Real Time Flow Control for an Advanced Java Co-Processor

Tero Säntti and Juha Plosila
{teansa | juplos}@utu.fi
Communication Systems Laboratory
Department of Information Technology
University of Turku 20014 Turku Finland

***Abstract*— This paper presents a novel method for real time flow control in Java execution. The approach is to use a co-processor for execution of Java bytecode, enhanced with timers. The co-processor is designed using asynchronous techniques to provide low power usage with reasonable performance. The co-processor can be used in a single CPU and single co-processor environment or in a network of multiple CPUs and co-processors. The co-processor does not need to know what kind of environment it is placed in, as all communication goes through an interface unit designed especially for that environment. This modularity of the design makes the co-processor more reusable and allows system level scalability. This work is a part of a project focusing on design of an advanced Java co-processor for Java intensive SoC applications.**

I. Introduction

Java is very popular and portable, as it is a write-once run-any-where language. This enables coders to develop portable software for any platform. Java code is first compiled into bytecode, which is then run on a Java Virtual Machine (hereafter JVM). The JVM acts as an interpreter from bytecode to native microcode, or more recently uses just in time compilation (JIT) to affect the same result a bit faster at the cost of memory. This software only approach is quite inefficient in terms of power consumption and execution time. These problems rise from the fact that executing one Java instruction requires several native instructions. Another source for inefficiency is the cache usage. As the JVM is the only part of software running natively, it occupies the instruction cache, whereas the Java bytecode is treated as data for the JVM, hence being located in the data cache. Also the actual data processed by the Java code is assigned to the data cache. This clearly causes more memory accesses missing the cache. When the execution of the bytecode is performed on a hardware co-processor this is avoided and the overall amount of memory accesses is reduced.

This work is a part of the REALJava [2] project, which aims to design a Java co-processor that is easily integrated to various systems. We have chosen to use asynchronous techniques in this project because then we can achieve good performance with reasonable power consumption and very easy integration with existing systems, since no clock limitations need to be considered. Asynchronous self-timed circuit technology [6], where timing is based on local handshakes between circuit blocks instead of a global clock signal, provides a promising platform for obtaining a highly modular low-power and low-noise Java accelerator implementation.

Overview of the paper We proceed as follows. In Section 2 we shortly describe the structure of any JVM, and show how the proposed co-processor fits into the specifications. Section 3 describes the timers and other real time devices. In Section 4 the invocation methods for the timers are presented. As an example we show how to use the timers to create a chess clock in Section 5. Finally in Section 6 we draw some conclusions and describe the future efforts related to the REALJava co-processor.

II. Generic Java Virtual Machine Structure

In the Java Virtual Machine Specification [4], Second Edition the structure and behavior of all JVM's is specified at a quite abstract level. This specification can be met using several techniques. The usual solutions are software only, including some performance enhancing features, such as JIT (Just In Time Compilation). We have chosen to use a HW/SW combination [2] in order to maximize the hardware support and minimize the power consumption.

The HW portion (highlighted in Figure 1) handles most of the actual Java bytecode execution, whereas the SW portion takes care of memory management, class loading and native method calling. This partitioning gives the possibility to use the co-processor with any type of host CPU(s) and operating systems, as all of the platform dependent properties are implemented in software and (most of) the common bytecode execution is done in hardware. We are also investigating possible benefits of adding hardware support for garbage collection.

Because Java supports multithreading at language level, it makes sense to integrate several co-processors as a SoC. This gives an ideal solution for complex systems running several Java threads and possibly some native code at the same time. This approach brings forth true multithreading and thus improves performance. Also large systems possibly contain several software subsystems, such as internet protocols, user interface controllers and so on, these can easily be coded in Java, and since they all are executed in parallel the user experience is enhanced. The multithreading also improves the predictability of the real time performance, as the threads do not get any wait states and the caches and stacks for each

