Architecture-independent predictable Java for multi-core platforms

Filip Pizlo Lukasz Ziarek Jan Vitek
Purdue University

October 25, 2008

Introduction

Cyber-physical systems (CPS) involve embedded computing devices that interact with physical processes. CPS require a software and hardware architecture that not only delivers good performance but also good predictability. In particular, CPS software must maintain good responsiveness while using a bounded amount of memory, processor, and power resources. Most importantly, CPS software must be *real-time*; i.e. the timing properties are part of the correctness specification.

In the past, advancements in computing applications arose from increased processor performance. A faster processor meant less time to complete a given task, allowing for a single machine to take on more roles. But future advancements in computing applications will come from the addition of processor cores. This presents a challenge to the CPS designer, since programming on multi-processor and multi-core platforms is much harder than programming on a traditional uniprocessor. In this work we outline the challenges presented by multi-core systems and show how they will be addressed through the introduction of Java.

Fundamental Limitations

The performance increases in singe-core architectures that we have seen over the preceding decades are coming to an end; future gains will come from additional cores. This presents the following challenges for CPS, as current engineering best-practices rely heavily on the implicit guarantees of uniprocessor architectures.

- Multi-core systems introduce additional impediments to predictability. Whereas a memory access on a uniprocessor may *at worst* result in a cache-miss, multi-core architectures implicitly synchronize memory between cores, leading to unpredictable slow-downs when memory is shared. Hence, while predictability is already a challenge on uniprocessors, multi-core systems exacerbate the problem.
- Predictability is further complicated when a task on a given core prevents the execution of a task on a different core. This occures from explicit synchronization via locks either explicitly present in the program or used by the operating system. Guaranteeing utilization of processor resources becomes difficult – how do we bound the time that a core is idle? By comparison, even in the presence of pervasive synchronization, it is easy to guarantee that a single-core system stays utilized.
- Scheduling on a single-core is a dramatically simpler problem than scheduling on multi-core systems. To ensure fair utilization of a multi-core systems, tasks may have to be migrated from one core to another; this introduces difficult-to-bound costs that may be prohibitive in practice.

Most Important Research Challenges

Current research on prectable scheduling for multi-core architectures has been limited to micro-benchmarks scaling to at most 32 cores[2]. But 864 core SMP systems are already available [5]. There is a fundamental disconnect between leading edge scheduler research and the systems on which these schedulers may be

deployed. Synchronization between real-time tasks in a multi-core setting has an additional impact on scheduler variability. It is unclear what effect migration, synchronization, and communication in real applications executed under a multi-core sytem has on WCET for a given scheduler.

Memory management (MM) has always been a difficult problem, especially in safety critical systems. It is difficult to ensure that the manner in which memory is managed does not break the correctness of a system. In the case of multi-core systems, MM faces additional challenges. First, MM on multi-core systems requires additional synchronization, which impedes predictability. Second, if the memory architecture involves non-uniform memory access, the MM system must be made aware of thread-processor-memory mapping. That is, if a given thread allocates memory, we wish for the memory allocated to be local to the processor that the thread is running on. Otherwise we must concede that every memory access may require following the longest path through the memory interconnect.

Synchronization on real-time platforms has always been a challenge. In the case of a uniprocessor system, the problem is priority inversion: if a high-priority task is blocked due to a lock held by a low-priority task, a middle-priority task may execute, starving the high-priority task. Traditional real-time systems avoid this problem using techniques such as priority inheritance or priority ceiling emulation. But on a multiprocessor, we face additional problems. Consider the case where the highest priority threads on processors A and B share a resource, and must contend on a lock to use it. Which thread will get the lock? High-performance locking protocols may take into account priority on a *single* processor, but taking priority across processors into account is much harder. Additionally, high-performance multi-core systems typically use atomic instructions such as CAS to enable lock-free communications. But these primitives are implemented using hardware locks that not priority aware – architectures do not guarantee which of the processors contending on the lock will proceed first. What we need is a platform that is vertically priority-aware. This may come in the form of either a hardware solution, or possibly a programming language one.

Promising Directions

On a uniprocessor, type-safe languages such as Java present implementation challenges. However, on large-scale multi-core architectures, we believe Java will be the solution.

Two philosophies towards managing multi-core support are likely to reach maturity over the coming years. A transparent approach, in which scheduling, memory management, and synchronization will behave similarly to current uniprocessor systems is likely to see continued development. A more rigorous approach, in which the programmer is given explicit control over how hardware resources are used, is likely to gain traction as well. Both approaches will increasingly involve higher-level languages such as Java.

