Using JOP to build a chip multiprocessor JVM for embedded realtime systems

José Pablo A. Andreotti.
Dpto. de Arquitectura de Computadoras, Universidad Nacional de Córdoba.
Córdoba, 5000, Argentina.
albertoandreotti@gmail.com

Abstract

Programming embedded devices has been historically a difficult task, involving the learning of the features of a specific device such as the assembly language of a processor. This led to the use of Java in embedded system as a way to improve the development process in such systems.
The challenge of using Java in embedded systems was targeted in part, by the research effort of the JOP (Java Optimized Processor) project carried out by Martin Schöeberl [1], at TU Vienna. JOP is a small processor specifically designed for the execution of real-time Java programs.
This paper discusses the possibilities and the challenges that arise when I built a CMP (chip multiprocessor) JVM (Java Virtual Machine) based on JOP as part of my thesis to obtain my degree in computer engineering. It describes which hardware modules were necessary to be added in order to achieve shared memory access and a proper synchronization of the processors. Then, the problem of scheduling multiple threads on multiple processors and meeting the requirements imposed by the JMM (Java Memory Model) is discussed. All the proposed solutions use a simplistic approach and serve as a baseline for further research.
Finally, a brief description of an implementation of the system in a FPGA (field programmable gate array) is given.

Keywords: JVM, JOP, CMP, FPGA.

1. Introduction

Our intention is to explore the design space for a multiprocessor system that meets the following characteristics:

a) Support for a scalable multiprocessor embedded JVM.
b) Support for multiple threads running on top of each processor.
c) The gain in performance is achieved with little impact on the programmer.
d) The sintaxis and semantics of the Java language remain unchanged.

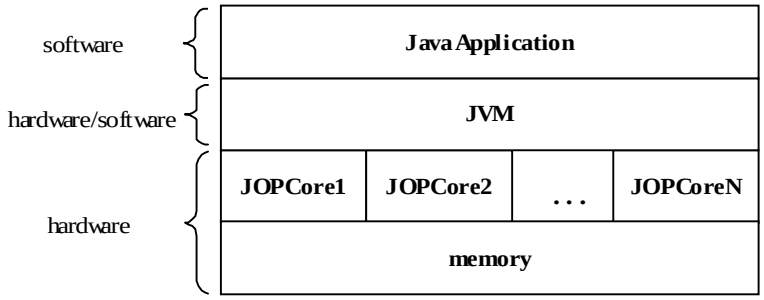

Figure 1.1 Organization of a system to meet the above mentioned requirements. The Java application lies on a JVM which is implemented in both hardware and software. To provide its functionalities the JVM relies on several processors that access a shared memory system.

A layered diagram showing a configuration for the desired system can be seen in figure 1.1.
In order to obtain a system like this the following problems need to be addressed:

a) Provide a way for the processors to share a common memory system.
b) Find a way to make communication available among the processors, so they can cooperate in performing tasks such as initialization and synchronization.
c) Provide a scheduler capable of assigning threads to several processors for their execution.
d) Verify that the system does not violate the restrictions imposed by the Java Memory Model.

The rest of this paper explains how to tackle these four topics.

2. Provide a way for the processors to share a common memory system

In order to implement a shared memory architecture, we need to provide shared access to a single memory interface. In JOP, the interconnection between the processor, the memory interface and the I/O devices is achieved using a standard interface called SimpCon [2].
SimpCon provides a way of accessing different kind of devices in a seamless manner. The SimpCon standard makes a division between slave and master devices. It also contemplates the possibility of multiple access to a single slave device through an arbiter device.
The arbiter performs two basic tasks:

1. Performs transactions on the shared memory system on behalf of the masters(i.e., the JOP cores).
2. Implements the resource sharing policy.

The resultant memory architecture is depicted in figure 2.1.

