The SHAP Microarchitecture
and Java Virtual Machine

Thomas B. Preußer, Martin Zabel
Peter Reichel

Institut für Technische Informatik

Technische Universität Dresden
Fakultät Informatik
D–01062 Dresden
Germany
URL: http://www.inf.tu-dresden.de/
The SHAP Microarchitecture and Java Virtual Machine

Thomas B. Preußer, Martin Zabel, Peter Reichel
Technische Universität Dresden, Germany
Email: {preusser,zabel}@ite.inf.tu-dresden.de, Peter.Reichel@mailbox.tu-dresden.de

Abstract—This report presents the SHAP platform consisting of its microarchitecture and its implementation of the Java Virtual Machine (JVM). Like quite a few other embedded implementations of the Java platform, the SHAP microarchitecture relies on an instruction set architecture based on Java bytecode. Unlike them, it, however, features a design with well-encapsulated components autonomously managing their duties on rather high abstraction levels. Thus, permanent runtime duties are transferred from the central computing core to concurrently working components so that it can actually spent a larger fraction of time executing application code. The degree of parallelity between the application and the runtime implementation is increased. Currently, the stack and heap management including the automatic garbage collection are implemented this way. After detailing the design of the microarchitecture, the SHAP implementation of the Java Virtual Machine is described. A major focus is laid on the presentation of the layout and the use of the runtime data structures representing the various language abstractions provided by Java. Also, the boot sequence starting the JVM is described.

I. INTRODUCTION

Object-oriented programming has led to fast and easy development of complex applications with a short time-to-market. In this domain, Java is very popular as it addresses also portability and security features through the definition of the Java Virtual Machine (JVM). As more and more target systems implement the JVM, the same application can be executed anywhere. Another important feature is the compact Java bytecode leading to small memory requirements and reduced download time. So it is predestined for the usage in resource constrained devices. This enables applications, which can be used anywhere at any time.

As execution of Java bytecode by interpretation is known to be rather slow, Just-In-Time compilation has been used to translate Java bytecode to the host processor’s native instruction set. As this requires much memory, the alternative of executing Java bytecode natively has been considered for embedded Java implementations. The challenge is to combine this execution with real-time constraints to support this application domain.

Our implementation of an embedded Java microarchitecture, called SHAP, fills this gap by implementing new techniques to support multi-threaded general-purpose applications in a secure environment under real-time constraints. Especially, all JVM concepts are considered here.

This technical report is divided as follows: Sec. II surveys related work on Java processors. An overview of the SHAP concept is given in Sec. III, whereas the instruction set architecture is presented in Sec. IV. The design of the microarchitecture is covered by Sec. V inclusive preliminary results. Sec. VI then explains the implementation of the Java Virtual Machine onto the SHAP microarchitecture. Finally, Sec. VII concludes the technical report. The appendix list the microcode instruction set of the SHAP microarchitecture.

II. RELATED WORK

Quite some research has already been carried out on the efficient execution of Java bytecode directly on hardware. This main goal is joined with the mapping of the basic JVM concepts on hardware structures to enable the fast operation in embedded systems without the help of an operating system. Several Java processors have been developed [1]–[21], which will be briefly surveyed in this section.

Except for the FemtoJava, all Java processors are initially designed to completely support all Java bytecodes. Thus, we will analyze the support of several JVM concepts and their implementation, particularly with regard to the support of multiple, real-time threads. An overview of the properties and the features of selected Java processors is given in Tab. I. Besides them, there are several other processors, however, with only limited information available, which prohibits their inclusion in the table: the reconfigurable simple Java core (R-Java) [13], the asynchronous Java processor [14], the VLIW Java processor [15], the JA108 Java coprocessor [16], the Jazelle extension of the ARM processor [17], [18], Lightfoot [19] and AMIDAR [20], [21].
TABLE I

