Real-time Java Processor for Monitoring and Test

Martin Zabel, Thomas B. Preußer, Rainer G. Spallek
Technische Universität Dresden
{zabel,preusser,rgs}@ite.inf.tu-dresden.de

Abstract — This paper introduces a new processing platform, called SHAP, which implements the agent concept in hardware and is ready to use in high-security applications. The SHAP concept is described and its implementation by the SHAP microarchitecture is explained. Also a short overview about the SHAP API is given.

1 Introduction

The concept of agents and multi-agent systems is widely known in computer science. It serves well for many different application fields, and thus has a high potential for upcoming developments.

This paper introduces a new processing platform, called SHAP, which is based on the agent concept and thus is ready for application in many different fields of information technology. One of these fields, "Monitoring and Test" is considered here.

The main part of the SHAP concept introduced here is a Java processor, which, on one the hand, implements the properties of an agent. On the other hand the concept provides a complete Java Virtual Machine (JVM) with the focus on autonomy, real-time and security.

This paper is divided as follows. At first, the SHAP concept is introduced in Section 2. Its implementation is covered by Section 3, and a short overview about the API is given in Section 4. Last but not least, Section 5 lists the work to be done in the future.

2 The SHAP Concept

The abbreviation SHAP reads "Secure Hardware Agent Platform" and claims the main targets. The platform is *secure by design*. Hence, it is ready to use in high-security tasks. It is an *agent platform* providing the associated properties [1], i.e., autonomy, responsiveness, pro-activeness, and social interaction, to the user. And finally, it is implemented in hardware without any interference through an operating system to achieve the best possible performance.

The agent concept also fits well for the application field of monitoring and test. Autonomy is a necessary criterion to implement transparent monitoring of other devices. Autonomy is also required at thread level to enable independent monitoring of many devices by just one (complex) monitor in parallel. Reactiveness is the second criterion required for successful monitoring. It demands that the agent perceives its environment and responds in a timely fashion [1]. Applying to monitoring, the agent should never miss a state change of the monitored device and should react to any event in the shortest time pos-

sible. This behaviour is only possible if the agent platform needs real-time constraints.

The SHAP concept covers all required parts to execute Java bytecode directly on hardware with the focus on real-time and autonomous constraints as required above. The main part is the SHAP microarchitecture constituting the SHAP Java Virtual Machine (JVM), which executes the Java bytecode. The term "virtual" seems to be unfitting here, but that is not true. SHAP also executes Java bytecode which cannot address special features of the underlying architecture directly. The distinction is that SHAP does not require any operating system in the middle. The second main part of the concept is the SHAP Linker which is described further down.

The decision in favor of Java bytecode has been made because, in contrast to other well known instruction sets, Java bytecode and the associated Virtual Machine Specification [2] already includes important security features. The lack of pointer arithmetic prohibits memory accesses beyond the object or array in process. Hence, a buffer overflow, which is the cause of undesired stack and heap modifications used by computer viruses, cannot occur by design. Also the implicit type checking prohibits incompatible assignments.

Before code in a Java class can be executed, the class has to be loaded, linked and initialized in this order. According to the VMSpec [2, § 2.17] the meanings of these activities are:

Loading: means locating the binary form of a class, typically stored in `.class` file.

Linking: is itself divided into three subprocesses: At first, the class is verified wether if it is structural correct, e.g., valid opcodes, valid branch targets, every instruction obeys type discipline. After that, the class is prepared by creating and initializing static fields. An implementation may also create helper structures, like method tables, in this step. The last subprocess is resolution, where all symbolic references (in terms of strings) are resolved. Resolution may either be static and thus done at once now. Or it may be lazy/late and consequently done at usage time, e.g., invoking of method, access to fields.

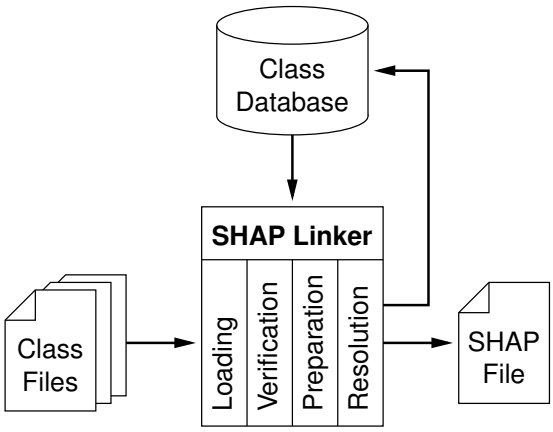

Figure 1: The Linking of Class Files to a SHAP File

Initialization: just runs the static initializer of a class in an order also defined by the VMSpec.

