A Scalable Architecture for Multi-threaded JAVA Applications

Michael Mrva, Klaus Buchenrieder, Rainer Kress
Siemens AG, Corporate Technology, ZT ME 5
D-81730 Munich, Germany
{Michael.Mrva|Klaus.Buchenrieder|Rainer.Kress}@mchp.siemens.de

Abstract

The paper presents a scalable architecture for multi-threaded Java applications. Threads enable modeling of concurrent behavior in a more or less natural way. Thus threads give a migration path to multi-processor machines. The proposed architecture consists of multiple application-specific processing elements, each able to execute a single thread at one time. The architecture is evaluated by implementing a portable and scalable Java machine onto an FPGA board for demonstration.

1. Introduction

Java was originally designed for the use in embedded electronic applications to overcome the major weakness of C and C++ [2]. Later on it was retargeted to the internet. Due to this, Java became very popular. It allows to run highly interactive, dynamic, and secure applets and applications on networked computers. Nevertheless, the use of Java for embedded systems is still interesting. It allows the building of graphical user interfaces very simply in the same programming environment. Maintenance can easily be done via the inter- or intranet by reloading class files.

Embedded systems have fewer resources and more specialized functionality than a network computer, such as set top boxes, printers, copiers, and cellular phones [5]. These devices have special constraints like small memory, low power, specified performance, etc. These requirements are fulfilled from so-called application-specific integrated processors (ASIPs) [8]. ASIPs are optimized processors for a few (or in most cases one) applications. Since such programs never (or less frequently) change, the hardware can be highly optimized towards specific goals. The idea, described here, is to combine the advantages of both Java and ASIPs, retrieving the Java application-specific integrated processor (JASIP).

The prototype of the JASIP architecture is realized on an FPGA board having in total a gate capacity of 100000 gates and an attached memory board. The prototype consists of several (currently up to three) JASIP processing elements (JPE). The architecture also scales well to more JPEs if the required gate capacity is available (figure 1).

Java allows a high degree of parallelism because of its multi-threading capabilities. Threads provide efficient multiprocessing and distribution of tasks. They enable modeling of concurrent behavior in a more or less natural way. Though threads are often helpful, they usually are not always trivial to master because of race hazards leading to corrupted data, deadlocks, fairness, or starvation. Java simplifies thread programming to a certain degree since multi-threading primitives are built into the language. Threads give a migration path to multi-processor machines. Thus the JASIP architecture is well supported by performing a single thread at one time on a JPE. The degree of parallelism of the application lies in full control of the user by using as much concurrent threads as possible.

The JASIP architecture is evaluated by implementing a portable and scalable Java machine as a demonstrator. Therefore the FPGA board was hooked up to a graphics liquid crystal display module (LCD) for presenting results visually.

The paper is organized as follows: Section 2 gives an overview on Java and its execution mechanism. The JASIP architecture is described in section 3. Then its programming environment is shown in section 4. The example is presented in section 5. Finally, the paper is concluded.

2. Java and its execution mechanism

Java is both, a compiled and interpreted language. Java source code is compiled into simple binary instructions called bytecode. But whereas C++ is compiled to native instructions, bytecode is a universal format. These instructions are interpreted by a Java run-time interpreter based on a Java virtual machine (JVM) [7], [6]. A general JVM starts executing an application with the method *main* of the selected class [2]. The following steps have to be performed.

Loading. If the class is not loaded already, the JVM uses a class loader to find and load the binary representation of the class, called class file. The class file contains the compiled version of either a Java class or a Java interface. The format of such a file is documented in the JVM specification [3].

Linkage. After loading the class file, it must be linked. This step includes verification, preparation, and optionally resolution. Verification checks that the loaded code obeys the semantic requirements of Java and the JVM. This includes the verification of the operation code, the signature of the methods, etc.

Preparation involves the allocation of static storage and data structures which are used by the JVM, such as creating *static* fields for a class or interface and the initialization of these fields to the default values. The preparation step does not require to execute code up to now.

Resolution means checking symbolic references of the loaded class to other classes and interfaces by loading these classes and interfaces. If the symbolic references are correct, they can be replaced by direct ones. When a problem is encountered in one of these link steps, an error is thrown.

