UPPSALA
UNIVERSITET

IT 14 068
Examensarbete 15 hp
November 2014

Deployment and Profiling of L4Re on an ARM Cortex A Platform

Michal Marciniewski

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Deployment and Profiling of L4Re on an ARM Cortex A Platform

Michal Marciniewski

The use of multicore processors introduces problems in the context of hard real-time systems. This is because real-time applications that are running in parallel on different cores may access the same memory location at the same time. Then inter-core contention over memory occurs and may yield unexpected waiting times being injected into the overall execution times of real-time applications. In this thesis the FIASCO.OC microkernel with an L4 Runtime Environment was deployed on a quad-core ARM cortex A9 platform, building the basic setup for measuring the interference resulting from memory contention of applications running at the same time on different cores. For the actual measurement of the expected deviation in execution times of applications, a benchmarking real-time application targeted at the automotive industry has been executed in parallel to a data-intensive application in this case as a media player. An average slowdown of 0.37%, with data-cache enabled, and 4.60% with data-cache disabled, was obtained by comparing a reference value of a non-interfered run with a run of the benchmark with interference. The deployed system can in future work serve to measure the impact of implementing various memory budgeting and scheduling techniques for coping with memory contention in the setting of real-time applications and multicore processors.

Handledare: Jonas Flodin
Ämnesgranskare: Kai Lampka
Examinator: Olle Gällmo
IT 14 068

Contents

1 Introduction

The use of multicore processors has vastly increased during the last decade yielding a significant boost in computational performance. [1] This has however also introduced new problems. One such problem is the trade-off between being able to fully utilize resources by sharing them and being able to easily guarantee timing correctness [2], by using core specific resources. This is because task execution times become difficult to predict, when sharing resources, as additional waiting times for accessing shared resources are injected into the tasks' execution times. These waiting times are difficult to predict due to the fact that they highly depend on the applications corunning on other cores and accessing shared resources. While the number of cores integrated into every chip is steadily increasing, thereby increasing the possible degree of computational parallelism, the trend of increasing parallelism is not shared by all other components such as memory controllers, memory banks, higher level caches or memory buses [3]. This can become a crucial bottleneck to memory intensive software deployed on multicore processors, which may be particularly severe for real-time systems that may miss their deadlines due to the unpredictability of memory accesses performed by other cores executing in parallel [4].

1.1 Purpose

The purpose of this thesis is to measure the performance of the L4 Runtime Environment (see section 2.4) deployed on top of a ARM Cortex A platform [5] and the impact on performance caused by interference resulting from running memory intensive software on another core in parallel.

1.2 Delimitations

Measurements in this thesis are limited to a Freescale SABRE Lite i.MX6 board with a ARM Cortex-A9 Quadcore CPU based on the ARMv7 architecture [6]. In order to test performance impact, the *AutoBench* [7] automotive benchmark suite from the *Embedded Microprocessor Benchmark Consortium (EEMBC)* [8] was used as reference.

2 Background

Computationally intensive tasks that can be parallelized benefit greatly from the use of multiple cores. Parallelization introduces a requirement of memory sharing in order to maintain data integrity and access control by methods such as semaphores or locks. It is the means by which threads may communicate and distribute workload among each other.

Several separate tasks may also be run simultaneously on separate cores. Shared memory for such tasks comes with both advantages and disadvantages. Sharing resources forces tasks to contend over them and may potentially lead to performance degradation for all tasks, in comparison to sequential execution of tasks. Using task or core exclusive memory relieves this problem, but introduces another; unused memory can not be used by other tasks or cores which may have use for additional memory. Thus a potential underutilization of memory resources may occur.

Generally, the use of exclusive memory is infeasible for a larger market-share due to the limitations imposed on the parallelization of single tasks, which is a key advantage of multicore processors.