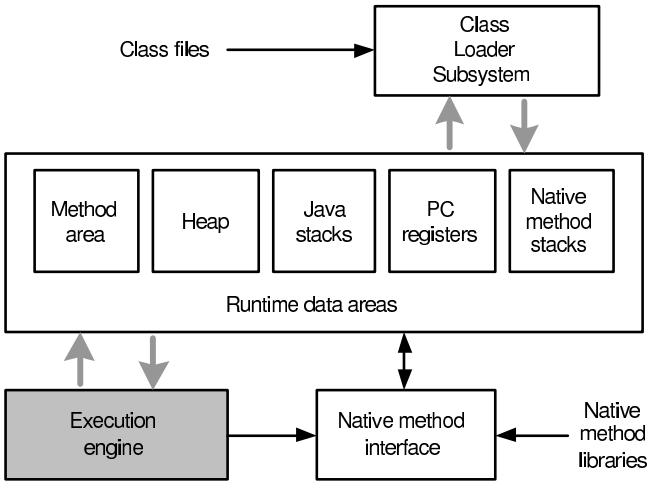

Fig. 1. Internal architecture of the JVM

thread are kept intact inside their respective JPUs. Also the addition of timers gives the programmer new methods for controlling timing of execution with real time events.

The system architecture can be chosen to be a network of any kind or bus based, as suitable for other components in the system. The structure of the underlying network or bus is rather irrelevant, as long as the lower level provides two properties: 1) the datagrams must arrive in their destination in the same order that they were sent, and 2) the datagrams arriving from two different sources to a same destination must be identifiable. The first property can be be achieved with a lower level network protocol, like ATM adaptation layer (AAL) for internet, or by the physical structure of bus. The example we use here is a pipelined bus structure which guarantees the order of the datagrams by structure. The second property seems quite natural, and should be present in all solutions. For demonstration purposes we have chosen to use the pipelined bus [3], since it provides a good platform for multiple processing units accessing the bus simultaneously. The structure is shown in Figure 2. A simple yet efficient protocol for this case is given in [7].

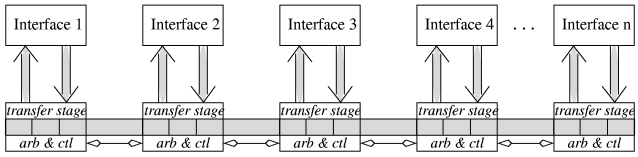

Fig. 2. Detailed view of the pipelined bus with the interface units.

III. Timer Modules

In order to take control of real time events during execution of Java bytecode we propose to add hardware timers to the co-processor module. These timers are to be programmable, with prescaler functions. As our co-processor it self is asynchronous, these timers need an external clock supply to be able to measure real time accurately. The asynchronity of the co-processor also forces us to use the timers as asynchronous counters. It is possible to use them as synchronous counters, but that requires use of external clock as a reference. We have chosen to include four timer modules and four external timing modules. The timers and external modules are connected to a main controller, that resumes the pipeline operation after receiving an event from any of the sources. The controller allows the sources to masked out, so that the software coder can exclude some event sources, when desired. These timers can be used for a variety of purposes, such as real time clocks, communications (baud clock for serial transmissions), time interval based control and simple pulse width modulation. A simplified view of the timers and their connections to the pipeline is shown in Figure 3. Please note that the pipeline differs from the "normal" 5 stage approach used for example in DLX [1]. This is due to the stack based operation of bytecode. All of the data manipulating instructions read the data from the stack and write it back to the stack. There is no need for a write back stage, as it would always cause the pipeline to stall. The pipeline architecture is covered in more detail in [8], where we have also shown other methods to speed up execution and reduce power consumption. The structure of the fold & decode unit and the rationale behind the idea of instruction folding can be found in [9].