To enable real-time execution, at least a static resolution has to be chosen. Otherwise the time of invoking a method would not be predictable. Also initialization has to be done directly after linking. From these facts and to avoid a complex bootstrap class loader, which must analyze the class file on the target, the steps of loading and linking are shifted to a host system which provides the class files. On this host system, a set of class files is linking into a SHAP file by the SHAP Linker as depicted in Fig. 1. The classes are verified, prepared and completely linked so that they can be directly executed by the SHAP JVM after downloading.

To enable dynamic loading of additional class files, like additional applications, a class database has been added. This database stores important information about already known classes. New classes are linked against these classes and after that also stored in the class database.

Note, linking is not bound to the host system.

After downloading the SHAP Linker's classes as well as the class database to the target SHAP system, loading and linking can be done by the SHAP system on its own as well.

The SHAP file must be digitally signed to keep its integrity during further distribution. The SHAP JVM relies on that the SHAP file is correct.

3 The SHAP Microarchitecture

The SHAP microarchitecture has to fulfill a couple of requirements defined by the agent concept as well as the Java Virtual Machine (JVM): real-time execution of Java bytecode and autonomous control flows. As a target for any processor, a high troughput at a low latency of instructions is requested. Because Java bytecode is based on a stack machine and consequently mostly each bytecode instruction depends on its predecessor, a high throughput is achieved if low latency is given.

The microarchitecture is divided into four parts as can be seen in Fig. 2. It consists of:

The CPU: which directly executes Java bytecode, controls all other parts and has direct access to every other component through dedicated busses.

The memory managment unit (MMU): which manages the main memory which is equivalent to the Java heap. The heap stores the Java objects as well as information about classes (class objects).

The bytecode instruction cache: which caches the current Java bytecode to execute. Because the Java methods are stored inside the heap (inside the class objects), this component needs access to the MMU bus. The connection to the I/O bus is for transfering commands. The caching is implemented to lighten the MMU.

The I/O subsystem: comprises several I/O devices, e.g., UART, LCD, PS/2 port, and so on, which are all connected to the I/O bus.

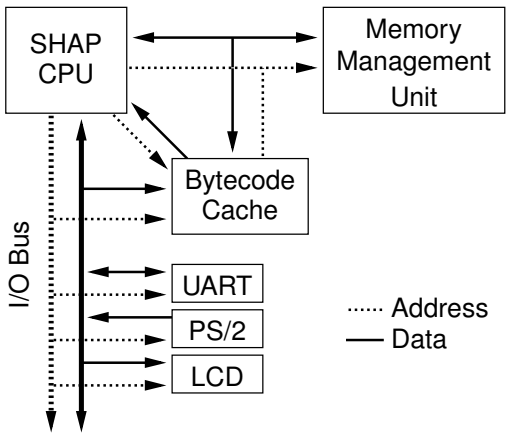

Figure 2: The SHAP Microarchitecture

A more detailed description is given in the following paragraphs.

At first, we take a look at the CPU. The Java bytecodes to be executed have different complexities and thus execute in different number of cycles. For example, the `iadd` instruction just takes one cycle because it only has to add the two topmost stack elements. In contrast to that the `invokevirtual` instruction has first to access the constant pool of the class, resolve the method pointer, load the method header, cache and start the method. It is possible to implement a huge state machine which handles all instructions directly but this is difficult to maintain. Hence, inspired by JOP [3], we decided to execute the Java bytecodes through microcode which is stored inside on-chip memory. This preserves the advantage of short method code and adds the ability of easy programming through microcode instructions as well as firmware updates. The count of logic is nearly the same in both variants because the state machine may also be implemented by meory.

The microcode uses 9-bit instructions because some of the real 52 instructions take a 6-bit immediate. This way, each instruction can be fetched in only one cycle which also results in a simple instruction decoder. 9-bit appears a little

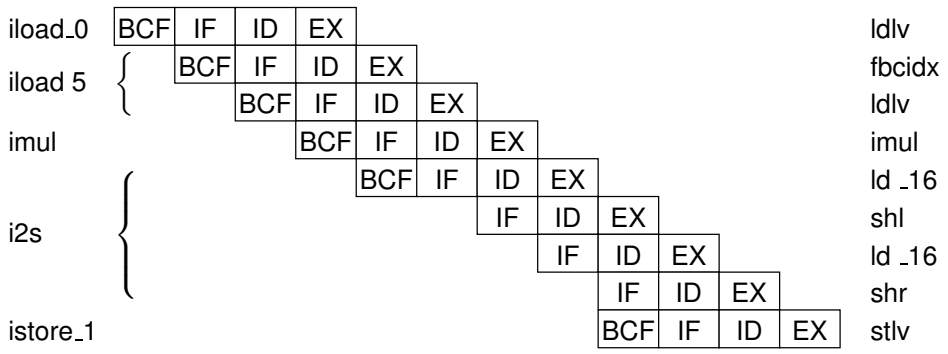

Figure 3: Execution of Java Bytecode (l.) through Microcode (r.)

