Fiasco.OC on the SCC

Markus Partheymüller, Julian Stecklina, Björn Döbel
{mpartheym,jsteckli,doebel}@tudos.org
Operating Systems Group, TU Dresden

***Abstract*—Our initial goal was to port the Fiasco.OC microkernel to the Single-Chip Cloud Computer in order to use it as an experimentation platform for our research. In this paper we describe the few hardships we encountered during this porting work and our solutions for those problems. With our kernel running on top of the SCC, we evaluated message passing performance between cores and came across a cache-related issue that can seriously decrease message-passing performance.**

I. Introduction

The Single-Chip Cloud Computer (SCC) experimental processor is a 48-core 'concept vehicle' created by Intel Labs as a platform for many-core software research. To facilitate our research in this area, we decided to port the Fiasco.OC microkernel developed in our group to the SCC. Establishing a working environment to run Fiasco.OC on the SCC will then pioneer experiments concerning all kinds of many-core related research topics, including energy saving, inter-core message passing and trusted, robust server applications.

With the SCC consisting of x86 CPUs and Fiasco.OC already available for this CPU architecture, most of the porting work was straightforward. However, on our way we encountered a couple of issues related to the non-standard SCC hardware features. In this paper we present these issues and describe our solutions.

We then describe a first experiment we conducted to evaluate message passing performance between instances of Fiasco.OC running on different SCC cores. This experiment led us to discover a caching-related hardware issue, which for unaware developers may pose a serious performance problem.

II. Fiasco.OC and L4Re

Our work takes place in the context of the Fiasco.OC microkernel [1]. Fiasco.OC is the only piece of software running in privileged processor mode and following the philosophy of L4 microkernels provides only mechanisms for constructing an operating system but does not implement any policies, such as resource management.

The main mechanism provided by Fiasco.OC are capabilities [2], which can be viewed as kernel-protected references to objects. The objects themselves are either implemented in the kernel (e.g., tasks, threads, interrupts) or by user-level applications (e.g., memory managers, device drivers, protocol stacks). An object's functionality can be used by clients possessing access to an object's capability through Fiasco.OC's invoke() system call. The close relation between objects and capabilities in Fiasco.OC also motivated choice of the .OC suffix.

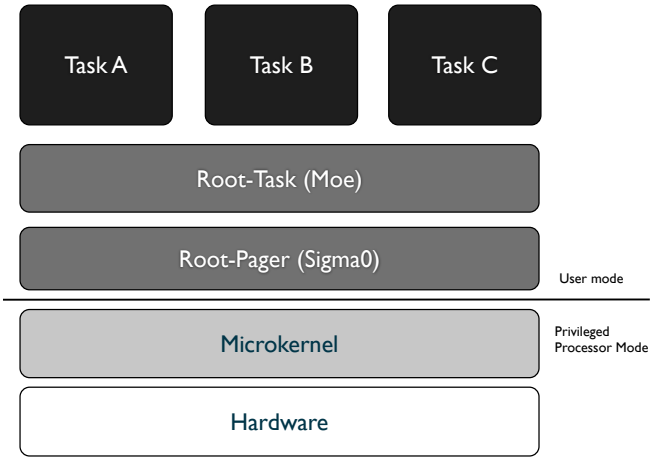

Fig. 1. Basic structure of an L4Re based System

Policies, such as resource management decisions, are implemented on top of Fiasco.OC as user-level applications. The most basic services, such as a C library, application memory managers, and a loader for ELF binaries, are implemented in a set of libraries and servers called the L4 Runtime Environment (L4Re) [3]. Apart from that, our public SVN repository contains a wide range of additional software such as a GUI service, ports of various libraries (Qt, libjpeg, freetype), as well as software packages, such as sqlite and Valgrind [4].

Booting an L4Re setup on Fiasco.OC involves first bootstrapping the microkernel itself. Thereafter, the sigma0 root pager is booted, which initially owns all resources in the system. However, sigma0 only serves as the user-level pager for the next application in line: the root task, called Moe. Moe then provides essential services required by user applications, such as a program loader, an address space manager that is injected into every task, and a memory allocator serving regions of virtual memory. An overview of this structure is given in Figure 1.

Fiasco.OC and L4Re provide all means necessary to implement user-level applications. One of the most demanding applications running on top of our system is L4Linux [5], a para-virtualized version of the Linux kernel running as a user-level application on top of Fiasco.OC. With this approach we are able to run arbitrary Linux binaries on top of our system, while it still allows for all the benefits of virtualization: as the Linux kernel is running as an unprivileged user application, it becomes isolated from the rest of the system and outside

services may impose resource constraints as well as enforce security policies. Furthermore, it is possible to increase resource utilization by running multiple instances of L4Linux on the same core.

III. Fiasco on the SCC

The SCC being an x86 architecture based Pentium processor system suggests to use the existing x86 port of Fiasco. However, the absence of typical hardware devices such as the BIOS, the Programmable Interval Timer (PIT), a keyboard controller and a graphics card did not allow to run the kernel unmodified. In addition, I/O requests issued from the cores are routed to the management PC, because there are no local devices that can handle them. The x86 boot process relies on these devices and would therefore fail immediately.

