Analyzing manycore OS design aspects in NIX

Abstract

Francisco J. Ballesteros
Gorka Guardiola
Enrique Soriano
Universidad Rey Juan Carlos, Madrid
{nemo,paurea,esoriano}@lsub.org

Noah Evans *
Jim McKie
Alcatel-Lucent Bell Labs
{npe,jmk}@plan9.bell-labs.com

Charles Forsyth
Vita Nuova
forsyth@vitanuova.com

Ron Minnich
Google
rminnich@gmail.com

NIX [2] is a novel OS designed for current manycore machines, which includes mechanisms to assign different roles to (heterogeneous) cores. In addition, NIX includes a NUMA aware memory allocator suited for new 64-bit x86 processors. The core roles available in NIX are: (i) *Time-sharing Core* (TC), which is a common core running kernel and user code in a time sharing fashion; (ii) *Application Core* (AC), which is a core running user code without any interrupt (even without clock interrupts); and (iii) *Kernel Core* (KC), which is a core that only runs kernel code on demand. The cores communicate by sending *active messages* that include a function to be executed, together with its arguments. For more details, please refer to [2].

We are currently working on different fronts, while performing an analysis of the design points in current manycore focused systems software projects. In order to perform such analysis, we have built a benchmarking and measuring framework which instruments the kernel, leverages the hardware facilities, and automates the benchmarks. The analysis includes performance measurements for running benchmarks based on real applications (not just micro-benchmarks), and statistical studies of the performance data provided by the hardware counters of our machines, and the data provided by an instrumented kernel. The performance data includes the L2 hits and misses, TLB misses, DRAM accesses, data/instruction cache refills and misses, cycles wasted in locks, number of system calls, number of context switches, and others. The results of these measurements will be instrumental in defining future directions of the project. We present some early results from a 32-core AMD K10 machine, which are open to discussion. This poster aims at reporting some of our preliminary results and obtaining some feedback from the community regarding the key design challenges we are facing:

- ***Role assignment to cores.*** The inherent flexibility for specializing cores of NIX makes it particularly suitable for the future heterogeneous multi-core chips. NIX permits to change the role of a core dynamically, that is, a core can be restarted with a different role on demand.

  We are also performing a quantitative analysis to detect the useful roles for different kinds of applications. For example, the *Application Core* (AC) role is appropriate for HPC applications, but it is not suitable for I/O-intensive applications. We are working towards automatic core role provisioning and management for applications matching different usage patterns (e.g. number of expired scheduling quantums or I/O usage).

  Work in progress also includes the implementation and evaluation of new core roles. For example, we are implementing a new role named *Exclusive Core* (XC). This new role wires a core to a process (instead of wiring a process to a core, as usual). An XC only runs usercode for a single process without interrupts, like the *Application Core* (AC) role [2]. The key difference is that an XC also runs kernel code exclusively for this process. When a process running on an AC requires a kernel service (e.g. it performs a system call), it has to use inter-core communication (ICC) mechanisms to perform the operation on a

* PhD student.

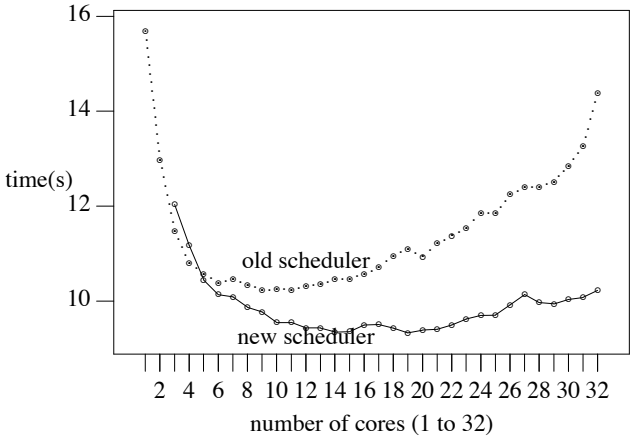

Figure 1. SMP ("old") vs AMP ("new") schedulers experiment.

