Distributed Operating System and Scheduling for MPSoC

Ali Ahmadinia
Department of Computer Science, California State University San Marcos

ACM Reference format:
Ali Ahmadinia. 2017. Distributed Operating System and Scheduling for MPSoC. In *Proceedings of High-Performance Parallel and Distributed Computing, Washington D.C., USA, June 2017 (HPDC'17),* 2 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Multiprocessor system on chips (MPSoCs) allow developers to leverage the flexibility of software development, such as multi-threaded and multi-tasked application writing, with the computational power and hardware design flexibility of field programmable gate arrays (FPGAs). However, where developers take advantage of the kernel of an operating system (OS), the large variability in MPSoC manufacturers makes the presence of such an OS an uncertainty, and therefore the development of multi-threaded and multi-tasked application difficult. For dedicated applications, where MPSoCs are readily deployed, tasks and threads can be statically allocated to available cores, with thread and inter-task dependencies resolved before runtime. Threads and tasks can also be scheduled on available cores by running an OS on each available core, allowing cores to service different priority tasks. Executing the same OS on available cores of an MPSoC allows us to conceptualize a distributed OS where no overseeing kernel is available to delegate tasks to the available cores, but tasks and threads can still be scheduled and prioritized as if they were. This concept introduces an interesting area of research, where we can allocate tasks to cores which can then schedule threads according to a predefined scheduling policy such as round-robin. The main question is: How does the scheduling of a distributed OS affect the performance of MPSoC executing data-parallel applications? To address this question, we first discuss how a distributed OS can be implemented on an MPSoC through the use of a *global system tick*. We then present a task and thread assignment methodology for such MPSoCs, and how memory customizations can be used to improve the performance. Previous work such as [1] created a centralized OS for MPSoCs; however we investigate a low-complexity implementation for the distributed unification of cores.

HPDC'17, June 2017, Washington D.C., USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 METHODOLOGY

We assume as input a multi-threaded and multi-tasked implementation of the desired application, in our case the application is the Viola Jones algorithm [3], where task and thread dependencies have been resolved. Consider the case where our data-parallel application consists of 2 main tasks (MTs), each of which can have up to n sub-threads (STs). We implement the thread dependencies into the ordering of main tasks on each processor, instead of imposing priorities on these threads, and relying on preemption. Each MT is multi-threaded, where STs are allocated to cores based on the maximal amount of data each thread might process. Based on the inheritance relationships defined above, we can distribute MTs and their respective STs across the available cores through their threads and schedule them on each cores' respective OS instance. Resolving these intertask dependencies before runtime and allowing them to be inherited by threads that will perform their computation allows threads to be scheduled easily by an OS. We discuss next how the concept of the distributed OS is formulated and realized on an MPSoC. A typical OS schedules threads and tasks, based on a predefined scheduling scheme, to available cores of the system. In this model, the OS sits between the application and the hardware. However, due to the large number of types of MPSoCs that are available, OSs are typically only available on a per core basis, such as the Xilkernel by Xilinx [4]. Therefore, without development of the OS, cores are not unified with respect to other cores. Cores access shared resources through their private cacheable accesses to main memory, and have access to timers to measure execution times. The global timer generates a periodic global system tick which is responsible for making cores context switch to a new thread every tick by using interrupts. Fig. 1 depicts the time line for a dual-core system, with 2 MTs. We can see how each thread of the task switches out at the same time based on the tick of the global timer. We can also see that the length of MTs does not matter in the system, that is the number of ticks required to service the threads of the MT, as the allocation of tasks and threads ensures a worst-case finish time of threads. This ensures a balance of core loads and a deterministic run-time of threads, in terms of global system ticks. Furthermore, we can customize the organization and layout of memory in a MPSoC to simplify the thread and task programming model, as described next.

3 MEMORY ARCHITECTURE

In order to optimize the program memory of the system for reuse and resource consumption, we need to analyze how instructions will be accessed from main memory and cached. For a processor (core) with instruction cache (IC), instructions will be fetched from main memory and then stored in

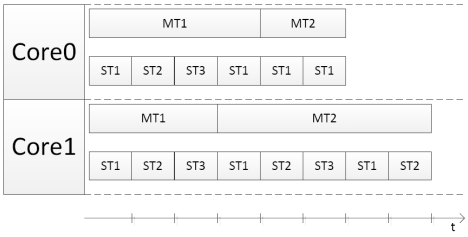
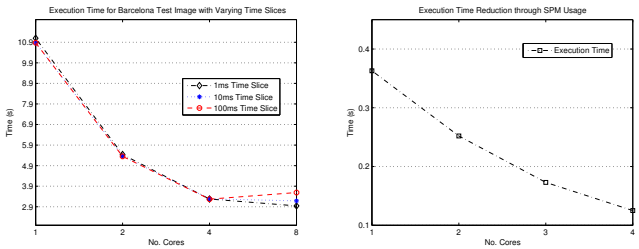