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline stages</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
</tr>
<tr>
<td>Instruction folding</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Microcode</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Software Traps</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Stack Realization</td>
<td>e</td>
<td>r</td>
<td>i</td>
<td>r</td>
<td>i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Objects</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Multiple Threads</td>
<td>-</td>
<td>-</td>
<td>(+)</td>
<td>(+)</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Thread HW-Support</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Scheduler</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Synchronization</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(+)</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Real-time Threads</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Garbage Collector</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>H</td>
</tr>
<tr>
<td>GC for Real-time Threads</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Dynamic Class Loading</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

H Hardware, M Microcode, S Software, + supported, - not supported, r Internal Mem., e External Mem., i Register

a statically allocated threads
b single global monitor

typically, the design of a Java processor follows that of a RISC processor. Pipelined bytecode execution is common, which leads to the well-known conflict between high clock frequencies (throughput) and the execution latency of branch instructions. This problem may be solved with branch prediction, which, however, requires additional chip area and energy and, thus, conflicts with the target application domain in embedded systems.

As Java bytecode applies a stack-based execution model, the stack implementation of a design is a performance-critical issue. So the stack is typically mapped onto a fast internal memory or even a register file. An exception to this rule is, again, FemtoJava, which locates the stack in external memory while mirroring the two topmost stack entries in registers.

The mapping of the stack on a register file requires the technique of instruction folding [23]–[26], which assimilates instructions for stack data movement, like the loading of local variables, with the actual computational operation. This yields a typical RISC instruction with direct operand addressing and may accelerate the execution of Java programs as it eliminates the unproductive cycles for data movement. The cost is additional chip area required for the implementation of the read/write ports of the register file. To bound the required chip area, only a limited number of registers is available, and, thus, an automatic stack spill and fill must be implemented (called: stack cache). Instruction folding is also available for stacks mapped onto internal memory but its effect is known to be much smaller, cf. [2].

Besides the direct implementation of the Java bytecodes by state machines, it is also common to rely on the help of microcode (firmware) as well as software traps, which themselves are backed by Java programs. This simplifies the implementation of complex bytecodes such as invokevirtual tremendously and has a positive effect on the maintainability. Because this approach requires additional memory, there is no general statement if it requires more or less chip area than an implementation with complex state machines only.

In addition to the Java bytecodes defined by the Java Virtual Machine Specification [27], the Java processors implement special bytecodes for the access to the runtime system and internal data structures, a task that is performed by native methods in software JVMs.

Java is an object-oriented platform storing all data except for primitive data types inside objects located on the Java heap. This is a memory area usually managed by a garbage collector (GC), which frees the memory occupied by unreferenced, i.e. unneeded, objects. The GC can be implemented in either software or hardware and is typically responsible for the memory used by regular (non-real-time) threads.

Multi-threading is another essential JVM concept that allows the parallel execution of multiple tasks. Usually, regular threads are distinguished from real-time threads, where real-time means that guarantees about the execution and answer time of a task can be given. Scheduling can either be done in hardware, via microcode or with software. For a fast thread switch needed for short response times, hardware assistance for context saving and loading is required. Last but not least, controlled access to objects used by multiple threads requires a synchronization mechanism. Such is commonly available on a per object basis. An exception is JOP, which provides only a single global monitor. Furthermore, this Java processor as well as Komodo support only a statically allocated number of threads.

The support for real-time threads by the currently available Java processors is limited. Particularly, there is no automatic garbage collection in the memory areas used by real-time threads. Instead, the approach defined by the Real-Time Specification for JAVA (RTSJ) [28] is used. It defines designated heap areas, typically one per real-time thread, with possibly different properties. Real-time threads then allocate their objects exclusively from their heap area without any garbage collector inference. These heap areas can only be destroyed as a whole as triggered manually by the programmer. Another approach is the Ravenscar-Java profile [29]– a subset of the RTSJ. Here, allocation of objects is only allowed in an initialization phase. On the other hand, some research results are available, which show that automatic garbage
Neither is dynamic class loading at run time standard. Rather, the whole application is pre-linked into a memory image, which enables fast execution but inhibits dynamic class loading afterwards. This is caused by the layout of the memory image, which would require new information to be inserted rather than appended. The linking step by JOP [7] is one example. To provide dynamic class loading, the information must be stored in objects on a per class basis instead. The JEM2 processor [12] takes another approach by managing two parallel Java Virtual Machines (JVMs). Here, each JVM uses a pre-linked memory image. It is not stated clearly whether these images can be replaced at run time.

