and the rest of the workload. It is not clear that such details should be hard-coded into the software. It is not clear, either, that fine-grained parallelism introduced solely as an opportunity for performance improvement should be expressed using the same programming constructs as are used to express mandatory elements of asynchrony and concurrency derived from the problem domain, or be burdened by their heavier-weight semantics.

Several problems with threads come from *implicit* sharing of their entire memory space. If the programming language and operating system make it easy for threads to share variables, programmers will access variables from multiple threads without careful consideration, or even without awareness of doing it. This is bad, since variable sharing has negative consequences for software reliability, performance, and schedulability analysis. The more shared memory accesses a task makes, the more load it puts on the data paths between processors, caches, and memory, and the longer and more variable its execution times become. The more variables are shared intentionally the more chance for errors in synchronization, and of course every instance of unintentional sharing is a ticking bomb.

What to do in Ada? It is a good time to take inventory of the Ada language, to see what it provides for constructing reliable highly concurrent real-time software using multicore processors. What does the language have that helps or hurts? It is good that Ada has a well developed concurrent programming model. However, some specifics may not be so good. Should the language should be changed, or should a particular style of usage be developed for multicore systems? Questions to consider include:

- How can we reduce casual memory sharing, and make intentional memory sharing more visible, with a view to modeling and managing the flows of data between cores?
- How can we design software that can make use of additional cores as they become available, without redesigning and re-coding, and in a way that is amenable to schedulability analysis?

Ada already has several features that make data sharing visible, including task entry parameters, protected objects, and the package sharing model of the Distributed Systems Annex. Would it be practical to program in Ada under the restriction that these be the *only* avenues for data sharing between threads? Would this be enough?

It might help to adopt an event-driven programming style, in which all computations are explicitly broken into jobs, which are queue-able data objects, executed by virtual servers. The servers could be implemented as Ada tasks. A system could be tuned by allocating multiple servers to a single queue, or having a single server serve multiple queues. Having the jobs carry much of their own data may reduce memory contention between cores. If the primary flows of data are via job objects, the flow of data between cores may be modeled and managed in terms of the flow of jobs. In this context, it may also be time to take a second look at the "featherweight task" model of [5] as well as other ways of expressing parallelism at a finer grain and with lower overhead than current Ada tasks.

These are just a few initial considerations. Inventive minds, with further practical experience using multicore processors, will surely think of more.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec. 1998.

[2] J. Anderson and A. Srinivasan. Mixed pfair/ERfair scheduling of asynchronous periodic tasks. In *Proc. 13th EuroMicro Conf. on Real-Time Systems*, pages 76–85, Delft, Netherlands, June 2001.

[3] B. Andersson. Global static-priority preemptive multiprocessor scheduling with utilization bound 38%. In *Proc. 12th International Conference on Principles of Distributed Systems (OPODIS 2008)*, pages 73–88. Springer, 2008.

[4] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Proc. 22nd IEEE Real-Time Systems Symposium*, pages 193–202, London, UK, Dec. 2001.

[5] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1):67–100, Mar. 1991.

[6] T. P. Baker. An analysis of EDF scheduling on a multiprocessor. *IEEE Trans. on Parallel and Distributed Systems*, 15(8):760–768, Aug. 2005.

[7] T. P. Baker. An analysis of fixed-priority scheduling on a multiprocessor. *Real Time Systems*, 2005.

[8] T. P. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. In *Int. Conf. on Real-Time and Network Systems*, pages 119–127, Poitiers, France, June 2006.

[9] T. P. Baker and S. K. Baruah. Sustainable multiprocessor scheduling of sporadic task systems. In *Proc. 21st Euromicro Conf. on Real-Time Systems (ECRTS 2009)*, pages 141–150, July 2009.

[10] T. P. Baker and M. Cirinei. Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks. In *Proc. 11th Int. Conf. on Principles of Distributed Systems*, pages 62–75, Guadeloupe, French West Indies, Dec. 2007. Springer.

[11] S. K. Baruah. Techniques for multiprocessor global schedulablity analysis. In *Proc. Real-Time Systems Symposium*, pages 119–128. IEEE Computer Society Press, Dec. 2007.

[12] S. K. Baruah. An improved EDF schedulability test for uniform multiprocessors. In *Proc. 16th IEEE Real-time and Embedded Technology and Applications Symposium*, Apr. 2010.

[13] S. K. Baruah. Schedulabilty analysis of global deadline-monotonic scheduling. Technical report, University of North Carolina, Dept. of Computer Science, 2010.

[14] S. K. Baruah and A. Burns. Sustainable scheduling analysis. In *Proc. 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, pages 159–168, Rio de Janeiro, Brasil, Dec. 2006.

[15] S. K. Baruah, N. Cohen, C. G. Plaxton, and D. Varvel. Proportionate progress: a notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[16] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proc. 17th EuroMicro Conf. on Real-Time Systems*, pages 209–218, Palma de Mallorca, Spain, July 2005.