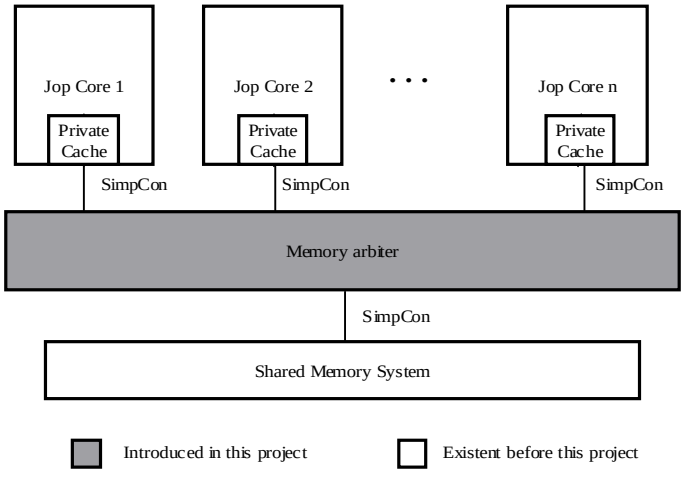

Figure 2.1 Resulting memory architecture after introducing the memory arbiter. The JOP cores are master devices and the shared memory interface is a slave device.

In this configuration, the memory arbiter acts as a master device to the memory interface and as a slave to the JOP cores. It reproduces the transactions started by the masters on the memory interface. The arbiter is transparent to the JOP cores, the only effect noticed by the cores is an eventual increase in memory latency cycles due to the time sharing mechanism.

The resource sharing policy implemented in this project is the simple round robbin. Further research might include a comparison of performance achieved with different resource sharing policies against different benchmark applications.

3. Find a way to make communication available among the processors, so they can cooperate in performing tasks such as initialization and synchronization.

To achieve proper initialization of the CMP JVM and to give support to Java monitors we need a means of communication among the processors. The proposed solution consists in introducing an extra I/O device which implements a common shared control bus. The resulting configuration is shown in figure 3.1.

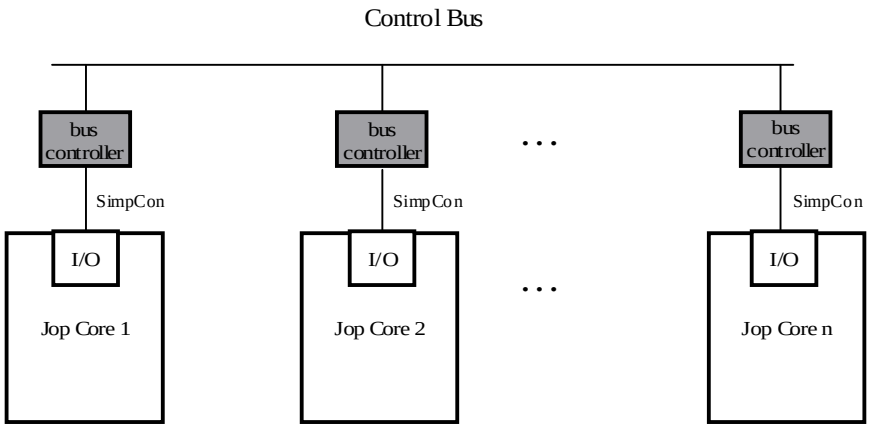

Figure 3.1 The control bus. The bus controller I/O device is used to provide access to a common control bus shared among all the procesors.

The control bus provides two functions:

Initialization: To properly initialize internal structures of the JVM such as internal memory addresses and the GC(Garbage Collector) we need a means of communication between processors. The alternative chosen in this project was to let one processor be the Master[1] and perform all the initializations. When the initialization is done, the master processor signals the other slave processors using a dedicated signal on the control bus. A sequence diagram showing the whole process is shown in figure 3.2.

Synchronization: Synchronization among threads in Java is performed through monitors. As stated in [3] to achieve mutual exclusion in O(1) time in a multiprocessor environment, we need to perform some kind of atomic access to the memory subsystem.
In order to provide such an atomic access we introduced memory locks which are implemented through the control bus as depicted in figure 3.3.

[1] The word “Master” here is no used in the context of the SimpCon standard.