2.1 DRAM organization

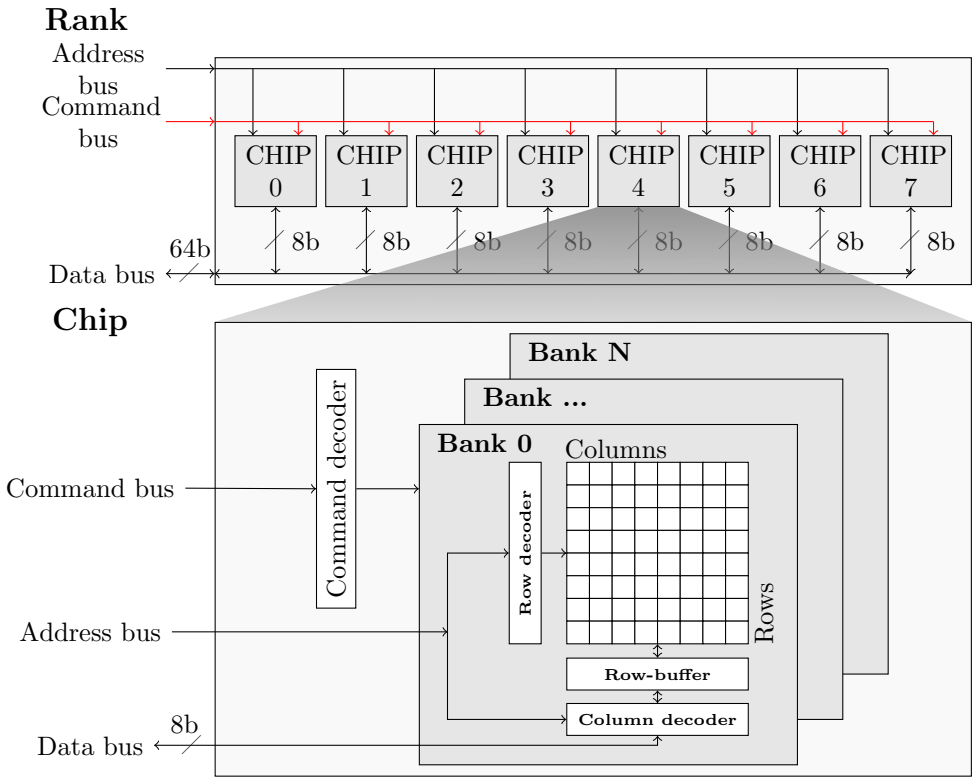

Figure 1: A DRAM System.

Dynamic random access memory (DRAM) [9] is a commonly used type of main memory. It consists of a series of ranks. Each rank in turn consists of a set of chips with narrow data interfaces, combined to a wider data interface. Each chip consists of multiple banks that allow for parallel servicing of data stored in separate banks. Banks in turn are two-dimensional data arrays of cells storing bytes. See figure 1, for an example of how the organization might appear. When accessing a cell in a bank, an entire row is brought into a row-buffer, also called opening a row, before the column containing the cell is retrieved from the row. If the requested data already resides in an open row, the data is directly and quickly retrieved from the row-buffer, also known as a row hit. If, however, the requested data is not present in the row-buffer, a row conflict occurs and a time-costly precharge operation is executed, closing

the row present in the row-buffer and activating a new row containing the requested data, before the actual retrieval can be performed.

2.2 Shared memory resource problems

There are a number of components of shared memory that can cause interference among tasks running on separate cores. Three notable culprits are

- *shared caches*, where multiple cores affect a cache's contents thus replacing each other cores' cached data thereby degrading the overall performance for a set of tasks,
- *bandwidth* to the memory controller, which may limit access to main memory and
- *the memory controller*, whose implementation has a great impact on the performance of a shared main memory, both in regards to the physically allocated location and access scheduling routines.