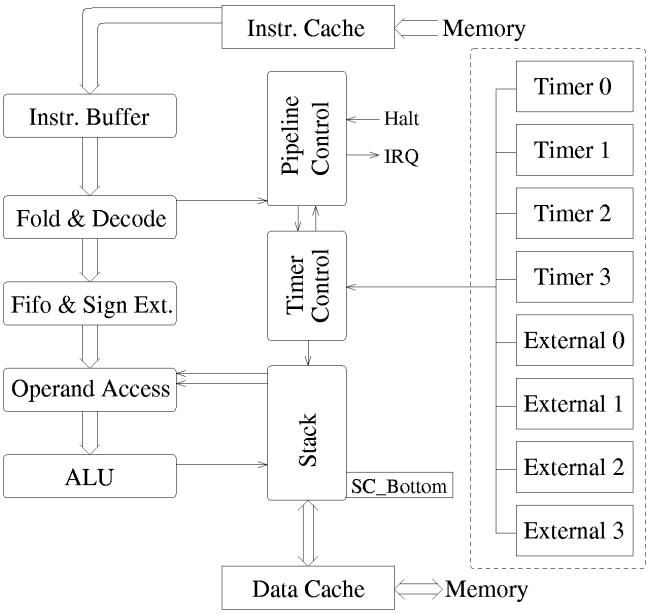

Fig. 3. Internal architecture of the JPU with timers

The timer modules are composed of a 32-bit counter, a prescaler and either a overflow detection unit or a comparator and a period register. In the first configuration, shown in Figure 4 the 32-bit counter can be accessed in read or write mode, to provide fine grain control of the time interval. The counter steps up by one point per every rising edge of the incoming clock signal. As the counter steps over the maximum value and generates an overflow, the module sends an event to the controller. The prescaler is provides longer units of time. The

default value after reset is zero, but it can be selected as 1, 2, 4, 8, 16, 32 or 64. If the prescaler is assigned the value 0 then the counter stops responding to incoming clock events, thus consuming only leakage current. Note that the incoming clock signal does not have to be periodic, these timers can be also used as pulse counters.

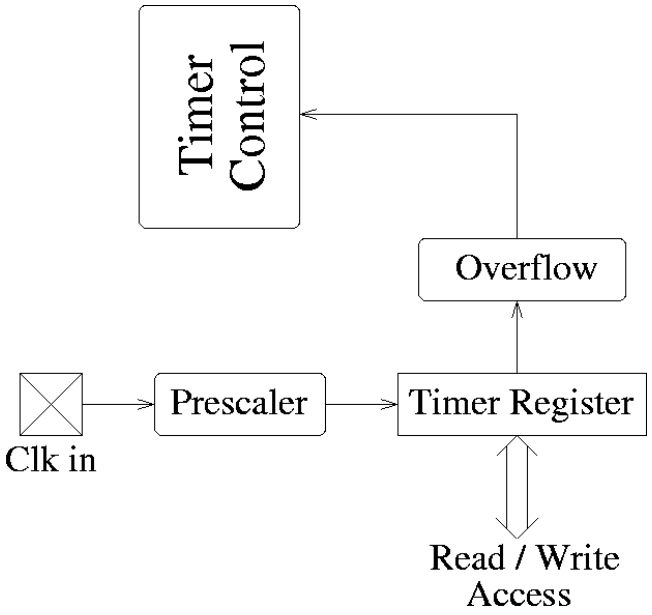

Fig. 4. The structure of a timer with simple overflow detection

The other option is to use a period register, as shown in Figure 5. With this scheme the programmer can set the number of clock ticks in the period register. When this number is reached, the unit resets the timer register to start a new cycle and sends an event to the controller. This way the programmer needs to set the period only once, and (s)he always gets exactly the same period. The prescaler works exactly as in the simpler version. If the timer control has received an event, as is still handling it, the new one will remain pending until the controller is free. The timer whose event is pending, will continue counting normally, thus the period will remain the same. It is up to the programmer to make sure, that the event handling does not take too much time, if accurate real time performance is desired.

The external timing sources have some configuration options as well. They can be configured to send an event for the rising or the falling edges of the incoming signal or for both, detecting any change on external timing source. If an event to trigger a new event arrives on the same external timing source before the previous one is handled, it is ignored. Events on the other external sources and the timers are held pending by the timer controller during the handling of any event. This way we do not lose events, even if we are already responding to one event.