The resolution step is optional at the time of linkage. Either the symbolic references are resolved very early in this step, or late, resolving the references only when they were actively used. The references are resolved one at a time or never when not used in the program. The early resolution is preferred in systems where the provided memory is fixed and all necessary class files are available and loaded at the beginning, e.g. in embedded systems. The resolution of symbolic references can then be shifted into "compile-time" rather than using "run-time" for it. On the other hand, systems with widely distributed sources like networks, loading of classes increases traffic. Thus it is more suitable to fetch classes only on demand, avoiding unnecessary loading.

Initialization. Initialization consists of the execution of any class variable initializers and static initializers of the currently loaded class. But before this class can be initialized, its superclass and also their superclasses must be initialized. Since Java is multi-threaded, the initialization of a class or interface requires careful synchronization, because another thread may try to initialize the same class or interface at the same time. A well defined procedure exists to avoid error in such situations, as described in [2].

3. JASIP architecture

The Java application-specific integrated processor (JASIP) is targeted to multi-threaded applications. A thread independently executes Java code that operates on values and objects residing in a shared memory. Compute instructions only operate on stack data and never on data of the shared memory. The JASIP architecture directly reflects these memory accessing schemes (figure 1).

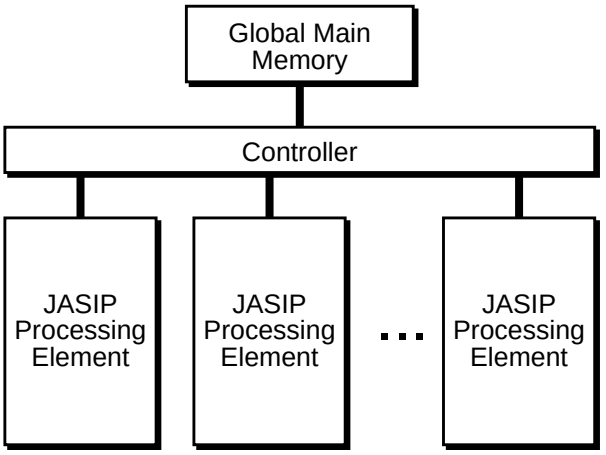

Fig. 1. Overview on the JASIP architecture

A global main memory is used for storing the class files and the heap for the objects. Each JASIP processing element (JPE) executes a single thread at one time. It implements a kind of JVM with own local memory for the bytecode of the current method and the method frame. The method frame contains the local variables, the operand stack, and the frame information.

The controller is responsible for two major tasks: for the scheduling of threads and for supervision of accesses to the main memory. Each available JPE gets a thread ready for execution. This thread is then completed unless the user program blocks the thread. After finishing, a new thread is assigned to this JPE by the controller.

A single JPE is allowed to access the main memory at one time. A token which is passed in a round-robin manner between the JPEs grants this access. Each time a read or write cycle has to be performed, the JPE waits for the token. After all actual memory cycles are finished, the token is released. This means, e.g. in the case of loading the bytecode from the class file to the local program memory, the token is released after complete loading. The performance loss due to blocking other main memory accesses is tolerable, since loading the byte code to the local memory of a JPE happens rarely.

3.1 JASIP processing element

A JASIP processing element (JPE) consists of four main parts: the datapath with the ALU, the program memory, the data memory with its addressing unit, and the access to the global memory (figure 2). The sizes of the memories and the datapath width are adjustable according to the size of the problem.

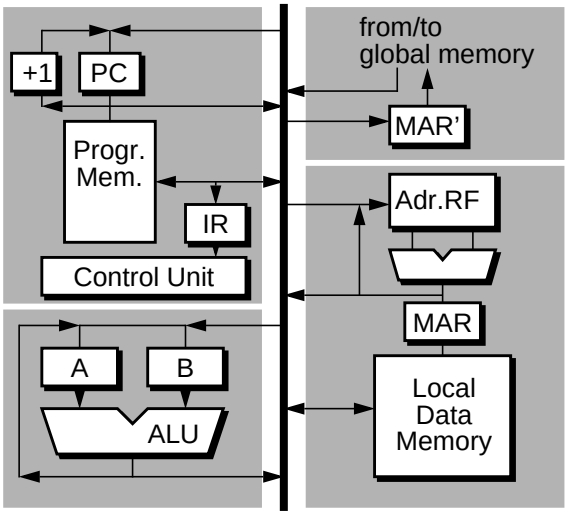

Fig. 2. A single JASIP processing element

Datapath. The datapath consists of an ALU and two registers (A and B). Register A is able to load its upper byte and lower byte separately. So in case of a 16 bit datapath, a word can simply be built out of two byte values. An instruction profiling of the current application leads to the instruction set of the ALU. This means, besides the basic functionality of the ALU, like add, subtract, shift, and conditions, custom operators can be added. Since each JPE should be able to run any thread of the application, all JPEs use the same architectural parameters.