The aspects of automatic GC for real-time threads and dynamic class loading at run time are covered in particular by our project in addition to providing a general-purpose embedded Java processor for secure, real-time and multi-threaded applications.

III. OVERVIEW

The SHAP concept spans not only the SHAP microarchitecture but also all the software components required to execute Java applications in hardware under real-time constraints. All components of the microarchitecture are required to perform their tasks in constant time to achieve this main goal.

The SHAP microarchitecture is divided into the five components depicted in Fig. 1. It consists of:

The CPU: which directly executes Java bytecode and controls all other parts. It contains the fetch unit, decoder, arithmetic/logic unit as well as the on-chip stack module.

The Memory Manager: which handles the main memory, i.e. the Java heap. The heap stores the Java objects as well as the class information (class objects). The memory manager is equipped with a GC, which autonomously cleans up the heap without burdening the CPU.

The Method Cache: which caches the currently executed Java method, which are regularly stored on the heap (inside the class objects).

An Integrated Memory Controller: which provides a direct interface to external memory, like SRAM oder DDR-SDRAM, and, thus, does not incur additional latencies due to external protocols.

An Integrated Devices Bus: attaching further devices to interface to the outside world and to provide internal secondary communication. This bus is mastered by the CPU.

A more detailed description is given in the following paragraphs. The currently available software components are:

- An implementation of the “Connected Limited Device Configuration” (CLDC) API [33]—a subset of the standard Java API especially designed for embedded devices.
- The SHAP linker, which preprocesses and links the input class files into a SHAP file ready for execution on the SHAP microarchitecture. The SHAP file is not a flat memory image, it rather contains a designated section with the information for the construction of a separate runtime class object for each class. This lays the foundation for dynamic class loading.
- An assembler for the microcode used internally by the core implementation.

IV. INSTRUCTION SET ARCHITECTURE

The instruction set architecture of SHAP is strongly related to Java bytecode. There are a few essential differences, limitations and extensions:

- All branch instructions use absolute target instruction offsets from the start of the method. This does not impose a tighter bound on the allowable size of method code blocks than specified in the current JVM specification [27, §4.10] but it eliminates the need for the GOTO_W instruction.
The SHAP Microarchitecture

V. DESIGN

The design of SHAP follows a strict modular approach. The interfaces of the individual components are on a high logical level as to achieve a high degree of encapsulation of a component’s responsibility and its implementation. The autonomous handling of their responsibility by the components enables a high degree of parallelism and frees the central computing core from continuous burdens as the management of the stack and the heap.

TABLE II
NEW JAVA BYTECODES INTRODUCED BY SHAP.