2.2.1 Shared cache and bandwidth

Shared caches are a key factor in memory based interference among cores. Assuming complete utilization of cache, a task bringing in data from main memory will need to evict some already present data, possibly belonging to another task, which may later need to be referenced and thus brought back into memory. Every such eviction increases the miss-rate and consequently bandwidth usage to main memory. Effectively, tasks running in parallel may produce the exact same amount of work but with a greatly increased bandwidth utilization, compared to a sequential execution of the same set of tasks. Due to the order of magnitude in missing a last-level cache, a set of memory intensive tasks might complete in a longer period of time during parallelized execution than in sequential execution.

Listing 1: Task A

```
Task A:
. . .
x := . . .
store(x)
. . .
load(x)
. . .
load (x)
```

Listing 2: Task B

```
Task B:
. . .
y := . . .
store(y)
. . .
load(y)
. . .
load (y)
```

Consider the tasks A and B. Assuming a multi-core system with a one-level shared write-through cache and main memory with access times of 10 and 100 cycles respectively, and the cache allocation function by coincidence hashes both x and y to the same physical location.

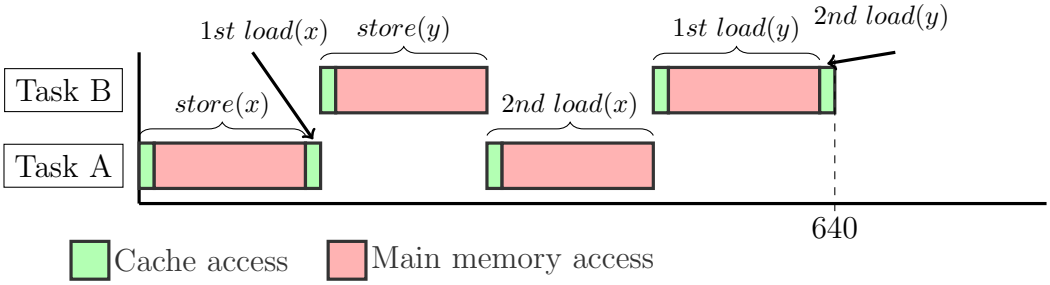

Figure 2: An example of task A and B running in parallel.

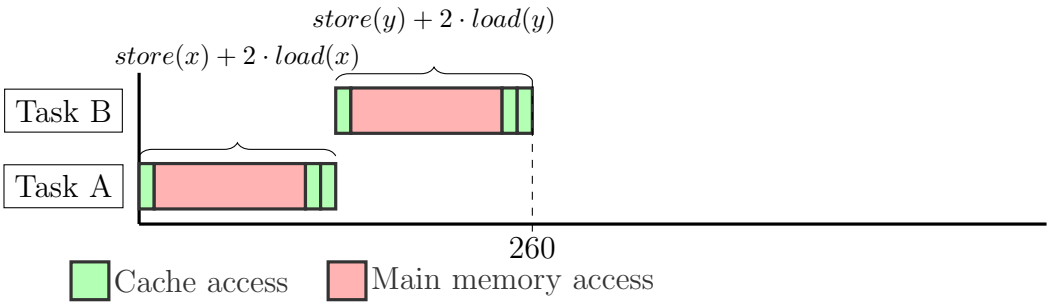

Figure 3: An example of task A and B running sequentially.

The execution may have a run-time as seen in figure 2 when running tasks A and B in parallel, with task B evicting task A's data from cache and thus

significantly increasing the total run-time in comparison to running the two tasks sequentially as seen in figure 3.

2.2.2 Memory controller

DRAM modules employing a open-row policy, keeping the most recently used row open, benefit greatly from physical spatial locality of data due to the lower number of costly precharge operations performed during memory accesses. Memory controllers exploit this in order to maximize bandwidth, by scheduling access requests on open rows with increased priority. In [10], Rixner et al. demonstrate the benefits of reordering the queue of accesses, on bandwidth. Such reordering, however, allows tasks with greater spatial locality to receive a larger share of accesses to main memory. While such a scheme guarantees a higher average throughput, it also introduces greater access delays to tasks with lesser spatial locality. This may have a particularly critical impact on real-time systems with hard deadlines if this interference is not accounted for or otherwise dealt with. A possible method to mitigate such interference is the use of memory access budgeting with time shares and/or memory access count.