Transparency is best achieved through the use of a platform such as a Java virtual machine. This approach is attractive because we wish to offset the burden of managing processor resources from the programmer. For example, a Java virtual machine can manage memory automatically, stream-line the handling of scheduling, and provide synchronization through the use if Java's primitives. A number of challenges exist in migrating such features to CPS. First, such features need to be scalable. Secondly, garbage collection itself must be predictable and fast. Though much work has been done [4, 3], some challenges remain. We expect that within five years, fully deterministic multi-processor real-time garbage collection will be the norm, eliminating the need to manage memory manually, and thus eliminating the need for type-unsafe languages such as C.

Unfortunately, it is unclear that any of the available scheduling, synchronization, or memory management techniques will scale well beyond 32 cores. Emperical evidence suggests that programs may perform better with a modular heap [3, 1], which would be ideal for scaling to > 32 cores. Programming models such as Heaplets [3] and Flexotasks [1] allow the programmer to give additional information to the runtime system about the structure of memory. Flexotasks provide memory structure information rich enough to allow for a deterministic thread-memory-processor mapping. They also provide a clean way of reasoning about communication between components, which can be used to make synchronization more predictable. While these systems are currently deployed on uniprocessors, they will likely be the mechanism of choice for eliminating the unpredictability traditionally associated with multiprocessors.

The current approach in Flexotasks is to mandate time-space partitioning of modules. Each module has its own schedule and private memory that is isolated from the schedules and memories of other modules. If

enough cores are available, such a system could be deployed with each task on its own core. This eliminates the scheduling problem entirely, leading to performance boosts by eliminating scheduling overheads (such as timer interrupts), while also making the system easier to analyze – the schedulability of task T is independant of the schedulability of tasks unrelated to T.

Whatever the philosophy, tasks on separate processors must be able to communicate in a predictable manner. Whether the communication is done using messages, locks, or lock-free algorithms, we must ensure that in the case of contention, priorities are enforced. Two solutions exist: either we change the hardware to be aware of priority in the case of contention, or we use a virtual machine to enforce priority guarantees on any use of shared resourced. The latter can easily be done in systems like Java, where code already runs in a controlled environment.

It is noteworthy that under all of these approaches, the features of Java that have traditionally been seen as performance and predictability bottlenecks now become the means for overcoming the predictability burdens of multiprocessors. In particular, garbage collection can naturally scale to many cores. While manual memory management requires each task to devote some amount of time to managing memory, a garbage collector can guarantee that memory requests are always fast, and memory reclamation can be offloaded to separate cores.

The type-safety of Java is the cornerstone of the Flexotask approach. For Flexotasks to offer their guarantees, we need a space partitioning. This is difficult to achieve in C without relying heavily on memory management hardware. In Java, the partitioning can be achieved entirely in the compiler, incurring no run-time cost. Thus, if the solution to the problem of large-scale multiprocessors is to rely on partitioning, Java becomes the more sensible approach.

Milestones for 5, 10, and 20 years

In five years we believe that RTGC for embedded real-time systems will be a viable, efficient technology that will be widely utilized.

In ten years we believe that Java virtual machines will be able to provide predictable parallelism through runtime system provided schedulers, communication, and synchronization in combination with new programming model such as Flexotasks.

In twenty years we belive that additional parallism will be extracted automatically through the use of compiler analysis, dynamic anaylsis, as well as profiling. New programming models will improve the quality of extracted parallelism.

Filip Pizlo is a 5th year graduate student at Purdue University working on programming languages and runtime environments for real-time and safety-critical systems. **Lukasz Ziarek** is a 5th year graduate student at Purdue University working on programming languages, virtual machines, and compilers. **Jan Vitek** is an Associate Professor in the Department of Computer Science at Purdue University, whose work focuses on programming languages, virtual machines, and program analyses real-time embedded and distributed systems.

References

[1] Joshua Auerbach, David F. Bacon, Rachid Guerraoui, Jesper Honig Spring, and Jan Vitek. Flexible task graphs: a unified restricted thread programming model for java. In *The 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 1–11, New York, NY, USA, 2008. ACM.

[2] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, December 2008.

[3] Filip Pizlo, Athony L. Hosking, and Jan Vitek. Hiearchical real-time garbage collection. In *Proceeedings of ACM SIGPLAN/SIGBED 2007 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 123–133, 2007.

[4] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *Proceedings of PLDI 2008*, Tucson, Arizona, June 2008.

[5] Azul Systems. Vega 3 series compute appliance. `http://www.azulsystems.com/`.