Figure 1: Task switching using the round-robin time slice

IC. If the .text section of the application can fit into the IC, then it would be more efficient to simply store the instructions within the core's cache. This concept applies to all cores of a multi-core system. Furthermore, unnecessary data duplication can exist, as the .text section occupies main memory as well as the caches. We can optimize the .text section by moving the IC resources to local memory, and pre-loading them before runtime with the .text section. This optimization also complements our task/thread allocation methodology, as threads only ever exist on one core: containing thread instruction accesses locally to each core. Three system-level improvements will be exhibited through these optimizations: 1) Main memory traffic will be reduced. 2) Access time to instructions will be reduced through the use of SPMs (Scratch Pad Memory). 3) Data duplication will be reduced. Dynamic data management (DMM) is an important consideration for MPSoC, due to the way tasks are delegated to cores. To compliment our task/thread allocation and scheduling methodology, we make use of static data allocations [2] to allow our distributed OS to access all dynamic data of the application at any point in time. This migration of dynamic to static data allocations complements our task and thread allocation methodology in two ways. First, it enables us to disregard the dynamic data dependencies between tasks and threads and use simple locking mechanisms whenever they are accessed in a critical region. And secondly, the dependencies encoded into each thread ensures allocations are accessed atomically (where required) and in the correct temporal order: thereby removing any race conditions that may arise.

4 EVALUATION

We have implemented up to 8-core MPSoC designs on a ML605 development board from Xilinx [4]. We use the Microblaze soft-core processor to instantiate cores of our MPSoCs. We tested the ability of the distributed OS to perform concurrent context switches with various different time slices. Fig. 2.a illustrates the average execution time of the application with each time slice when tested with the Barcelona test image. We first see that our methodology reduced the execution time with increasing core counts, as is to be expected. The frequent context switches from the 1ms time slice leads to the largest execution times for core counts up to 4. However, for core counts from 4 to 8, we see that the less frequent



(a) (b)

Figure 2: (a) Execution times for different time slices. (b) Impact of SPM on Execution Time

100ms time slice actually leads to an increase in execution time. The reduction in execution time for the smaller time slice with larger core numbers, and the increase in execution time for small core numbers is an interesting observation. We also evaluate the memory customizations of the detection library with 1-4 cores and how effectively they complement our distributed OS. From Fig. 2.b we can see that the the incorporation of SPMs to accommodate the detection library reduces the execution time of each custom memory design. We can attribute the leveling off of the improvements to the contention that can be experienced in centralized SPMs. With more cores accessing the centralized resource, more arbitration and contention may be experienced leading to increased wait times for the data.

5 CONCLUSIONS

We have presented a distributed OS for MPSoCs, where commercial OSs that only execute on one core are used. We described how the application executing on the MPSoC can be decomposed into threads and tasks. We also optimized memory for this distributed OS. Our testing with MPSoCs with up to 8 cores revealed that there is a trade-off between the number of cores instantiated in the system and the frequency of context switches. Through analysis of the cross-over point of core counts and time slice periods from execution time graphs, we found that low core counts favored a large time slice period of 100ms, whereas large core counts favored a small period of 1ms. Our future work will focus on providing theoretical analysis of these observations, the implication of the time slice on thread computation complexity.

REFERENCES

[1] E. Matthews, L. Shannon, and A. Fedorova. 2012. Polyblaze: From one to many bringing the microblaze into the multicore era with Linux SMP support. In *Field Programmable Logic and Applications (FPL), International Conference on*. 224–230.
[2] R. Panda, F. Catthoor, D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. Kjeldsberg. 2001. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Elec. Syst.* 6, 2 (2001), 149–206.
[3] P. Viola and M. Jones. 2001. Rapid object detection using a boosted cascade of simple features. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition*. I–511 – I–518.
[4] Xilinx. 2017. http://www.xilinx.com. (2017).

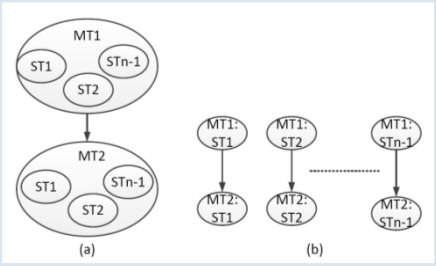
Distributed Operating System and Scheduling for MPSoC

Ali Ahmadinia
Department of Computer Science and Information Systems
California State University San Marcos
Email: aahmadinia@csusm.edu

Abstract:

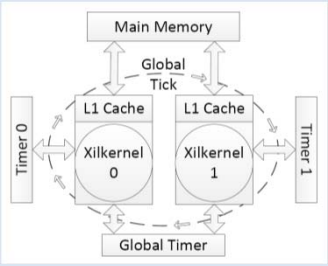
- Scheduling and managing tasks and threads for multiprocessor system on chips (MPSoCs) has been typically achieved with the help of centralized operating systems (OSs).
- In this work, we investigate the scheduling and allocation of threads and tasks on MPSoCs through a distributed OS, where cores run their own separate instance of an OS, and OS instances themselves are coordinated through a global system tick.
- Through our collaborative task/thread assignment and memory customization methodology, we find that the time slice used for context switches affects the performance of MPSoCs, based on the number of cores present.
- Our tests with a object detection application reveal that a small slice benefits larger core counts, whereas a large slice benefits smaller core counts.