2.3 FIASCO.OC

FIASCO.OC [11] is a multi-platform microkernel belonging to the L4 microkernel family, developed for the TUDO:OS system at Dresden University of Technology. Due to the minimalistic nature of microkernels, FIASCO is well suited for both complex systems and small embedded applications providing full preemption making it particularly well suited for real-time applications.

The L4 microkernel was initially developed by Jochen Liedtke in x86 assembly with the primary purpose of performance. This was achieved by among other things only including *absolutely* necessary functionality in the kernel and any other functionality should be provided on top of the kernel as user applications.

In order to increase portability and maintainability many new implementations of the initial L4 were made in higher-level languages such as FIASCO, which is written in C++. This has now come to be called the L4 microkernel family.

2.4 L4 Runtime Environment

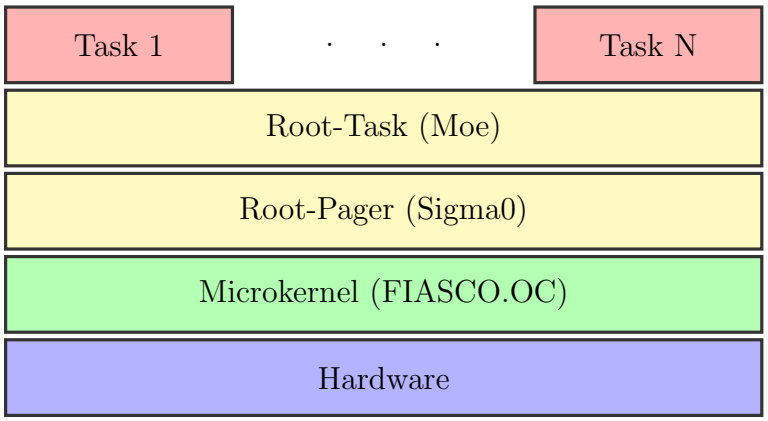

Figure 4: Structure of an L4Re system.

The L4 Runtime Environment (L4Re) [12] provides infrastructure for convenient development of applications to be run on top of FIASCO. While the FIASCO microkernel provides a set of interfaces, L4Re extends on these and provides additional services.

The two key components provided by L4Re are the root pager, Sigma0 and the root task Moe. Sigma0 is primarily used to resolve page faults for Moe and runs, like everything else but the actual microkernel, in unprivileged mode. The root task, Moe, is responsible for bootstrapping the system and provides resource management for tasks running on top of L4Re. See figure 4 for a overview of the hierarchical structure.

Additionally L4Re provides an initialization process, Ned, responsible for coordinating the startup of tasks and communication channels between them.

3 Method

3.1 Deployment

A modified version of Fiasco.OC revision r62 was used during all measurements compiled for ARMv7A and Freescale i.MX6. In order to obtain time measurements, the kernel was modified with instructions allowing user-mode access to a performance counter, used for counting the number of cycles performed by the CPU.

In order to cause a certain level of interference, a media-player was deployed on top of L4Linux co-running with the performed measurements. MPlayer v1.1.1, including a FFmpeg [13] snapshot (version 54.23.100), was used in the role of a media-player, obtained from the MPlayer website [14].

L4Linux version 3.13.0-14-svn46 was run with a modified version of the ARM RAM disk available at the L4Linux website [15]. The modifications include increasing the size of the RAM disk to 75MiB, including the previously mentioned media-player and a media file. The media file contains 1080p video encoded with H264 running at 24.833 fps amounting to 14112.7kb/s and audio encoded with AAC with a data-rate of 123.5kb/s. All above mentioned parts were compiled using gcc 4.7.3-12ubuntu1.

The above mentioned components were then packaged into a uImage placed on an SD-card, used to boot on the Freescale SABRE Lite i.MX6 board using a provided version of U-Boot [16] (2009.08). Interfacing was performed using the available RS-232 serial port.