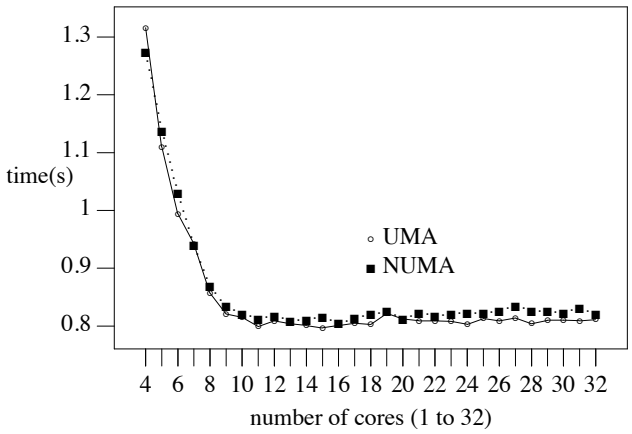

Figure 2. NUMA vs UMA allocators experiment.

Time-sharing Core (TC). When a process running on an XC requires a kernel service, it just runs kernel code.

- ***Scheduling.*** We are experimenting with different symmetric and asymmetric multiprocessor scheduling schemes to find out which ones make sense for current manycore architectures. In order to evaluate them, we are performing a quantitative analysis using different numbers of cores and schedulers, on different memory socket domains. Recent experiments have provided some interesting results in this regard. For example, we found that for one of our benchmarks (described later), using 4 cores with standard symmetric multiprocessing (SMP) scheduling is faster than using 8, 16 or 32 cores. That is, in this experiment, adding more cores implies a slowdown past some point, which is somewhat non-intuitive.

  We are experimenting with asymmetric multiprocessing (AMP) schedulers, in which only one core (core zero) runs the scheduler, to assign the rest of the cores to the running processes. Our current results show that AMP schedulers perform better than SMP schedulers for our benchmarks. For example, Figure 1 shows the results of an experiment comparing SMP ("old scheduler" in the figure) with AMP ("new scheduler" in the figure). The figure shows the average time for ten executions of the benchmark. The benchmark for this experiment compiles the NIX kernel creating as many concurrent processes as necessary to compile all the source files (executing the C compiler and the assembler). We can observe that the performance of the SMP scheduler is affected negatively when using more than 10 cores (TCs). We do not have a theory to explain why the performance of the AMP scheduler degrades when it uses more than 15 cores, but we are conducting experiments to learn the causes of these effects.

  Another interesting result provided by the experiments shows that, for our benchmarks, the difference between using the NUMA aware allocator and a UMA allocator is not significant when the number of cores (TCs) grows. In some cases, the UMA allocator performs better than the NUMA allocator. Figure 2 shows the results for an experiment measuring the time required to compile the NIX kernel using a RAM file system. Please refer to [1] for the whole evaluation data.

- ***Zero-copy memory framework.*** Zero-copy I/O may lead to significant performance improvements due to avoidance of unnecessary data copies within data paths. Many such frameworks have been implemented in the past but are not yet used in production. We are trying to design and implement yet another one for NIX but keeping it simple so that it might be effectively used in practice, unlike others. In general we try to follow these design guidelines: (i) data belongs to a single process (or kernel entity) at a time, as data is passed around, ownership may change; (ii) metadata used for data handling (e.g., reference counters, if used) must be accessible only to the entity responsible for allocating the data; and (iii) copies are not avoided at all cost: if it is not clear and simple how to avoid copying, data is copied. This framework will be used by IX, a new high performance file system which is also work in progress.

The NIX source code is available on Google Code: `http://code.google.com/p/nix-os/`

This work was funded in part by the Comunidad de Madrid grant S2009/TIC-1692 and the Spanish Government grant TIN2010-17344.

References

[1] F. J. Ballesteros and G. Guardiola. Rosac-2012-1. tech. report. nix measurements., 2012.

[2] F. J. Ballesteros, N. Evans, C. Forsyth, G. Guardiola, J. McKie, R. Minnich, and E. Soriano. Nix: a case for a manycore system for cloud computing. *Bell Labs Technical Journal*, 2012. To appear.