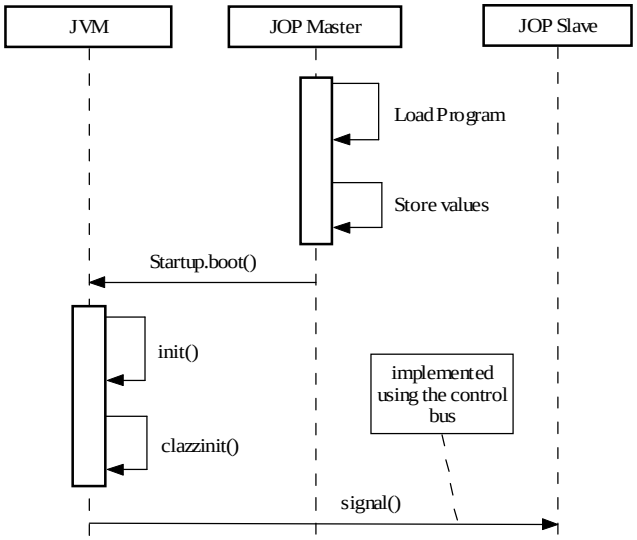

Figure 3.2 Initialization of the system. The master processor performs initializations such as loading the program into main memory, storing values, initializing the GC, etc. When all initialization is done, the master signals the other processor(s).

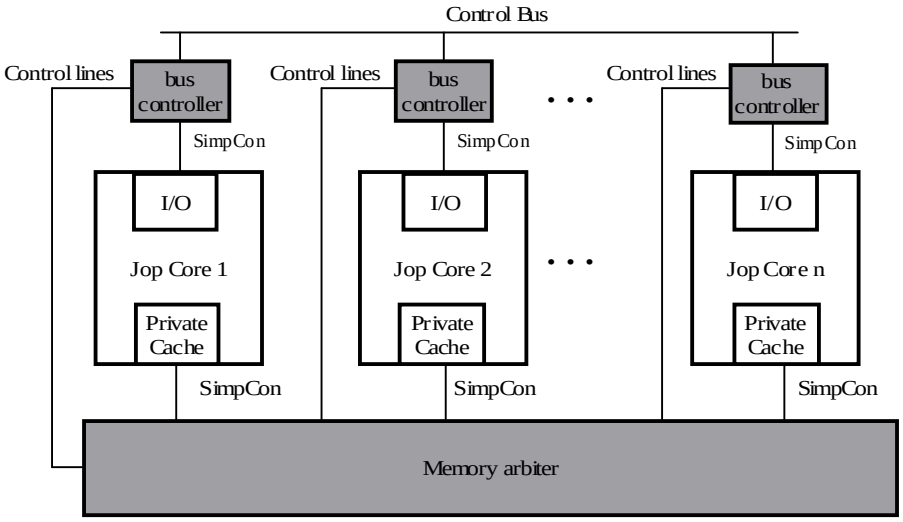

Figura 3.3 Interconnection of the bus controller to the Memory Arbiter. The lines labelled "Control lines" are used to negotiate atomic access to memory through a simple protocol. The responsible of granting atomic access to memory is the memory arbiter.

Using memory locks, the time for a thread to enter its critical section is constant. We only use memory locks inside the code of the JVM. We use them both to impose and order on the accesses to certain internal structures of the JVM and in the implementation of Java monitors.
For example, we have the following pseudocode[1] for the implementation of *monitorenter* and *monitorexit*[2],

monitorenter:

1. Request atomic memory access.
2. Read the owner_id field (identity of the owner of the object).

monitorexit:

1. Load the cnt field.
2. Decrease the cnt field.
3. Store it back to memory.

[1] Bytecodes can be implemented in hardware, in Java or in the microcode native to JOP.
[2] Monitorenter and monitorexit are the bytecodes used to implement synchronized blocks in Java.

3. If the owner is that thread, release memory and jump to 6.
4. If the owner is other thread, release memory and jump back to 1.
5. If the object is not owned by any thread, put this thread id as the new owner, release memory.
6. Increase cnt (a counter which represents the number of times the monitor was entered)
7. Fetch the next bytecode.