As the SCC allows running dedicated OS instances on every core, we decided to initially port Fiasco.OC in a way that each core executes one Fiasco.OC instance and explore how the SCC's specific communication mechanisms can be used for messaging between the instances. In later experiments we plan to also explore the concept of a *Single System Image*.

To cope with the SCC's lack of firmware, we implemented an sccKit application called *sccLoad*. This application is responsible to set up a multiboot-compliant boot environment, which provides a memory map, a couple of register values, and a kernel command line. From then on, it is possible to boot the kernel using its native x86 boot code.

While booting Fiasco.OC, a timer calibration is performed, which calculates the ratio of clock cycles (accessible through the TSC register) to time values like nanoseconds. This is usually done using a second timer with configurable frequency. By setting this frequency to a known value and measuring the clock cycles during a fixed time interval, the ratio can be derived. On the SCC a second timer (e.g. the PIT) is missing and therefore there is no way for the core to determine the ratio on its own. To address this issue two solutions are possible: The first one is to calculate the ratio beforehand and hard-code it into the kernel. However, this assumes a fixed core frequency which is not necessarily the case on the SCC because of frequency scaling mechanisms. The second solution is to pass the core frequency as a command line parameter. It allows for this flexibility and was therefore preferred.

Beside the boot environment, *sccLoad* emulates a basic serial console using an I/O handler and can currently run, monitor and even control (via keyboard input) multiple instances of the kernel ELF [6] binary from the management PC. The I/O handler ensures that all I/O related code can be used without modification. When we started our work, the Fiasco.OC kernel was not able to use the Local APIC as configurable source for interrupts other than the timer. However, the SCC provides inter-core interrupts through the Local APIC and therefore, we needed to modify Fiasco.OC. We added an additional initialization section to the kernel binary, which sets up the Local APIC to trigger an interrupt vector directly representing the inter-core interrupt requests.

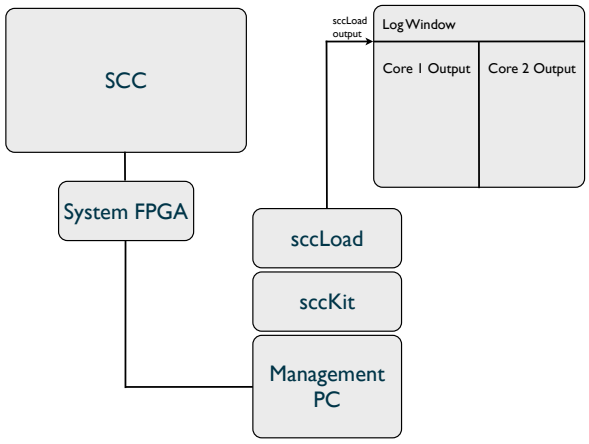

Fig. 2. Environment setup for monitoring two Fiasco.OC instances

Further addtional SCC features include the new MPBT memory type and the message passing buffers. To support these, we disabled support for *Page Size Extension* in the kernel and at the same time enabled the new MPBT feature. In the current state of our port, the addresses of the Message Passing Buffer (MPB) and the control register buffer (CRB) are not provided as regular RAM, but instead hardcoded into the kernel, assuming the standard memory layout described in the EAS. This enables us to view these memory regions as I/O memory. An L4Re server called *io* then provides these locations as flexibly mappable I/O regions.

While working on the *io* server, we discovered an issue that arises when the MPB is declared as uncacheable [7]. When doing so, reading data from MPB memory results in the first eight bytes of the memory content being repeated four times. The remaining 24 bytes of each cache line are inaccessible. To circumvent this, we had to make the *io* server work with the MPB as cacheable memory, which had not been necessary for any previous hardware resources managed by this server. While this combination of settings is not a usual use case for the MPB, it would have been interesting to examine the performance of shared DDR memory tagged as MPBT, because the need for an L2 cache flush routine could be eliminated while using the Write Combine Buffer to circumvent performance issues caused by *non-allocate-on-write*.

Altogether we now have an environment that can run basically any ELF binary produced by the L4Re build infrastructure, for example a Fiasco microkernel with a message passing application or even multiple instances of L4Linux.

IV. Preliminary results

As a first experiment on top of our newly ported OS infrastructure, we tried to evaluate message passing performance between different cores. We started with two cores, both located in tile [0,0], where core 1 sends a 32 MiB large message (separated into packets of 4096 bytes) to core 0 through the MPB. Core 0 copies the content into its own

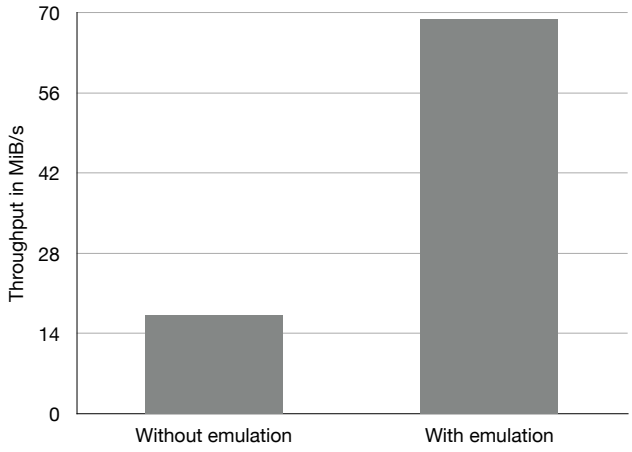

Fig. 3. Performance impact of non-allocate-on-write for `memcpy`.

memory and acknowledges the transfer to core 47, which measures the time that is needed to transfer the whole 32 MiB. The left bar of Figure 3 shows the obtained throughput result for the standard frequency configuration (533/800/800).