IV. Invoking Timer Functions

Since Java gives a lot of freedom in implementation of the JVM, we simply add the required classes to our JVM. These

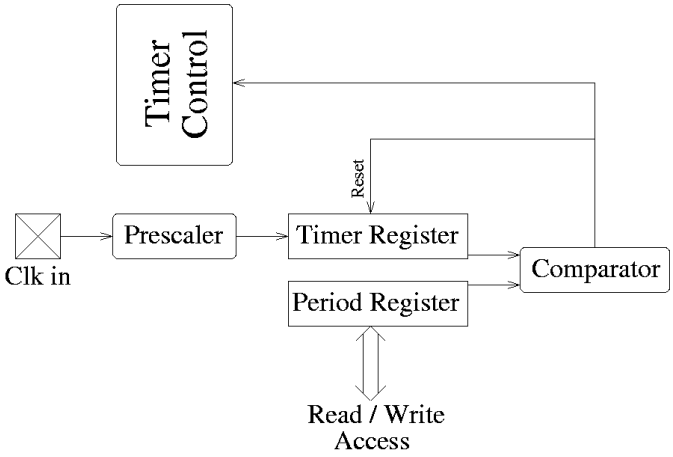

Fig. 5. The structure of a timer with a period register

new methods can be invoked as any other Java methods, and the coder does not need to know the internal functionality of the methods. Note that we will use the "xx()" notation to refer to a method call in Java language used by a programmer and xx to refer to a bytecode instruction performing the functionality. Similar methods can be added to software only JVMs to provide compatibility, but they would not achieve the same accuracy, because of the unpredictable real time performance of software only Java execution.

When a timer method is invoked, the SW portion of the JVM loads the class containing the method, and performs verification and other chores, such as initializing a new stack frame. After that the actual bytecode section is executed. Our JVM implementation notices calls to the timer class, and instead of the normal procedure performs required manipulations on the co-processors timer registers. As an example the coder could call a method to halt execution, until timer0 has overflowed. He would do this by calling first "set_timer_mask(1)" to select timer0 as the only active timer module and then "set_timer_prescaler(1,1)" to start the timer0 with prescaler value '1'. Finally a call to "timer_halt()" would keep the co-processor idle until the event occurs. The JVM would perform these task as follows. First, when encountering the call to "set_timer_mask()" the SW portion writes the requested mask to the co-processors timer controller register. Then the SW portion receives the call to the "timer_halt()" and writes the timer_halt command to the co-processor. Finally, in the SW portions point of view, the control is passed back to the co-processor. The co-processor, upon receiving control and finding the timer_halt command remains idle until the timer sends an event through the controller. Naturally the co-processor can also be waken up by the SW portion of the JVM to provide interrupt services.

The timers may also be used with the methods and classes presented in The Real-Time Specification for Java [5]. If this method is used, the SW portion of the virtual machine must be coded to suit the specification. It must be noted that the timers presented in this paper are suitable for use as real time

timers, not as timers for simulation time or cpu time. If the Java application requires timers based on alternative time lines, they must be implemented in the SW portion of the virtual machine.

V. Example: A Simple Chess Clock

As an example of using the timers and external timing connections a simple chess clock is presented. The whole software is coded in Java, using standard techniques except for the time keeping and event handling. The user interface is assumed to be a simple panel display showing the times remaining for each player and two buttons to select whose turn it is.

The buttons are connected to two of the external timing sources. The buttons are equipped with pull-up resistors and connect to ground, if pressed. At the beginning the period registers of two timers are set to a value corresponding one second (depends on the speed of incoming clock signal). Both of the timers remain idle at this time. The external timing sources are set to send an event on the falling edge. Now the program activates the the white players timer, and remains idle, waiting for an event from the timer controller. If the timer sends an event to signal the processor that a second has elapsed, the display is adjusted as necessary. If the white players button is pressed, signaling that the white player has completed his turn, the white players timer is halted and the black players timer is (re)started. The procedure is repeated, interchanging white and black for every turn, until the game is over or either player runs out of time. The starting and restarting of the timers is done simply by setting their respective prescalers to either 0 (stopped) or 1 (running). A flowchart of the operation is shown in Figure 6. In the wait for event -state the leaving edge is chosen based on the event received. The update display -state also directs control to two different directions, and the selection here is done based on remaining time.