4. If cnt=0, release the monitor. This is accomplished by setting owner_id to zero.
5. Fetch the next bytecode.

Further research might contemplate the relative performance of different algorithms and the potential advantage of using lock avoidance mechanisms like the ones described in [4].

4. Provide a scheduler capable of assigning threads to several processors for their execution.

4.1 Introduction

The problem of handling multiple flows of instructions in Java is contemplated at the language level. As we can find in The Java™ Language Specification *Third Edition* [5], Java threads are the means by which Java give support to concurrent programs. The specification also makes no difference on whether Java threads are implemented on one or several multiprogrammed processors.
A task switch in a real-time system consists of two parts: *scheduling* and *dispatching*. Scheduling involves the selection of the next task to be executed and dispatching involves the actual context switch of the processor. So, given a set of threads, the design space is determined by the decision of how to distribute these tasks among the processors.

4.2 Single centralized Scheduler vs several Schedulers

As we stated above for each thread we can:

1. Let just one processor perform its scheduling and dispatching (i.e., “tie” the thread to a processor).
2. Let any processor perform the scheduling and dispatching of a thread.

The second option above is an attractive alternative because it allows the inclusion of interesting features in the scheduler such as allowing one thread to switch among processors. Implementing a centralized scheduler involves some modifications in the way JOP handles interrupts. That’s because the JOP scheduler is very dependent on the interrupt subsystem of each processor. So, in order to implement a centralized scheduler we need a centralized interrupt subsystem shared among all the processors.
On the other hand, the first option is simpler and involves no modification of the scheduler or the interrupt subsystem. This is the one that was implemented for that project. We spend the rest of this section describing how this option was implemented.
The overall architecture for a dualcore JVM is depicted in figure 4.1. In this configuration, the user

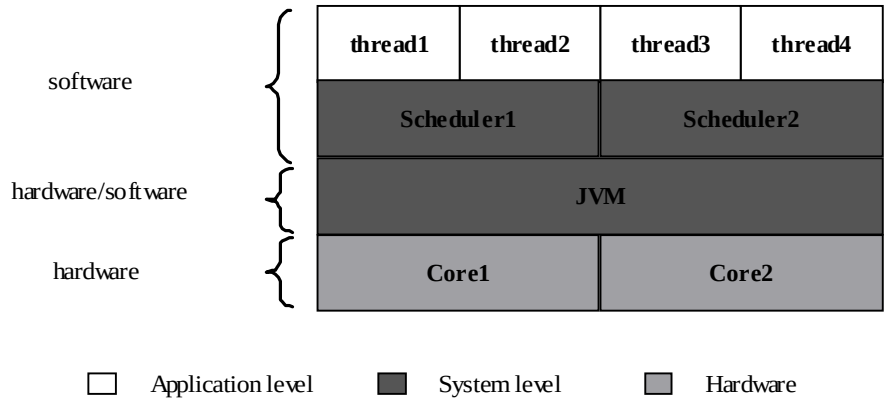

Figure 4.1 Overall architecture of a dual core JVM with two independent schedulers. Each application is written as a set of threads. Each thread is then "tied" to a specific scheduler which runs on one processor. JVM code is accessed by scheduler 1 and scheduler 2 and is thus executed on both cores.

chooses the processor each thread will run on at the time of the creation of the thread objects. This object creation process is shown on figure 4.2.