Our experimental data shows a significant impact of a cache-related hardware design choice. The cache property, often referred to as *non-allocate-on-write*, affects the performance of message passing or rather the actual memory accesses. When a program causes a write miss by writing to a memory location that is not present in the cache hierarchy, the caches do not allocate a new cache line for the write request. The request is instead passed on to the memory controller. In the worst case of byte-wise writes, each write issues an individual transaction to main memory.

The significance of this performance decrease can be demonstrated by implementing a software emulation of *allocate-on-write*. For a `memcpy` operation, before writing to a destination address, the location is read. This happens every 32 bytes, thus causing the caches to allocate a cache line for each address the operation will write to. We implemented this emulation and the right bar in Figure 3 shows, that using software emulated allocate-on-write significantly improves the message-passing throughput.

The results of these experiments will be evaluated to determine our next steps concerning message passing on top of Fiasco.OC. We did not intend the application to become a real message passing framework, but rather use it as a prototype to assess the difficulties and opportunities encountered when implementing communication software for the SCC in our microkernel environment.

V. Future Work

Currently we are conducting preliminary experiments exploring the inter-core message passing facilities in terms of performance and implementation complexity. We expect these experiments to give us results helping to decide if and how we should continue our work on message passing. Choices include a combination of a message passing library and an L4Re server multiplexing the MPBs as well as a RCCE port. Related to message passing is the performance issue mentioned above, leading to the question how to compensate for this shortcoming of the cache architecture and where to implement the improvement.

Also in connection to message passing is the inherent security issue when allowing all cores to access every memory location with full rights. In the standard layout, each core can read and modify all configuration registers, MPBs and DDR memory. By modifying its own LUT entries it can basically control the entire system. When the MPB is used for communication, there is no means to prevent cores from tampering with MPB data once they have access to the MPB locations.

Exploring ways to establish security measures could be an interesting topic. For example, one could think of sandboxing cores by not giving them access to configuration registers. This, of course, would render inter-core interrupts, which are implemented using those registers, impossible. A slightly weaker restriction, allowing access to all configuration registers but the cores' own one, would enable interrupts again but also gives the cores the ability to mess with other cores.

Instead of the configuration registers, also the recently added functionality of a global interrupt controller could be used along with the other additional features (global timestamp counter, global atomic increment counter), which are currently not implemented in our software.

In addition to covering performance issues, we also believe the SCC to be a good platform to research energy-related issues, because it provides regulators allowing to modify tiles' frequency and voltage settings. These regulators also allow for controlled *undervolting*, which may lead to energy savings as well failures with yet unknown characteristics. We plan to evaluate the concrete manifestations of such failures and tradeoffs between power saving and fault probability in future experiments.

VI. Conclusion

Although porting Fiasco.OC to the SCC seemed straightforward, we have encountered several problems that made the port more difficult than expected, such as bugs in the sccKit and cache misbehavior. Nevertheless we could establish a working environment for future experiments with microkernels on many-core systems in order to investigate particularities and problems that can occur on such systems, as well as possible approaches to solve them.

Acknowledgments

The authors would like to thank Adam Lackorzyński and Alexander Warg for sharing their Fiasco.OC knowledge. Additional thanks go to Intel and the MARC community, especially Ted Kubaska, Jim Held, and Michiel W. van Tol, who provided insights during discussions in the community forum.

REFERENCES

[1] TU Dresden OS Group, "Fiasco.OC microkernel," http://os.inf.tu-dresden.de/fiasco/, 2010.

[2] A. Lackorzynski and A. Warg, "Taming Subsystems: Capabilities as Universal Resource Access Control in L4," in *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*. Nuremberg, Germany: ACM, 2009, pp. 25–30.

[3] TU Dresden OS Group, "L4 runtime environment," http://os.inf.tu-dresden.de/l4re/, 2010.

[4] A. Pohle, B. Döbel, M. Roitzsch, and H. Härtig, "Capability wrangling made easy: debugging on a microkernel with Valgrind," in *VEE '10: Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. New York, NY, USA: ACM, 2010, pp. 3–12.

[5] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The performance of μ-kernel-based systems," in *Symposium on Operating Systems Principles*, Saint Malo, France, 1997, pp. 66–77.

[6] "Tool Interface Standard - Executable and Linkable Format," http://www.rcollins.org/intel.doc/Tools.html, 1998.

[7] M. Partheymüller, "Strange behaviour when reading MPB," http://communities.intel.com/message/133047, 2011.