Program memory. The program memory contains the bytecode of the actual method. Its width is 8 bit, as specified in the Java virtual machine. The program counter (PC) can be loaded, stored (in the method frame), or incremented. The instruction register (IR) triggers the control unit.

Local data memory. The local data memory contains the actual method frame and the frame of the calling method. Thus a reloading of data after a return from the called method is not necessary. The address generation unit for this memory consists of a small ALU with register file (Adr.RF). The register file contains the references to the top of the stack (*optop*), the local variables (*vars*), and the frame information (*frame*). A memory address register (MAR) addresses the memory. Figure 3 shows the organization of the local memory. Each time a new method is invoked, a new method frame is created (figure 3 right side). The stack is constructed to allow overlap between the methods, enabling direct parameter passing without requiring any copying of the parameters, like in the picoJava architecture [4]. Parameters values of the operand stack become part of the local variables of the new frame. The frame info is used to store the references *optop*, *vars*, *frame*, the return PC etc. The new references of the new method frame can easily be computed since the size of the memory for the local variables and the frame info is known. The operand stack is empty at the beginning. In case of a return to the caller method, the old values are restored from the frame info.

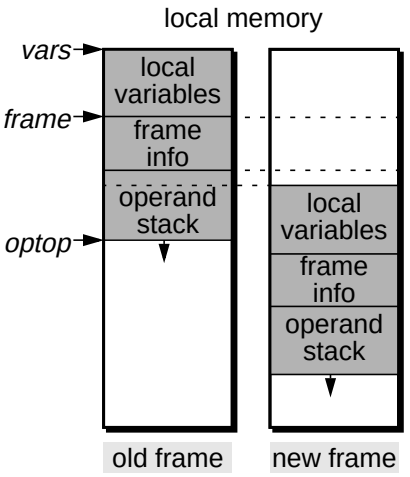

Fig. 3. Local memory with its method frames

In the current prototype there is no mechanisms to avoid stack overflow errors, like e.g. a dribbling stack mechanism. The size of the memory is adjusted according to the requirements of the Java program following the ASIP approach. These requirements are given by the JASIP programming environment.

Global memory access. The global memory is addressed indirect via a lock table (figure 4). The object reference from stack, points to a lock table with the actual base address of the main memory and a bit indicating whether the targeted object is locked. The lock table is enlarged, each time a new object is created. If the lock is free (zero), the addition of the base and the offset (taken from the bytecode) results in the actual address. Otherwise, the access is denied and the token allowing memory access of the current JPE is transferred to the next JPE. After the token passed one round, another access can be tried.

Garbage collect. Java removes objects which are no longer needed with a low priority garbage collection process. There is no special algorithm specified in the Java language. Thus no behavior can be guaranteed. In the JASIP architecture, the controller is responsible for garbage collection. It just again uses the space on top of the heap which is not used anymore. So currently no gaps in the heap are removed and no *finalize()* method is evaluated. Of course the entries in the lock table are removed too.

3.2 Handling of threads

In addition to the platform-specific run-time system, Java has approximately 22 fundamental classes that contain architecture-dependent methods [7]. These native methods serve as a gateway to the real world. Though Java, in general, is architecture-neutral, the concrete thread package is platform-dependent (least common denominator approach).

A new thread is born when an instance of the *java.lang.thread* class is created. The thread remains idle until its *start()* method is called. The thread then wakes up and proceeds to execute the *run()* method of its target object. Once it is started it runs until the *run()* method finishes, the thread's *stop()* method is called, or it is destroyed. All runnable threads are maintained in a list by the controller of the JPEs. On every free JPE, a thread from the top of the list is scheduled. Usually, this thread runs until it finishes, realizing a round-robin schedule.

There are several methods which allow to control the execution of a thread. Figure 5 shows the relations of theses methods in the thread state diagram. The bubble 'runnable' contains also the bubble 'running'. For graphical reasons, it is drawn separately.