3.2 Memory layout

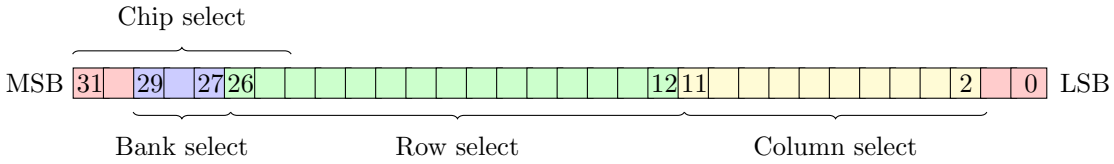

Figure 5: Bank decoding example scheme with bank interleaving *disabled*.

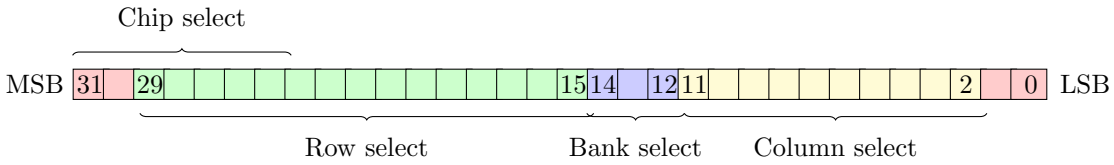

Figure 6: Bank decoding example scheme with bank interleaving *enabled*.

The memory controller on the i.MX6 board implements two modes of address decoding. For both modes the 7 most significant bits of the physical address are used for selecting chips. The difference lays within what bits are used to select banks. In non-bank interleaving mode, bank selection is performed towards the significant bits e.g. assuming x32 DDR with 8 banks and a 15

bit assignment for rows, bits 27 to 29 will be used for bank selection, counted from the least significant bit [17], see figure 5. With bank interleaving mode enabled the bank selection bits are preceded by row selection e.g. assuming the same assignment as previously, bits 12 to 14 will be used for bank selection [18], see figure 6.

3.3 Performed measurements

In order to measure the impact on performance caused by running memory intensive applications, a benchmark suite was used as a reference point. *AutoBench* from EEMBC was used for this purpose. The benchmark was first run without any interference, and then run again with interference caused by a media file running on MPlayer under L4Linux, assigned to a different core. The test was then repeated with data cache disabled.

4 Results

The results seen in table 1 were obtained from AutoBench running with data cache enabled, and display the difference in run-time experienced by the benchmark components when running with interference, compared to running without interference. Multiple samples were collected and with the exception of the inverse fast Fourier transform benchmark, all collected run-time samples without interference were lower than the lowest run-time sample collected with interference.

Intuitively, the average measured slowdown of 0.37%, may seem insignificant, however if the interference is not uniformly distributed and spikes on a real-time system during a hard tasks execution, the results may be of significant effect.

A mitigating factor towards the interference caused may also be memory bank interleaving, allowing a certain level of parallelization leading to minimized interference over time on average. This however might not guarantee a uniform interference, even if assuming fairly uniform memory access requests from the media-player, as the address to bank decoding may lead to certain time frames with a greater level of collisions. These time-frames might also prove to be most critical.

Generally, the *AutoBench* benchmark targets embedded systems and is therefore fairly minimal, while the board has fairly high specifications in relation

to embedded use, especially with regards to cache size. Therefore cache conflicts may have been sparse, explaining the fairly low measured level of interference.

Benchmark	**Slowdown %**
Angle-To-Time Conversion	0.25%
Fast Fourier Transform	0.82%
Finite Impulse Response(FIR)Filter	0.10%
Inverse Fast Fourier Transform	-0.03%
Basic Floating-Point	1.30%
Bit manipulation	0.20%
Cache Buster	0.10%
Response to Remote Request(CAN)	0.14%
iDCT,Inverse Discrete Cosine Transform	0.20%
Low-Pass Filter(IIR)and DSP functions	0.10%
Matrix Math	1.51%
Pointer Chasing	0.27%
Pulse-Width Modulation	0.37%
Road Speed Calculation	0.11%
Table Lookup	0.29%
Tooth-To-Spark	0.19%
Total	**0.37%**

Table 1: AutoBench run-time interference caused by media-player running under L4Linux, running with data-cache enabled.