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>cb</td>
<td>(color bind) Associate a reference with an interface method table (IMT).</td>
</tr>
<tr>
<td>cb</td>
<td>(color change) Change the association to another IMT.</td>
</tr>
<tr>
<td>wait_enter</td>
<td>Release an object’s monitor and yield thread to wait for a notify on the object.</td>
</tr>
<tr>
<td>wait_wait</td>
<td>After the scheduler activates the thread, check if object was notified. If not, yield the thread again.</td>
</tr>
<tr>
<td>wait_leave</td>
<td>Re-acquire monitor after object has notified.</td>
</tr>
<tr>
<td>notify</td>
<td>Notify an object.</td>
</tr>
<tr>
<td>get_time</td>
<td>Returns the current system time.</td>
</tr>
<tr>
<td>arraycopy</td>
<td>Copies an array.</td>
</tr>
<tr>
<td>set_class</td>
<td>Registers a new class object.</td>
</tr>
<tr>
<td>get_class</td>
<td>Returns the class object of a specified object.</td>
</tr>
<tr>
<td>thread_current</td>
<td>Returns the active thread.</td>
</tr>
<tr>
<td>thread_start</td>
<td>Starts a new thread.</td>
</tr>
<tr>
<td>thread_yield</td>
<td>Yields the current thread.</td>
</tr>
<tr>
<td>thread_finish</td>
<td>Finish the current thread.</td>
</tr>
<tr>
<td>io_read</td>
<td>Read from I/O device. Yield thread if device is busy.</td>
</tr>
<tr>
<td>io_write</td>
<td>Write to I/O device. Yield thread if device is busy.</td>
</tr>
</tbody>
</table>

- Extra bytecodes for system interfacing have been introduced taking responsibilities of typically native code of software JVMs.
- Extra bytecodes for interface type coercion.
- The LDC, LDC_W and LDC2_W group of instructions is substituted by a single LDC for 32-bit words (integers, pointer to data structures) and an LDS for loading string references.

Due to the lacking support for floating point arithmetic and long 64-bit data types, the corresponding bytecode instructions are missing. It should also be noted that instructions with constant pool references work on a constant pool readily resolved by the SHAP linker. The basic principle of a stack-based execution is kept as it is known, so that their execution time cannot be included. To invokeinterface. The called method may be unavailable information, such as the size of a method or the number of exception table entries. This requirement enables the calculation of the execution time of a method and consequently for a complete application. Their is one exception with invokevirtual and invokeinterface. The called method may be unknown, so that their execution time cannot be included. Special care has to be taken by the programmer in this case.

A. Core

1) Features: The core directly executes Java bytecode with the differences given in Sec.IV. As defined by the JVM Specification [27], several constraints have to be checked before the execution of a Java bytecode. Static constraints, such as the appropriate types of stack operands, are verified at link time by a bytecode verifier. Thus, the core must only perform truly dynamic checks such as testing for null references. The handling of exceptions whether so raised by the system or by user code is fully supported.

Thread scheduling is implemented in microcode and assisted by the multi-context capability of the stack, which is described there. For the scheduler, various techniques can be considered, of which we chose a preemptive round-robin scheduling to distribute the execution time fairly. Blocking accesses to devices on the integrated devices bus are exploited by the scheduler, which suspends the blocking thread’s execution for high core utilization in favor of the next in line. Finally, also monitor synchronization is implemented in microcode where monitors are associated with object instances including the instances of class objects for the synchronization of static code blocks.

The theory behind the rather complex issue of the handling of interface and the efficient implementation of the INVOKEINTERFACE bytecode is covered in a separate paper. Appropriate support is provided, and the necessary data structures inside the JVM are described in Sec.VI.

The main demand on the core is that all Java bytecodes are either executed in constant time or that their execution time is known in advance based on statically available information, such as the size of a method and the number of exception table entries. This requirement enables the calculation of the execution time of a method and consequently for a complete application. Their is one exception with invokevirtual and invokeinterface. The called method may be unknown, so that their execution time cannot be included. Special care has to be taken by the programmer in this case.

The core also provides an interface to the GC as to enable its scanning of the stack for alive references. All references on the stack are marked with an additional 33rd bit set so that the stack module actually handles 33-bit instead of 32-bit data.