The methods *suspend()* and *resume()* operate on the current thread object and therefore require no argument. They are used to arbitrarily pause and restart the execution of a thread. In the JASIP architecture, a thread is paused, but it is still remaining on its JPE. Obviously, if the threads in all JPE are suspended, a deadlock occurs and an error is thrown. This decision was taken because saving the status of a thread will require very much time. The local data memory has to be saved in the main memory. The idea of *suspend()* is used in situations where the setup of a thread is very expensive. So if *suspend()* is used rarely in these cases, it is cheaper to freeze the status of a JPE instead of saving the status into the global memory.

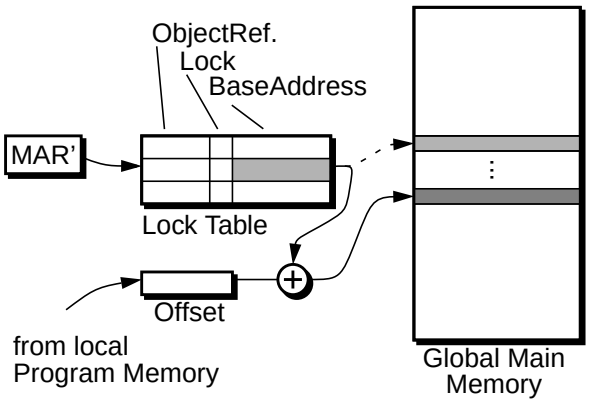

Fig. 4. The lock mechanism of the global main memory

The method *sleep()* will cause the thread to sleep for a given period of time. A counter in the controller will take care of waking up the thread. During sleeping, the thread still remains on its JPE.

The most common need for synchronizing threads is to serialize the access to an object. Only a single thread at a time can manipulate on an object. This is done by marking the method with the keyword *synchronized*. When the method is called, a lock is given the current object and no other method has access to that memory (see also the part of section 3.1 *'global memory access'*). The methods *wait()* and *notify()* extend this capability. A thread gives up its lock by using *wait()* at any arbitrary point and then waits for another thread to give it back by calling *notify()*. In this case the thread still remains on its JPE, just giving away its lock and waiting to receive it back. When *wait ()* is called with an argument, the thread will continue working after the time specified by the argument. With *notifyAll()* all waiting threads are continued.

Java allows to change priorities of threads. In the current prototype a thread with higher priority is scheduled earlier, but no other thread is pre-empted. If each JPEs is executing a thread, the high prior thread has to wait until a running thread finishes on a JPE. Thus calling *yield()*, a thread will not give up its JPE. It still will continue. Due to the availability of multiple JPEs there is no necessity to yield a thread. I/O blocking is transparent and not utilizable by the user.

4. JASIP programming environment

Programming of the JASIP consists of two parts: synthesizing the hardware and compiling the Java source code (figure 6).

Java compiler. The source code can be complied with any Java compiler to get the Java bytecode (class files