bit uncommon but is not a problem with the on-chip memory of current FPGAs. The microcode instructions are executed in three stages: instruction fetch (`IF`), instruction decode (`ID`) and execute (`EX`). Together with an additional stage for bytecode fetch (`BCF`) we get a 4-stage pipeline, which is illustrated in Fig. 3. Note that the `BCF` stage is only executed when a new Java bytecode instruction or a bytecode immediate has to be fetched. All memory accesses to the internal stack are requested in the `ID` stage and fixed in the `EX` stage. Thus no additional pipeline stage, like `MEM`, is needed. Memory accesses to the heap are described below.

Most of the microcode instructions are executed in one `EX` cycle but some need more than one resulting in pipeline stalls. Yet, the number of `EX` cycles and, thus, the execution time is known for all instructions and does not depend on the input data. Hence, for a given piece of microcode, like the program for a bytecode instruction, the execution time (in cycles) can be calculated in advance. Continuing, one can calculate the execution time for a piece of bytecode, a Java method and a Java thread. Thus calculation of response times in advance fulfills the real-time constraint. Care has to be taken only, if loops are encountered in bytecode as well as microcode, which may depend on input data.

The CPU includes an integrated stack and thread management to fulfill the requirement of autonomous control flows. The stack also uses on-chip memory of the FPGA so that all memory accesses can be done within one cycle. Furthermore, the two topmost stack elements are stored inside registers so that they can be accessed directly by the processing units, see also [3].

The first property of our stack managment is stack frame con- and destruction. If a method is invoked a stack frame is created because of local Java variables which are addresses relative to the current stack frame. Also on return, stack frames are needed because the state of the stack in unknown when Java throws an exception.

To allow n_t multiple threads, a simple technique is to divide the whole stack memory into n_t equal parts. The problem here is that the threads of a typical application have different requirements. Even if the main (huge) thread of an application fits in such a stack part, the (small) helper threads for communication would only utilize a small amount of their stack wasting a lot of memory.

Our solution is to divide the whole stack memory into n_b equal blocks where n_b is significantly greater than the typical number of threads. Each thread is assigned a list of stack blocks which can dynamically grow and shrink and represents a continuous logical stack. The physical blocks themselves need not to be consecutive. This is pictured in Fig. 4. The free stack blocks, which are not assigned to any thread, are handled in a free-list. The system starts with one thread having a list with one block. New threads are constructed by creating a new list and adding one block from the free-list. Each time a thread's

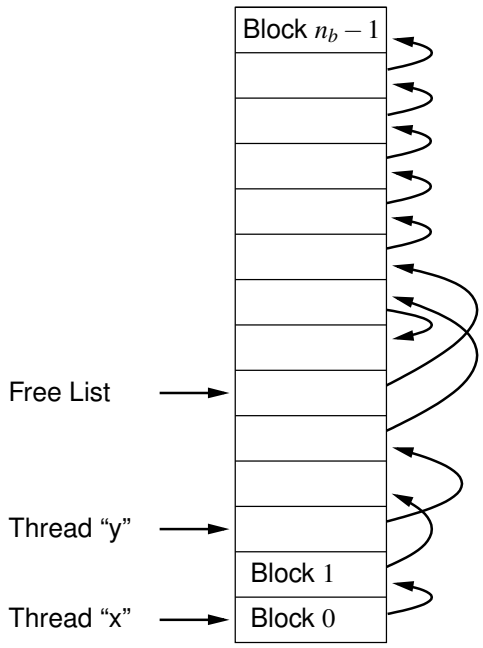

Figure 4: Linking of Stack Blocks to form a dynamic continouos logic stack.

stack has to grow, one block is taken from the free-list and is added to the thread's list. If the last block in a list is not needed anymore because of a shrunken stack, it is removed from the list and put back into the free-list. List manipulation is always done in parallel so it does not influence the execution time of the microcode instructions. Last but not least, the maxmimum number of threads constructable equals to the number of blocks but it is not typical that this corner case is really in use.