With data cache disabled, there was, as expected, a significant overall run-time slowdown. Additionally, the overall interference was increased. These results can be seen in table 2.

Two of the benchmark suites components had a decreased run-time with the media player running simultaneously. This may be the result of the benchmarks physical address location, and its mapping to different bank, allowing for more bank-interleaving. Swapping the order of execution of the benchmark components and their starting times, had a noticeable, although minor, effect on run-time. This has however not been included in the tables and the same order of execution was preserved between both the data-cache enabled, and disabled measurements for reasons of consistency. It does, however, suggest that their is a an impact on run-time introduced by varying the order of loading the benchmark components into memory and/or time of execution, which may be a potential source of fault within the obtained results.

Benchmark	**Slowdown (%)**
Angle-To-Time Conversion	8.46%
Fast Fourier Transform	4.96%
Finite Impulse Response(FIR)Filter	-0.44%
Inverse Fast Fourier Transform	3.27%
Basic Floating-Point	5.04%
Bit manipulation	6.38%
Cache Buster	9.67%
Response to Remote Request(CAN)	-0.93%
iDCT,Inverse Discrete Cosine Transform	7.32%
Low-Pass Filter(IIR)and DSP functions	3.32%
Matrix Math	5.94%
Pointer Chasing	0.86%
Pulse-Width Modulation	1.88%
Road Speed Calculation	12.77%
Table Lookup	3.42%
Tooth-To-Spark	1.33%
Total	**4.60%**

Table 2: AutoBench run-time interference caused by media-player running under L4Linux, running with data-cache disabled.

Further complications with interpreting the results come from the memory controller mode default, as described in section 3.2, being unknown and the translations from the virtual address to the physical address also being unknown.

With bank-interleaving mode enabled, the worst case run-time may turn out to be significantly greater than that represented in the test results as a greater level of parallelization is likely to have occurred, assuming some spatial locality in physical address space among the tests. Considering the tests have a fairly low memory space allocation, non-bank-interleaving mode is less likely to be affected as the difference between two addresses has to be far greater in order to switch bank.

Judging by how few test modules presented faster run-times when co-running with the media-player and the low variance between the collected samples in each test module, I believe it is likely that the memory controller was set to non-bank-interleaving mode during the time of the tests. This is however only speculation and further investigation is required.

5 Related work

In the past, several methods have been proposed for relieving the effects of memory contention in systems employing the sharing of memory resources.

One approach bases itself on the partitioning of shared main memory [19,20] by controlling the virtual to physical memory mapping on the operating system (OS) level, thereby mapping task memory to different banks and therefore allowing for parallelized service of memory access requests assuming bank interleaving is present.

Several other articles and proceedings, such as [21–23], explore reducing interference by means of partitioning memory access time, and bounding task response times. This approach is feasible for real-time applications as waiting times can be bounded with a fair amount of ease.

Another explored approach in [24–26], is the budgeting of number of memory accesses, where tasks or cores exceeding a periodically replenished quota, are suspended from accessing the memory. These, however, assume the worst case scenario in regards to memory accesses, thereby limiting the overall throughput for tasks that finish without utilizing their entire assigned budget. In [27], the possibility of reclaiming unutilized memory budget for other tasks is explored.

6 Conclusion

In conclusion there was a noticeable degradational effect on run-time performance when running interference on another core in parallel. The means by which this interference may be dealt with, especially in real-time applications is budgeting.

In future work, the budgeting techniques of [27] could be implemented and benchmarked on the system presented in this thesis. This would be of great interest as it would provide a good comparison of the viability of the different techniques on a single setup.