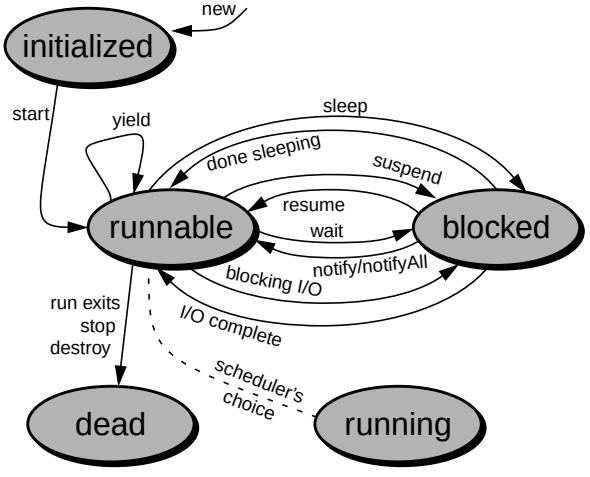

Fig. 5. Thread state transition diagram

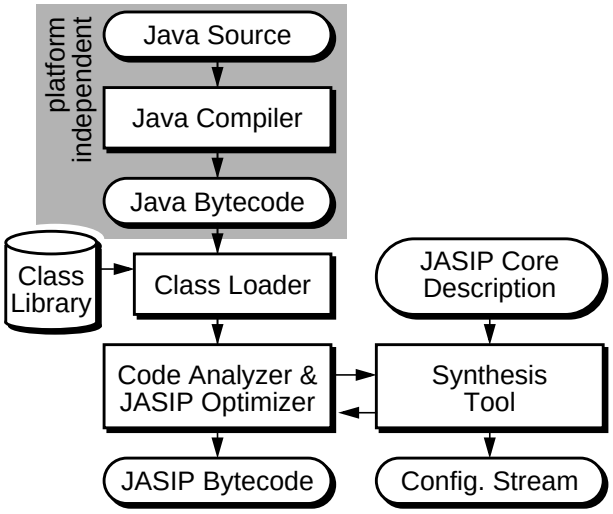

Fig. 6. Overview on the JASIP programming environment

of the source code). This step is completely platform independent. While an interpreter based on the Java virtual machine start now with the execution, our approach analyzes and optimizes the bytecode of the current application for the JASIP architecture.

Class loader. The class loader searches for all necessary class files to run the current application and assigns them to actual addresses of the main memory.

Code analyzer. The code analyzer tries to find out how large the program and the local data memory has to be and which instructions occur. Therefore it analysis the hierarchy of the method invocations using the profiling information of the *.prof* file of the *java* interpreter. Each time, a new method is invoked, the local variables, the frame information, and the rest of the operand stack is added on the local data memory. Thus, the deeper the method hierarchy, the larger the memory has to be. The size of the requested stack depth and the number of local variables required is taken from the class file. Of course, the same is done with the program memory. The memory sizes, which directly influence the necessary addressing space, determine the datapath width of the address generation unit. Depending on the occurring instructions in the bytecode the complexity of the ALU is chosen. These architectural parameters are passed to the synthesis tool on the hardware part.

Synthesis tool. Input to the synthesis tool is a generic core description of JASIP. The tool itself consists currently of the Synopsys synthesis tool [9] and schematic entry using Xilinx tools like X-BLOX [10]. Output to the JASIP analyzer is whether the requested architecture can be build or not, and for configuring the FPGA board the configuration bit stream.

JASIP optimizer. The JASIP optimizer resolves the symbolic references in the Java bytecode to actual addresses following the principle of early resolution. Thus speeding up the application. How this is done, is shown in the two examples method invocation and accesses to fields.

Field access. In the case of a field access via *putfield*, a JVM uses the two indexbytes from the bytecode which follow the keyword *putfield* to construct an index to the constant pool of the current class. The constant pool item will be a field reference to the class name and a field name. The item is resolved to a field block pointer which has both the field width in bytes and the field offset in bytes. The field at that offset from the start of the object referenced by the object reference (on the operand stack) will be set to the value on top of the operand stack [3]. In the JASIP bytecode, the field width and the field offset directly follows the keyword *putfield*. Thus the main memory can be directly set to the value on top of the stack. It will be addressed via the object reference from the stack and the offset.

Method invocation. In the case of a method invocation, the arguments are on top of the stack followed by the object reference. The lookup in the constant pool is similar to the field access. If the method is not found in the current class file, its superclass is searched, and so on (in the case of *invokevirtual*). Details can be found in [2]. In the JASIP architecture, the address of the required method block follows directly the keyword *invoke*. The number of arguments are found in the method block. If the method is marked synchronized, the lock of the object reference set. The object reference and the arguments become the initial local variables of the new method (see also figure 3). Execution continues with the first instruction of the new method, as specified in the JVM specification.

5. Example

As an example we developed a Java applet that draws an oscillating parabola curve on the screen. It consists of two classes WPDriver (WP stands for WavePainter) and StripePainter. The idea is to partition the applet frame into a number of vertical stripes and to assign a separate thread to each stripe. In figure 7, eight threads are shown for simplicity.

The task of each thread is to draw an appropriate part of the desired curve on its own stripe. To accomplish the job, we used a pattern which we called the “Wrapper-Driver-Runner” pattern. In its abstract form (figure 8; in UML notation [1]), it consists of a subclass of class Thread and a second class Driver that implements the interface Runnable.