Task and Thread Allocations:



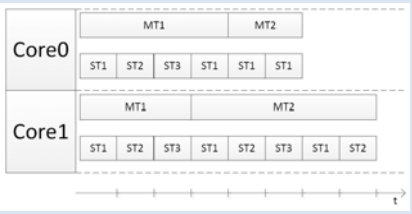
Task dependencies: (a) Main tasks (MTs) consist of sub-threads (STs) (b) STs inherit the dependencies of their parent tasks

Distributed Operating System:



Example of dual-core MPSoC where cores execute their own instance of the Xilkernel, under synchronicity of the global tick generated by the global timer

Thread Scheduling:



Example of task switching using the Round-Robin time slice. Each MT can consist of a variable number of STs, which can have a variable runtime as a result of input data, but it does not introduce anomalies due to the precedence of the task dependencies.

Memory Architecture:

1. Distributed Program Memory

- If the .text section of the application can fit into the instruction cache (IC), then it would be more efficient to simply store the instructions within the core's cache
- Optimize the .text section by moving the IC resources to local memory, and preloading them before runtime with the .text section
- Complements our task/thread allocation methodology, as threads only ever exist on one core: containing thread instruction accesses locally to each core
- Three system-level optimizations will be exhibited through these:
  1) Main memory traffic will be reduced.
  2) Access time to instructions will be reduced through the use of SPMs.
  3) Data duplication will be reduced.

2. Dynamic Data Accommodation

We make use of static data allocations to allow our distributed OS to access all dynamic data of the application at any point in time. This migration of dynamic to static data allocations complements our task and thread allocation methodology in two ways:

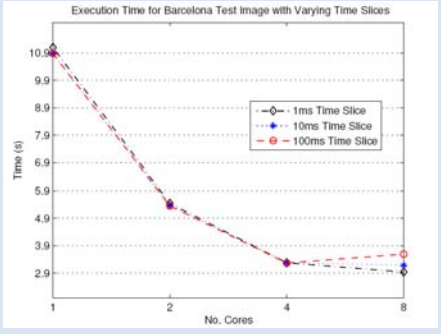
1. Enables us to disregard the dynamic data dependencies between tasks and threads and use simple locking mechanisms whenever they are accessed in a critical region.
2. The dependencies encoded into each thread ensures allocations are accessed atomically (where required) and in the correct temporal order: thereby removing any race conditions that may arise.

Evaluation:

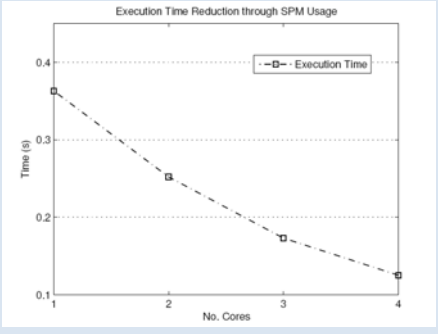
- We have implemented up to 8-core MPSoC designs on a ML605 development
- board from Xilinx with the XC6VLX240T (-1) FPGA.
- We use the Microblaze soft-core processor to instantiate cores of our MPSoCs with 32KB DC and local memory, with IC contained within local memory.
- The object detection library is stored in three 64KB SPMs, whose address space is unified to create a contiguous block of SPM resources accessible by all cores
- Execution times are obtained via separate onboard timers, and the global system tick is varied through the board support package (BSP) of each Microblaze core.

Attribute	Description
Microblaze	32KB DC and local memory, IC contained with local memory
SPM	192KB contiguous address space
Main Memory	256MB accessed over AXI4 interface
Global Syste Tick	Generated via global system timer, varied via BSP, and used to schedule threads based on round-robin Polling
Timers	One timer per core, accessed over AXI-lite interface
Mutex	Core Used for atomic resource accesses, accessed over AXI-lite interface
Test images	Barcelona, Faces, and Die-Hard with 15,29, and1 face(s) each respectively

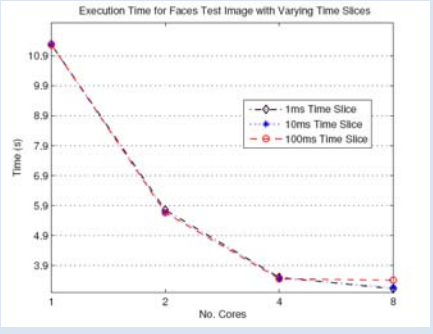
Thread Scheduling Interval



Memory Customizations



Trade-off



Conclusion:

- We have created a distributed OS for MPSoCs, where commercial OSs that only execute on one core are used, where threads are scheduled according to the time slice of the round-robin polling method through a global system tick.
- There is a trade-off between the number of cores instantiated in the system and the frequency of context switches.
- Through analysis of the cross-over point of core counts and time slice periods from execution time graphs, we found that low core counts favored a large time slice period of 100ms, whereas large core counts favored a small period of 1ms.
- Our future work will focus on providing theoretical analysis of these observations, the implication of the time slice on thread computation complexity, and also the testing of other scheduling schemes.