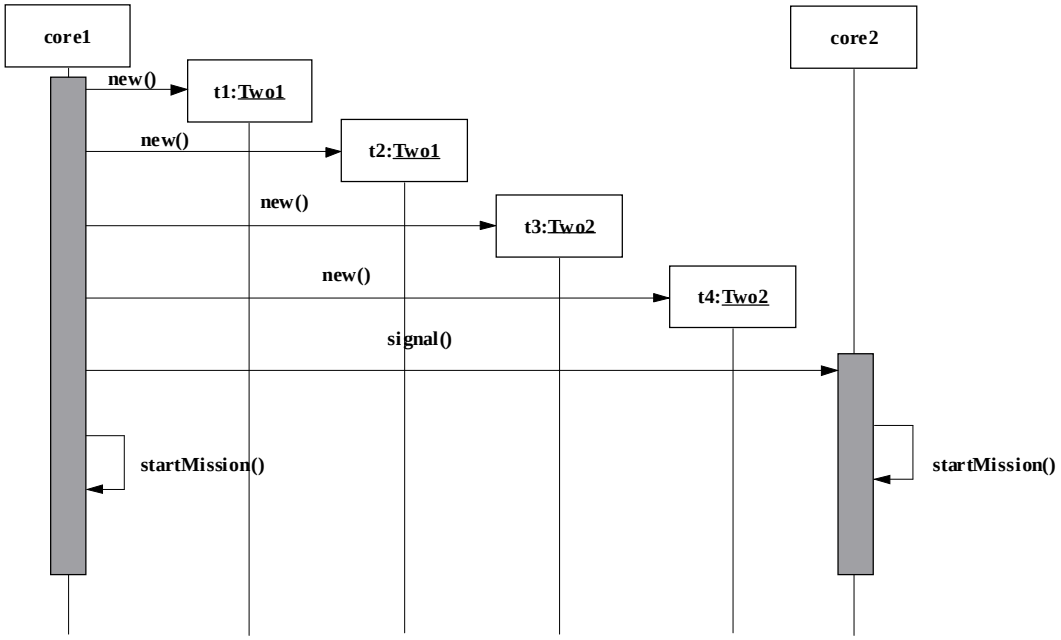

Figure 4.2 Creation of thread objects. Objects of the classes Two1 and Two2 represent threads that will be scheduled on the first and second cores respectively. In this application Core 1 creates the objects and then "signals" the other core. Core 2 starts scheduling its threads with the call to startMission().

An obvious disadvantage of this approach is that the user must be aware of the existence of different processors. On the other hand, it might result attractive in some real-time applications where it is necessary to ensure the availability of certain resources for a specific task. In such a case we just need to "tie" the thread representing the task to the processor that has access to the resources. This is accomplished by creating an appropriate thread object for that task.

5. Verify that the system does not violate the restrictions imposed by the Java Memory Model

As stated in [6], the access to memory in a multiprocessor system is ordered by a shared memory consistency model. The JMM is an abstract model that defines the allowed behaviours of a multithreaded program.
In a regular system, we have two memory models: the memory model of the underlying architecture and the JMM. This is shown is figure 5.1. The JMM defines which are the valid transformations when a compiler produces bytecode, when the JVM produces native code and when the hardware applies optimizations on the native code.