Switching between threads is equal to make the target list the active one. But note that the set of threads is not maintained by the CPU themself. The threads, or rather their associated stack block lists, are identified by a handle which must be stored inside a Java object or similar. As a result, hardwired logic is minized because the (complex) scheduling is done by the SHAP JVM and only switching the stack lists is left to the CPU.

There is also a timer integrated into the CPU which triggers an interrupt handler. The handler is implemented in microcode and only called between two Java bytecode instrucions. Thus a bytecode instructions is always executed completely and timing is not affected. The handler is mainly used for preemptive switching of threads by the SHAP JVM.

The MMU is the second most important part. The objects created by JVM are managed through references. This means that the microcode (thus the Java code too) only acts on references which are mapped to real memory addresses inside the MMU. This enables concurrent garbage collection because that way the objects can be moved inside the main memory. The garbage collector is in progress but one condition can already be given: garbage collection runs in parallel inside the MMU and does not influence the execution of microcode nor does it change the execution times of the MMU commands. This is essential not to compromise the real-time confirmation. The MMU commands take a couple of cycles to execute. Thus they run in parallel to the CPU (microcode). For example, at first you activate the reference, then you set the offset inside the object and, finally, you announce the value. Between these three steps other work can be done, as calculating the offset/value directly before setting.

The method cache was added because the `BCF` stage must always execute in one cycle and, therefore, must not produce a pipeline stall. This is conditional on the fact that the execution time of the microcode instructions must be known to fulfill the real-time requirement. The current solution is a single-method cache, which stores the complete code of the actual Java method. The method cache is filled explicitly during `invoke` or `return` Java bytecodes. Filling is done in parallel to the CPU but with the help of the MMU. So you can do any further processing as long as you have no request to the heap. In the current implementation of the SHAP JVM, any method which is less than 28 bytes long, does not stretch the indicated bytecodes. For all other cases the execution time can be calculated if the invoked method is known. (Which may not be true if you invoke a virtual method.) Because of

this limitation, we put the design of a new cache subsystem onto the agenda.

The last part of the SHAP microprocessor is the I/O subsytsem. The I/O bus is divided into address and data lines where the latter is subdivided into read and write lines because tri-stating is not applicable on chip. The I/O device which is activate — allowed to read/write from/to the I/O bus — is selected through the I/O bus by the I/O address. The state of the I/O device is signaled by two flags: "available" , which indicates wether data can be read from the device, and "ready", which indicates that data can be written to the device. The state of an device has to be checked prior to accessing it. This leads to a one-cycle execution of the I/O microcode instructions and enables scheduling (by the SHAP JVM) based on I/O device state.

4 The SHAP JVM

This section just gives a brief overview of the Java features implemented in our SHAP JVM. The implementation of these features will be described in more detail in a future paper or technical report. The features which are implemented completely include:

- all access variants to variables and constants, like load/store variable, load number/string constant,
- integer arithmetic (currently, without division),
- dynamic object and array creation,
- invocation of static and virtual methods as well as interface methods of simple interface hierachies,
- exception handling,
- multi-threading with guaranteed time slots and preemptive round-robin based scheduling to enable autonomous control flows for several agents,
- synchronization through monitors,
- thread-aware I/O functions.

With this features we can implement the classes specified by the "Connected Limited Device Configuration" (CLDC) [4] from SUN. This API includes the most important classes, so you can run a lot of Java applications which do not need a graphical user interface (GUI). But note, the SHAP JVM is not limited to this API. It can be extended by utility classes such as hash maps or a GUI.

The feature currently in work is the support of arbitrary interface hierarchies. Not implemented features are long and floating-point arithmetic. Currently, no decision was made if we implement this in hardware or emulate this in software by Java code.

5 Future Work

The main focus is currently on implementing the real-time garbage collector as well as completing the SHAP JVM. Parallel to this we will perform benchmarks to compare our project with other ones. The next step will cover dynamic class loading and a sophisticated method cache. The linking of many SHAP processors to form a multi-agent system will be another big step.

References

[1] N. R. Jennings, K. Sycara, and M. Wooldridge, "A roadmap of agent research and development," *Autonomous Agents and Multi-Agent Systems*, vol. 1, no. 1, pp. 7–38, 1998.

[2] T. Lindholm and F. Yellin, *The Java(TM) Virtual Machine Specification*, 2nd ed. Addison-Wesley Professional, Apr. 1999.

[3] M. Schoeberl, "Jop: A java optimized processor for embedded real-time systems," Ph.D. dissertation, Vienna University of Technology, 2005.

[4] *Connected Limited Device Configuration Specification, Version 1.1*, Sun Microsystems, Mar. 2003.