[17] M. Bertogna, M. Cirinei, and G. Lipari. New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors. In *Proc. 9th Int. Conf. on Principles of Distributed Systems*, volume 3974/2006, pages 306–321. Springer, Dec. 2005.

[18] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Trans. on Parallel and Distributed Systems*, 20(4):553–566, Apr. 2009.

[19] S. Blagodurov, S. Zhuravlev, S. Lansiquot, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. Technical report, Simon Fraser University, 2009.

[20] A. Block, H. Leontyev, B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proc. 13th IEEE Embedded and Real-Time Computing Systems and Applications Conf.*, pages 47–56, Aug. 2007.

[21] B. Brandenburg and J. Anderson. A comparision of the M-PCP, D-PCP, and FMLP on LITMUS^{RT}. In *Proc. 12th Int. Conf. On Principles Of Distributed Systems (OPODIS 2008)*, pages 105–124. Springer, Dec. 2008.

[22] B. Brandenburg, J. M. Calandrino, and J. H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proc. 29th IEEE Real-Time Systems Symposium*, pages 157–169, Dec. 2008.

[23] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova. Evaluation of the of the Intel Core i7 Turbo Boost feature. *IEEE Workload Characterization Symposium*, 0:188–197, 2009.

[24] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, Feb. 1978.

[25] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proc. 22nd IEEE Real-Time Systems Symposium*, pages 73–83, Dec. 2001.

[26] P. Gai, M. D. Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the janus multiple-processor on a chip platform. In *Proc. 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 189–198, May 2003.

[27] J. Goossens, S. Funk, and S. K. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real Time Systems*, 25(2–3):187–205, Sept. 2003.

[28] N. Guan, Z. Gu, Q. Deng, S. Gao, and G. Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In *Software Technologies for Embedded and Ubiquitous Systems*, volume 4761/2007, 2007.

[29] N. Guan, M. Stigge, W. Yi, and G. Yu. Fixed-priority multiprocessor scheduling with Liu & Layland's utilization bound. In *Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr. 2010.

[30] Intel Corporation. Terra-scale computing program. `http://techresearch.intel.com/articles/Tera-Scale/1421.htm`, 2009.

[31] K. Jansen and L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. In *Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 490–498, 1999.

[32] S. Kato and N. Yamasaki. Portioned EDF-based scheduling on multiprocessors. In *Proc. 8th ACM International Conference on Embedded Software (EMSOFT 2008)*, pages 139–148, New York, NY, USA, 2008. ACM.

[33] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *Proc. 30th IEEE Real-Time Systems Symposium*, pages 57–67, Dec. 2009.

[34] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[35] J. M. Lopez, J. L. Diaz, and D. F. Garcia. Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling. *IEEE Trans. Parallel and Distributed Systems*, 15(7):642–653, July 2004.

[36] J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia. Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems. In *Proc. 12th EuroMicro Conf. Real-Time Systems*, pages 25–33, 2000.

[37] V. B. Lortz and K. G. Shin. Semaphore queue priority assignment for real-time multiprocessor synchronization. *IEEE Trans. Software Engineering*, 21(10):834–844, Oct. 1995.

[38] G. Lowney. Why Intel is designing multi-core processors. In *Proc. 18th ACM symposium on Parallelism in Algorithms and Architectures (SPAA 2006)*, pages 113–113, New York, NY, USA, 2006. ACM.

[39] L. Lundberg. Analyzing fixed-priority global multiprocessor scheduling. In *Proc. 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 145–153, San Jose, CA, USA, 2002. IEEE Computer Society.

[40] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proc. 9th IEEE Real-Time Systems Symposium*, pages 259–269, 1988.

[41] J. Ras and A. M. K. Cheng. An evaluation of the dynamic and static multiprocessor priority ceiling protocol and the multiprocessor stack resource policy in an SMP system. *Proc. 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 13–22, 2009.

[42] C. S. C. P. Santiprabhob and T. P. Baker. Reducing priority inversion in interprocessor synchronization on a fixed-priority bus. Technical report, Florida State University, Dept. of Computer Science, Tallahassee, FL, Aug. 1991.

[43] P. Santiprabhob, C. S. Chen, and T. P. Baker. Ada run-time kernel: The implementation. In *Proc. 1st Software Engineering Research Forum*, pages 89–98, Nov. 1991.

[44] B. Sprunt, L. Sha, and L. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[45] A. Srinivasan and S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84:93–98, 2002.

[46] J. A. Stankovic, M. Spuri, M. D. Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28:16–25, 1994.

[47] I. Tuduce, Z. Majo, A. Gauch, B. Chen, and T. R. Gross. Asymmetries in multi-core systems – or why we need better performance measurement units. In *Exert 2010: The Exascale Evaluation and Research Techniques Workshop*, Mar. 2010.

[48] J. Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. Softw. Eng.*, 19(2):139–154, 1993.