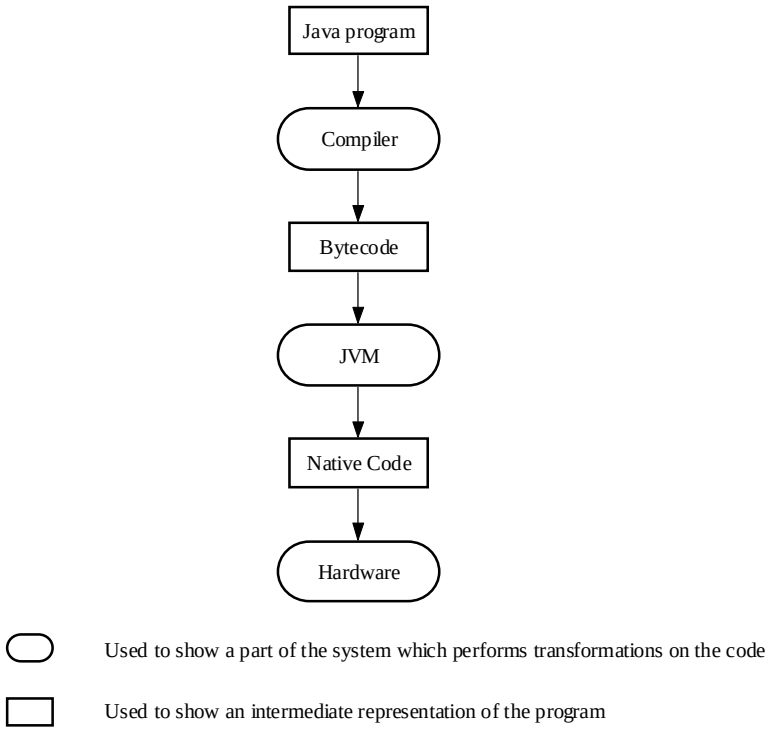

Figura 5.1 Different transformations and intermediate representations for a Java program. In a typical Java system a program is first compiled to a standard bytecode representation, then the JVM generates object code native to the underlying processor. Finally the hardware might perform some transformations on the resulting instruction flow such as instruction scheduling.

Many restrictions of the JMM arise when running a multithreaded program on processors that perform *instruction reordering.* Instruction reordering is a technique used to increase the instruction throughput in a processor pipeline. The idea is to rearrange instructions to get rid of data dependences among them and reduce the number of stalls in the processor pipeline. These data dependences are known as *data hazards*[1].
As demonstrated in chapter 5 of JOP: A Java Optimized Processor for Embedded Real-Time Systems, [1] data hazards are not present in the processor pipeline of JOP. So, when building a CMP JVM using JOP it is not necessary to consider JMM issues related with instruction reordering.
Another issue commonly found in multiprocessors is *cache coherence*[2]. Due to the memory hierarchy in JOP, no problems arise related to cache coherence when interconnecting several

[1] For an in depth study of data hazards, see Patterson & Henessy, Computer Architecture a quantitative approach, [7].
[2] See Parallel Computer Architecture, David E. Culler and Jaswinder Pal Singh, [8].

processors together. In JOP, only data local to a thread is cached[1]. This data is never shared among threads, so no coherence problems are possible . Class variables are stored in main memory. At any moment only a single copy of these variables exists. As there is no data duplication data coherence problems are not possible.

6. Description of an implementation of the system in a FPGA

Both JOP and all the hardware devices introduced in this project are implemented using the VHDL(VHSIC hardware description language)hardware description language. Devices described in VHDL can be implemented in a FPGA. In this section a description of an implementation of the system in a FPGA is provided.

The current implementation consists of a dual core JVM in an FPGA provided by Altera. Devices found in a typical JOP configuration such as the serial interface and external I/O ports were distributed among the processors. Each processor has its own timer and bus controller modules.
A resource utilization comparison of the system against a single core JVM is provided in table 6.1.

	Logic Cells	Memory	Utilization
Single Core JVM	2661	35840 bits	100%
Core	1094		41.11%
Extension	205		7.70%
Scio	316		11.87%
Mem	968		36.37%
Rest of the system	78		2.93%
Dual Core JVM	5409	71680 bits	100 %
Memory arbiter	420		7.76 %
Bus controller	15		0.27 %
Rest of the system	4974		91.97 %

Table 6.1 FGPA utilization for both the single core and dual core versions of the JVM using JOP. The single core version has a 4Kb cache while the dual core version has a 2Kb cache for each core

.

We can see that the device utilization for the bus controller and memory arbiter is 8.3%. So that, we can conclude that little amount of resources are necessary to perform the interconnection of a dual core JOP based JVM.

References

[1] Martin Schöberl. JOP: A Java Optimized Processor for Embedded Real-Time Systems.
[2] Martin Schöberl. SimpCon – a Simple SoC Interconnect.
[3] Michael L. Scott. Programming Language Pragmatics.
[4] Kiyokuni Kawachiya, Akira Koseki, Tamiya Onodera. Lock Reservation: Java locks can mostly do without atomic operations.
[5] Tim Lindholm, Frank Yellin. The JavaTM Virtual Machine Specification 2nd Edition.
[6] Sarita V. Adve, Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial.

[1] The local data cache in JOP is called the *stack cache*, see [5].

[7] John Hennessy and David Patterson. Computer Architecture: A quantitative approach 3rd edition.
[8] David E. Culler y Jaswinder Pal Singh, Parallel Computer Architecture.