In case budgeting is to be applied for example on a real-time system, with multimedia applications running as soft tasks and limited memory resources, it may be feasible to investigate the behaviour of memory accesses of different types of multimedia formats, both in terms of locality and access timing. If patterns emerge, such as spikes in access timing, this could be used in order

to efficiently schedule the multimedia applications in order to maximize the interactivity and smoothness of playback to the multimedia application while guaranteeing timing-correctness for hard-tasks. This might however prove difficult due to the vast amount of different media formats and types.

Future improvements may try to clarify the effects of the different modes selected in the memory controller and compare the level of interference caused when using either mode. A major improvement would be to isolate memory accesses to a single bank in order to investigate more reliably what level of interference can be expected and how the budgeting of resources might affect multi-media applications such as media-players.

Additionally, the chosen benchmark suite is specifically designed in order to test performance for applications in the automotive industry and might therefore not be a representative outcome for other applications. This could partially be improved by including more benchmarks and of greater target diversity.

References

[1] S. Naffzinger, "Technology impacts from the new wave of architectures for media-rich workloads," in *VLSI Technology (VLSIT), 2011 Symposium on*, pp. 6–10, 2011.

[2] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, pp. 46–61, Jan. 1973.

[3] N. Rafique, U. Purdue University, W. T. Lim, and M. Thottethodi, "Effective management of dram bandwidth in multicore processors," in *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pp. 245–256, IEEE, 2007.

[4] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, (3001 Leuven, Belgium, Belgium), pp. 741–746, European Design and Automation Association, 2010.

[5] ARM, "Cortex-a series." `http://www.arm.com/products/processors/cortex-a/`. Accessed: 2014-08-29.

[6] ARM, "Cortex-a series." `https://www.element14.com/community/servlet/JiveServlet/downloadBody/51164-102-2-287410/Freescale%20Sabre%20Lite%20User%20Manaul%20V1.3.pdf`. Accessed: 2014-08-29.

[7] EEMBC, "AUTOBENCH an EEMBC benchmark." `http://www.eembc.org/`. Accessed: 2014-07-08.

[8] EEMBC. `http://www.eembc.org/`. Accessed: 2014-07-08.

[9] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*, ch. 7. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[10] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, (New York, NY, USA), pp. 128–138, ACM, 2000.

[11] TU Dresden OS Group, "Fiasco.OC microkernel." `https://os.inf.tu-dresden.de/fiasco/`. Accessed: 2014-07-07.

[12] TU Dresden OS Group, "L4Re Runtime Environment." https://os.inf.tu-dresden.de/L4Re/. Accessed: 2014-07-08.

[13] FFMPEG. https://www.ffmpeg.org/about.html. Accessed: 2014-07-08.

[14] MPlayer. http://mplayerhq.hu/MPlayer/releases/MPlayer-1.1.1.tar.xz. Accessed: 2014-07-08.

[15] TU Dresden OS Group. http://os.inf.tu-dresden.de/download/ramdisk-arm.rd. Accessed: 2014-07-08.

[16] DENX, "Das U-Boot – The Universal Boot Loader." http://www.denx.de/wiki/U-Boot/. Accessed: 2014-07-08.

[17] Freescale, *i.MX 6Dual/6Quad Applications Processor Reference Manual Rev 2*, 6 2014. pp. 3838-3839.

[18] Freescale, *i.MX 6Dual/6Quad Applications Processor Reference Manual Rev 2*, 6 2014. pp. 3839-3840.

[19] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 367–376, ACM, 2012.

[20] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing dram locality and parallelism in shared memory cmp systems," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pp. 1–12, Feb 2012.

[21] S. Goossens, B. Akesson, and K. Goossens, "Conservative open-page policy for mixed time-criticality memory controllers," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, (San Jose, CA, USA), pp. 525–530, EDA Consortium, 2013.

[22] J. Rosn, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pp. 49–60, 2007.

[23] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury, "Bus-aware multicore wcet analysis through tdma offset bounds," in *23rd Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 3 –12, 2011.