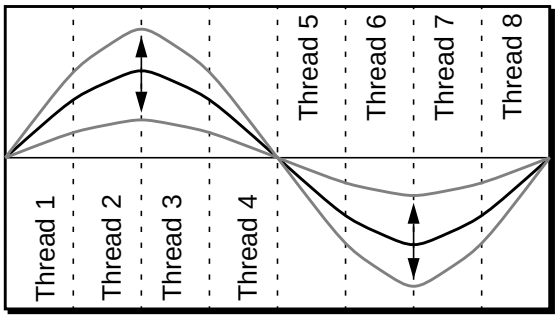

Fig. 7. The oscillating parabola wave and the responsible threads

The superclass Thread serves as a wrapper for the Runnable Driver which additionally becomes the simultaneous parent of all instances of the Subclass. The concrete application of the "Wrapper-Driver-Runner" pattern is derived from the abstract pattern by substituting Subclass by StripePainter, instance by runners, n by grid, and Driver by WPDriver, resp. A part of the Java source code of the example is shown in listing 1.

Prototype. The proposed JASIP architecture will be evaluated by implementing the above Java example. To build the demonstrator a proprietary FPGA board was hooked up to a high resolution graphics liquid crystal display module (LCD). The FPGA-board carries four Xilinx 4025 chips with a total of 100000 equivalent gates. Three FPGAs are used for implementing a JPE each and the forth is used for the controller. Once the FPGA-board is configured, the platform is running stand alone, and may be used as a portable prototype of the implemented application.

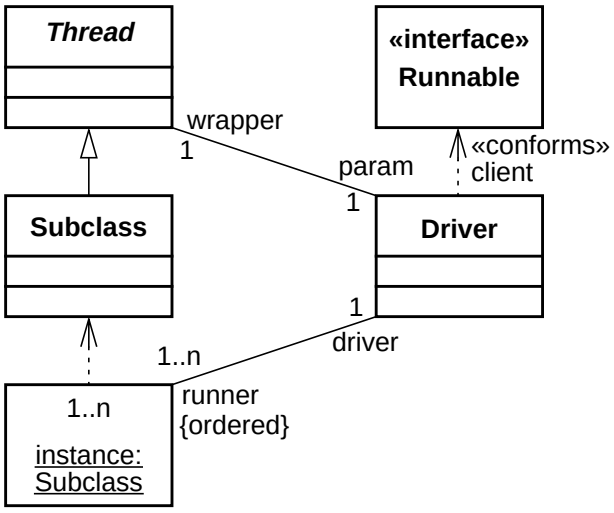

Fig. 8. Abstract "Wrapper-Driver-Runner" pattern

```
public class WPDriver extends
         java.applet.Applet
         implements Runnable {
  ...
  public void start() {
    for (int i = 0; i < grid; i++) {
      runners[i] = new
        StripePainter(this, i);
      runners[i].start();
    }
  }
  ...
}

public class StripePainter extends
         Thread {
  ...
  StripePainter(Runnable parent, int
         idx) {
    super(parent);
    iMT = idx;
  }
  ...
}
```

Listing 1. Part of the Java source code of the "Wrapper-Driver-Runner" example

The LCD module provides a resolution of 480 x 128 pixels on a screen of 242 mm by 69 mm. The whole screen is divided into four independent segments. A single segment supplies 240 x 64 pixels (figure 9).

The 12-pin edge connector at the lower right corner establishes the interface to the display control circuit. The LCD module clocks pixel data serially into the specific segment with a clock rate of up to 2.5MHz. Each of these segments is addressed individually and allows to write to the display in parallel mode. This important feature results in a speed up factor of four accessing the display memory. Shared memory access times are crucial to the overall performance of highly parallel systems. The display memory is an exclusive resource, and is usually the bottleneck for data exchange. We used an interleaved memory structure, that is reflected by the display memory organization. This speeds up the overall system performance significantly.

In our example, the demonstrator needs high data exchange rates from the FPGA board to the display to draw smooth curve-movements along the display cycles. An important task to gain maximum performance, is balancing the processing time of each JPE corresponding to the display access rate. Ideally, all JPEs are calculating data in parallel. Using the presented scheme no JPE has to access memory during a display read cycle, since each JPE has his own local memory associated. In rare cases a JPE needs to write to the display memory in a display-read-

cycle, execution of the JPE will be blocked until the end of this display-read-cycle.

In order to gain highest flexibility, which is a basic feature of the presented JASIP architecture, a commercial display-controller circuit is not considered for this application. Synthesizing individual I/O drivers according to the specification gives maximum freedom in scaling the performance of the display mechanism. If certain applications require other resolution factors, we will provide a straight forward method to connect such devices as well. The presented JASIP architecture controls signal generation and data conversion required for the LCD module by its own. So no special hardware interface is required. The LCD module allows direct connection to the FPGA-board as depicted in figure 9. A photograph of the demonstrator is shown in figure 10.