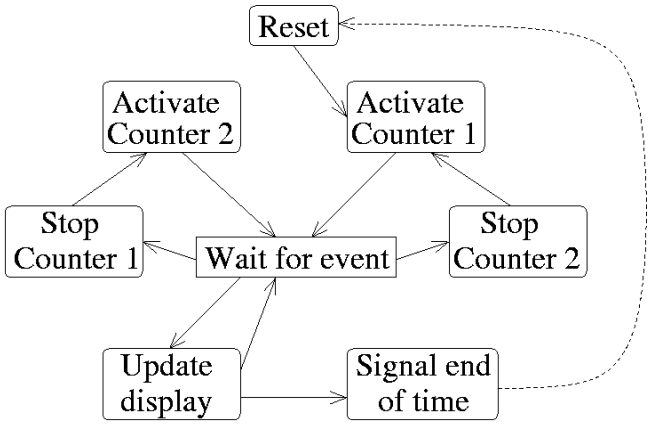

Fig. 6. Flowchart of the chess clock

VI. Conclusions and Future Work

A novel timer structure for real time flow control in Java execution was presented. The structure takes into account the peculiarities of the Java bytecode streams, and provides reasonable performance with low power usage. The predictability of real time behavior is greatly enhanced.

The approach chosen here is energy aware, even in comparison to running compiled C code on the CPU. This is achieved by using asynchronous circuit techniques. Asynchronous circuits excel especially in situations where the workload of the processing unit is not constant. Synchronous systems waste a lot of power by clocking internal latches even when no processing is done. This type of energy wasting is not present in asynchronous system. Also the current consumption of asynchronous systems (usually) is more stable, resulting in lesser noise.

We proposed integrating hardware timers to the JPU and expanding the JVM with own functions designed to make use of the timers. These functions would make it possible for users to execute pieces of code based on timer information. The proposed set-up has 4 internal timers and 4 external timing connections. All of these would be configurable and they could be masked out, when not needed. This would bring the real time performance of Java applications to a new level.

We plan to continue with designing the REALJava co-processor as follows. The co-processor concept and hardware-software co-operation will be verified by building a FPGA demonstrator. The FPGA based JVM will also be used to validate design decisions, such as weather or not to include a floating point unit to the ALU. At the same time dedicated memory structures supporting advanced garbage collection (GC) methods are investigated in order to minimize the fluctuations in execution times caused by traditional GC algorithms. After the FPGA phase we will continue manufacturing the co-processor as a separate ASIC. Later a larger SoC / NoC system with several CPUs and JPUs will be designed to implement a real-life application.

References

[1] J. Hennessy and D. Patterson. “Computer Architecture: a Quantitative Approach”, Second Edition, Morgan Kaufmann Publishers, Inc., 1996.

[2] Z. Liang, J. Plosila, and K. Sere. “Asynchronous Java Accelerator for Embedded Java Virtual Machine”, *In Proc. of IEEE CAS Symposium on Emerging Technologies, Frontiers of Mobile and Wireless Communication*, Shanghai, China, June 2004.

[3] P. Liljeberg, J. Plosila, and J. Isoaho. “Self-Timed Communication Platform for Implementing High-Performance Systems-on-Chip”, *the VLSI Integration* Journal 38, Elsevier, 2004.

[4] T. Lindholm and F. Yellin. “The Java Virtual Machine Specification”, Second Edition, Addison-Wesley, 1997.

[5] The Real-Time for Java Expert Group. “The Real-Time Specification for Java”, Addison-Wesley, 2000.

[6] J. Sparso and S. Furber. ”Principles of Asynchronous Circuit Design - A System Perspective”, Kluwer Academic Publishers, 2001.

[7] T. Säntti and J. Plosila. “Communication Scheme for an Advanced Java Co-Processor”, *In Proc. Norchip 2004*, Oslo, Norway, November 2004.

[8] T. Säntti and J. Plosila. “Architecture for an Advanced Java Co-Processor”, *In Proc. ISSCS 2005*, Iasi, Romania, July 2005.

[9] T. Säntti and J. Plosila. “Instruction Folding for an Asynchronous Java Co-Processor”, to appear *In Proc. Tampere SoC 2005*, Tampere, Finland, November 2005.