2) Implementation: The core is comprised of the instruction fetcher, the decoder and the arithmetic/logic unit and is directly connected to the stack module. Both components, core and stack, form the CPU, which implements the JVM. For the access of objects stored on the heap, the core addresses the memory manager, which performs the requested operation in parallel. Details are described in Sec.VC.
Java bytecodes are not directly decoded by the core but used as an index to jump to a microcode subroutine, which will execute the requested Java bytecode. This subroutine can be as short as a single instruction executing simple Java bytecodes in a single cycle. Due to the pipeline described below, no overhead in time is incurred by this approach. The use of microcode simplifies the development and decreases the complexity of the system, and, thus, reduces the probability of design errors.

During the study of other Java processors (cf. Sec. II), we saw that instruction folding has no significant gains and, thus, decided against using this technique.

The instructions of the microcode subroutines, further called just instructions, constitute the internal 9-bit instruction set of the core. A list of these instructions together with a short description is given in Tab. V in the appendix. These instructions are executed in a 3-stage pipeline: fetch, decode and execute. An additional fourth stage, bytecode fetch, is prepended to fetch the Java bytecode from the method cache.

The subroutine for one Java bytecode might be many instructions long so that the fetch stage either loads the first instruction of a microcode subroutine, according to the Java bytecode, or the next one in sequence. The subroutines are not separated in memory, so that they can re-use common code sequences. Also branch instructions are supported but they must be used with care to fulfill the real-time demand.

The decode stage decodes the instruction into control signals sent to all components including the stack. They can start operations, which do not rely on the top of stack value, before the operation is completed in the following execute stage, during which also the stack is updated.

The arithmetic/logic unit is 32-bit wide and completes all operations in a single cycle. No forward logic is required, which would consume energy and chip area.

The instructions utilize an internal microcode data memory with 128 32-bit entries storing (a) 16 variables for scratch as well as JVM status information as references to the currently executed method and to its defining class for constant pool access, (b) 48 constants as an alternative for instruction immediates, and (c) 64 entries (target and type) used by branch instructions. This memory may also store references to objects and is, thus, equipped with an interface for scanning by the GC. Of course, the GC has to scan the first 16 entries only.

For all unimplemented bytecodes and for throwing exceptions in case of the violation of a runtime constraint, the core can call software traps, which are themselves implemented in Java bytecode. This way, also the support for long and floating-point arithmetic may be easily added, which is currently lacking in hardware.

The preemptive round-robin scheduler is implemented in microcode. The status information of threads is stored inside the corresponding Java object. Thread switching is performed with the help of the stack module, which provides a separate stack for each thread. The scheduler, explained in detail in Sec. V-F, is triggered periodically by a timer and starts execution at Java bytecode boundaries only. Thus, Java bytecodes are always executed atomically.

Nearly all microcode instructions execute within one cycle. Exceptions are some stack operations which handle frames and some operations of the memory manager, which are, however, executed in parallel. The requirement here is that the corresponding modules execute their operations in constant time and the lowest number of cycles possible. Despite this, the execution time of a microcode subroutine is known, and, thus, the real-time requirement is fulfilled. There is a special note on the method cache, which is described in Sec. V-D.

Two Java bytecodes include a loop in their microcode subroutine. A lookup switch searches a table for the branch target in dependence of the topmost stack value. As the length of the table is known after linking, the worst-case execution time, can be calculated. The other bytecode is athrow, which scans the exception table and may also have to unroll the stack to calling methods. Although the size of the exception table is known, too, the caller stack might be unpredictable. Special care has to be taken so that exceptions either do not occur or are caught at well-known points.

B. Stack

1) Features: The stack is implemented in a designated component using on-chip memory for the storage of application and internal management data. The current FPGA implementation utilizes the block RAM components widely available on modern chips.

The stack component provides high-level stack operations executed in constant time to the core. Moreover, the operations frequently required by the JVM are implemented such that they are shadowed totally by the regular execution of the core pipeline and are guaranteed to never slow the execution by requiring stalls. These operations are, in particular: push, pop, ld_var, st_var and ld_frame. A short description of their and the other operation’s function can be found in Tab. III. For fast access by the core, the two topmost stack values are stored inside the registers (top-of-stack) and NOS (next-on-stack).