6. Conclusions

A scalable architecture for multi-threaded Java applications called JASIP has been presented. It combines the advantages of Java with its simple mechanism to handle threads, and application-specific processors (ASIPs) with their highly optimized architecture. The parallel and scalable architecture of JASIP directly reflects the execution of Java source using concurrent threads. Thus it provides an

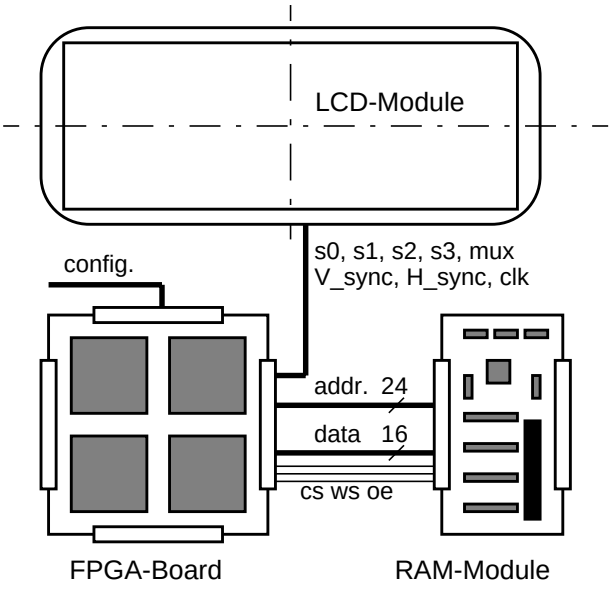

Fig. 9. LCD-Interface

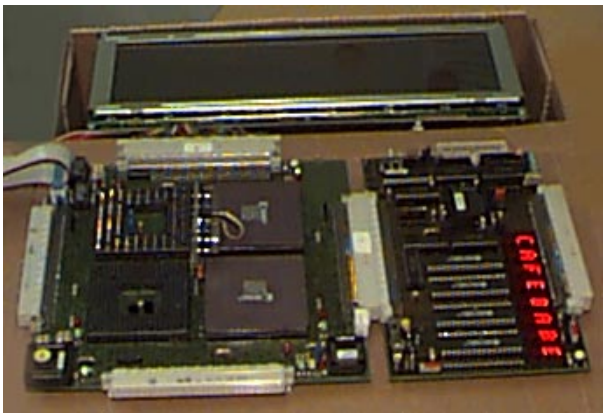

Fig. 10. Prototype platform showing the FPGA board and the display

architecture which is optimized to the current problem and archives high speed-ups due to the parallel execution of threads. A prototype of the JASIP architecture is realized on an FPGA board having 100k gate capacity.

Future work will consist of the integration of pipelining, a dribbling stack mechanism for the local data memory, as well as supporting exception handling.

7. References

[1] G. Booch, I. Jacobson, J. Rumbaugh: The Unified Modeling Language for Object-Oriented Development; Sept. 1996, http://www.rational.com

[2] J. Gosling, B. Joy, G. Steele: The Java™ Language Specification; Addison-Wesley, Reading Massachusetts, 1996

[3] N.N.: The Java Virtual Machine Specification; Release 1.0, Sun Microsystems Computer Corporation, Aug. 1995

[4] P. Knarr, M. Breya: picoJava I™ Microprocessor Core Architecture; White Paper, Sun Microelectronics, Oct. 1996

[5] D. Kramer: The Java™ Platform; A White Paper, JavaSoft, Mountain View, CA, 408-343-1400, May 1996

[6] P. van der Linden: Just Java; The Sunsoft Press, Java Series, 1996

[7] P. Niemeyer, J. Peck: Exploring Java; O'Reilly & Associates, inc., Sebastobol, CA, May 1996

[8] J. Sato et. al.: PEAS-I: A Hardware/Software Codesign System for ASIP Development; IEICE Trans. Fundamentals, vol. E77-A, no. 3, March 1994

[9] N. N.: Design Compiler, vers. 3.0; Synopsys, Inc., San Jose, CA, Dec. 1992

[10] N. N.: X-BLOX User Guide; Xilinx, San Jose, CA, April 1994