[24] M. Behnam, R. Inam, T. Nolte, and M. Sjödin, "Multi-core composability in the face of memory-bus contention," *SIGBED Rev.*, vol. 10, pp. 35–42, Oct. 2013.

[25] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pp. 299–308, 2012.

[26] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS) 2013*, pp. 55–64, 2013.

[27] J. Flodin, K. Lampka, and W. Yi, "Dynamic budgeting for settling dram contention of co-running hard and soft real-time tasks," in *The 9th IEEE International Symposium on Industrial Embedded Systems*, (Pisa, Italy), pp. 151–159, 2014.

Appendix

Below are the instructions for setting up the system used in this thesis.

Get source files

Donwload the zip file from:
`https://drive.google.com/file/d/0B_qe8hQP_xoYWTRfMlU2LUVxbGM/view?usp=sharing`

To get FIASCO and L4RE run:

```
$ svn cat http://svn.tudos.org/repos/oc/tudos/trunk/repomgr |
    perl - init
http://svn.tudos.org/repos/oc/tudos fiasco l4re
   l4linux_requirements
```

Go to the source dir, and make sure the checked out revision is 62:

```
$ svn up -r62
```

Build FIASCO

Goto src/kernel/fiasco dir Replace the file src/kern/arm/cpu-arm.cpp with the one provided in the downloaded zip-file. This enables cycle-counting. Run:

```
$ make BUILDDIR=/path/to/fiasco/build/dir
$ cd /path/to/fiasco/build/dir
```

Add the files globalconfig.h and globalconfig.out to the new builddir Run:

```
$ make
```

Build L4Re

```
$ mkdir -p /path/to/l4re/build/dir/
```

Go to src/l4/ dir Run:

```
$ make O=/path/to/obj/tudos oldconfig
$ make O=/path/to/obj/tudos -j2
```

Copy bm and libc_be_stdin to src/l4/pkg folder run:

```
$ make O=/path/to/l4re/build/dir from src/l4/pkg/ dir
```

Copy modules.list to src/l4/conf/ dir

Copy contents of l4f dir to /path/to/l4re/build/dir/bin/arm_armv7a/l4f

L4Linux

Get the source:

```
$ svn co http://svn.tudos.org/repos/oc/l4linux/trunk l4linux
```

Go to l4linux src dir:

```
$ cd l4linux
```

```
$ make L4ARCH=arm CROSS_COMPILE=arm-linux-
    O=/path/to/l4linux/build/dir
arm_defconfig
```

Then run: (Note set L4 tree build directory to l4 build dir)

```
$ make L4ARCH=arm CROSS_COMPILE=arm-linux- O=/path/to/obj/l4linux
    menuconfig
```

Finally run:

```
$ make L4ARCH=arm CROSS_COMPILE=arm-linux- O=/path/to/obj/l4linux
```

Copy the vmlinuz.arm file to l4/build/dir/bin/arm_armv7a/l4f dir.

Creating a uimage

Go to the L4 build directory Create UIMAGE using:

```
$ make uimage
    MODULE_SEARCH_PATH=/absolute/path/to/fiasco/build/directory
```

Choose the module to be turned into an image.

The bm module contains a benchmark only version. The l4lxh2 module contains both the benchmarks and l4linux.

Images will apear in l4/build/dir/images Use the .uimage files.

Running image on target board

Put the image on the provided SD card (note that only one image seems to be recognised on the SD card so remove any old images).

Connect a RS-232 cable for interfacing to the connector marked "DEBUG".

Start a serial communication with Baud rate 115200 and 8 bits.

In order to load the image use:

```
$ mmc dev 1 && ext2load mmc 1 10800000 ui
```

In order to boot the loaded image use:

```
$ bootm 10800000
```

In order to launch a video in mplayer, use

```
$ ./mplayer <video_file_name>
```

Provided in the ramdisk, is a video with the name 20140521_180957.mp4 which was also used in the thesis.