Besides these rather standard stack operations, also the method frame management is performed autonomously through the operations enter and leave. As an exception, enter requires two cycles in total, but still runs in constant time. The leave operation imposes a few restrictions on the stack operations requested in the immediately following cycle as the reconstruction of
the microcode will be able to use this slot sensibly. Of no practical relevance as a simple reordering within frame variables is forbidden. This restriction is usually particularly an immediately following access to local and the previous method frame is only completed then. In

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>Pushes a word onto the top of the stack.</td>
</tr>
<tr>
<td>POP</td>
<td>Pops a word from the top of the stack.</td>
</tr>
<tr>
<td>RD_VAR</td>
<td>Loads a local variable (application data) from the current stack frame onto the top of the stack.</td>
</tr>
<tr>
<td>ST_VAR</td>
<td>Stores the top of the stack into a local variable of the current stack frame.</td>
</tr>
<tr>
<td>RD_FRAME</td>
<td>Loads a frame variable (JVM data) from the current stack frame onto the top of the stack.</td>
</tr>
<tr>
<td>RD_BW</td>
<td>Use top of stack to index this many positions down into the stack and replace it by the value found there.</td>
</tr>
</tbody>
</table>

ENTER: Establish new method frame with the number of arguments and local variables just pushed.
LEAVE: Destroy current method frame and activate preceding one.
SWITCH: Activate stack of the (possibly new) thread specified by the top of stack.
KILL: Destroy stack of the thread specified by the top of stack.

Fig. 2. Exemplary Stack Block Organization

the previous method frame is only completed then. In particular an immediately following access to local and frame variables is forbidden. This restriction is usually of no practical relevance as a simple reordering within the microcode will be able to use this slot sensibly.

Java method invocation is further supported by the operation RD_BW, which allows to copy a value located further down in the stack onto its top. Such an operation is necessary to retrieve the this argument of a method invocation to resolve interface and virtual method calls by its runtime class type.

The stack component also takes care of the stack substitution due to thread switching. Although thread switching by SWITCH occupies the stack component for at most 4 cycles, it is still performed in constant time. The destruction of a thread’s stack that finishes normally takes 2 cycles, a forceful destruction of a large stack space may take longer.

2) Implementation Overview: The storage space used by the stack module is divided into equally-sized blocks currently holding up to 64 words. These blocks are organized in multiple disjunct singly-linked lists. Each list, except for the list of unused blocks, represents the stack of one thread. Active thread stacks are linked backwards so that the topmost block is the root of the list. The state maintained by the stack module for the basic management of these lists is limited to the index of the first block of the free list FB and to the stack pointer SP identifying the block and the internal offset of the top of stack of the currently active thread. The heads of the stacks of inactive threads hold management information internal to the stack module and are identified by a handle passed to and returned by the SWITCH operation. These handles are managed by the runtime system typically as part of the state of the Thread objects. An exemplary situation is depicted in Fig. 2. Also, the creation of a new thread stack is achieved through the SWITCH operation by passing it the special handle $-1$.

The organization of the stack blocks in linked lists enables a fast dynamic growing and shrinking of the active stack by relinking a block between the free block list and the list of the active stack. As the occupied stacks are linked backwards, this only requires fast manipulations at the heads of the lists.

The constant execution time of block relinking can, however, only be guaranteed when the number of blocks to be relinked is bound. In our case, we restrict ourselves to a single block whose relinking can be performed within one cycle. Java, unlike C/C++, defines all objects to be allocated on the heap through the new operator and has no stack allocation model for larger objects. Thus, having only small primitive and reference values on the stack, this is not a severe limitation. It, however, somewhat limits the extend of a method frame, which must be restricted to a block size with all its local variables (including arguments) and remaining operand stack contents, at least, on method exit. The currently available space of 64 words seems, however, to be sufficient for just about all practically relevant applications, cf. [34], [35].

There are three immediate approaches yet to push forward this bound: (a) increase the block size, (b) accept possibly multi-cycle method frame operations, (c) box local variables and/or arguments in a synthetic heap object. Approach (a) has the drawback of decreasing the granularity of the distribution of the available memory space so that more of it might become unnecessarily unusable in blocks with low utilization. Although (b) would be potentially slower than (a) for methods with

1 A JIT compiler might, of course, be able to identify an object as only being used locally and to move its allocation to the stack, which is tremendously simpler and faster and reduces the pressure on the garbage collection. We did not take this approach.
larger frames, its time requirement is still fully deterministic only now depending on the space requirements of the particular method invoked or left. Approach (c), finally, would not require any modifications to the stack component but would definitely be the slowest as it is a software-based solution.

Concurrent reading access to the stack data is granted to the GC module to enable its scanning for live object references. This access is enabled by the utilization of the free bandwidth of the write port to the internal stack memory making it a read-write port. While, the GC would normally only scan the part of the stack really used by the threads, this would require that the GC started scanning at the top of each thread’s stack to its bottom according to the block linking. Since the top of a stack is directly known only for the active thread, this information would have to be retrieved from the Thread objects on the heap for all others. We did not take this approach. Instead, we mark each block whether it is used or not. This extra bit is stored along with the linking information. Now, the GC scans the blocks in linear order skipping all unused blocks. The disadvantage is, that the GC might encounter actually dead references, which are stored in the unused upper parts of partially utilized blocks.

3) Implementation Detail: The external interface of the stack module is depicted in Fig. 3. It comprises the standard control signals, Reset, Clock and Enable, a set of command signals specifying the desired stack operation, a data interface, a state messaging and a wider connection to the garbage collection module, which will be discussed there.

The state messaging consists of only two signals. The Ready signal marks the end of the few multi-cycle thread operations and is also raised during system startup when the initialization of the stack module is finished. This initialization is dominated by the linking of the list of free memory blocks. The other signal Fail implies a stack underflow or a stack overflow depending on the requested operation. While a stack underflow implies an incorrect usage pattern like the execution of a POP operation on an already empty stack, the stack overflow signifies an exhaustion of the available memory space, i.e. the growing stack must be extended by a new block while no free one is available.

The data interface provides permanent access to the two topmost stack elements called top of stack (TOS) and next on stack (NOS), respectively. For performance reasons and simple access, these stack positions are registered. In particular, they are not even mirrored into the memory so that the stack pointer SP actually points at the stack element below NOS during regular operation. Since the stack might have fewer than these two elements, these registers are extended by a valid flag indicating whether or not they contain actual data.

The behavior of the standard stack operations PUSH and POP is depicted in Fig. 4. While the PUSH operation, indeed, only pushes the new value received via DataIn onto the stack, the POP operation is somewhat more sophisticated than its name suggests. In particular, the TOS position is not filled with the previous NOS value but with the value received on the DataIn channel. A classic POP is, thus, achieved by forwarding the value from the NOS output to the DataIn input. Additionally, operations taking two operands from the stack and producing one result on the stack can be easily shortcut by the core using this stack design. As their net effect is the reduction of the stack by one value, they can be executed as stack POP operations joint with a TOS substitution. Recall that both required operands are available through the TOS and NOS outputs. This enables the fast execution of frequent arithmetic operations and thereby allows the mapping of any such Java bytecodes to a single SHAP microcode instruction.

Quite a few pointers to stack memory are maintained within the stack module. While full pointers are composed of a block number and a block-internal offset, block linking uses block numbers only. Pointers that are allowed to mark the end of the list are prepended an additional NIL bit that conceptually belongs to the block identification. As we currently provide the stack module with 64 memory block with 64 data words each, both the block number as well as the internal offset occupy 6 bits each.

Next to the SP, there are two more full pointers that establish the current method frame. While the variable pointer VP identifies the beginning of the local variable area started by the method arguments, the frame pointer FP points directly behind it at the frame word generated by the stack module for the previous frame to be restored.
upon \textsc{Leave}. A memory access relative to these pointers enables the fast access to local variables or frame data, respectively. The memory is bypassed in the case that the requested data lies in one of the two topmost registered stack values.

The method frame as established by \textsc{Enter} is depicted in Fig. 5. The targets of the pointers \textsc{VP} and \textsc{FP} are identified. Note that two stack values following the argument list are transferred from the caller’s into the callee’s stack frame. These two values are utilized by the microcode subroutine for method invocation. Avoiding to equip the \textsc{Enter} operation with an immediate operand, it retrieves the argument and local variable count from TOS. The cost of this decision is that the \textsc{Enter} instruction so must restore one of the JVM frame words from memory. As two distinct ports for reading and writing are used, this is shadowed by storing the stack-internal frame word to memory. Note that no further data movement within stack memory is required. The arguments passed stay precisely at the location where they are. Merely the new \textsc{VP} is calculated from the original \textsc{SP} and the argument count to point at the first of them. The additional locals do not need to be initialized. Only the space required for them is reserved.

The \textsc{Leave} instruction destructs the current method frame and reactivates its predecessor as illustrated in Fig. 6. The information necessary to restore the previous variable and frame pointers is obtained from the frame word generated by \textsc{Enter} and identified by the current frame pointer. All the stack slots up to and including the first argument or local slot are released by having the \textsc{SP} point one position below the current \textsc{VP}. The \textsc{Nos} and \textsc{Tos} values remain unchanged enabling the transfer of a return value into the frame of the caller returned to. The transfer of two stack slots is sufficient for all return values supported by Java, the widest ones being \texttt{long} and \texttt{double} values occupying two slots. The reconstruction of the JVM state not directly related to the stack must be performed by its runtime implementation. It can query the corresponding frame words it stored relatively to the frame pointers before the \textsc{Leave} operation and it may have to \textsc{Pop} the transferred values if they do not constitute valid return values.

The dynamic extension and shrinking of the active thread is handled transparently inside the stack module. This somewhat complicates the implementation of the individual stack operations. In general, any calculation of an offset to a pointer into the stack data area is invalid if it produces an underflow or an overflow into the block number. Thus, the recalculation of the stack pointer \textsc{Sp} may substitute the block number from the root of the free block list \textsc{FB} or from the link memory whenever the boundaries of the current block are crossed. The corresponding update of \textsc{FB} and the link memory is performed concurrently. The restriction to relink a single memory block per operation guarantees a constant execution time per operation, mostly even within one cycle. The only exception to this rule is the forceful destruction of a thread’s stack by \textsc{Kill}.

As a thread’s stack is generally not a contiguous memory region, also value accesses relative to the variable or frame pointers may require the substitution of the block number when block boundaries are crossed. As these require the next memory block while the previous one is linked through link memory, its block number must be obtained from somewhere else. In principle, the block number identified by the current stack pointer can be taken as a method frame is not allowed to span more than one block size anyhow. This is, however, only easily guaranteed at bytecode boundaries. As to achieve a greater freedom in the microcode implementation, we rather use a designated \textsc{Nb} (Next Block) register to identify the memory block following the one \textsc{VP} points into – if this block exists. \textsc{Enter} and \textsc{Leave} operations also save and restore this 6-bit register together with \textsc{VP} and \textsc{FP} inside the frame word.

The most complex stack operations are \textsc{Switch} and \textsc{Kill}. Both expect a handle to a thread stack on the top
of the currently active thread. This handle previously generated by the stack module is essentially an inactive stack pointer into the stack memory. An empty stack is represented by the special handle $-1$. This choice minimizes the exceptional handling of such a handle as it points at the last word of a non-existent block as its offset and block number have all bits set including the end-of-list bit for the latter. In contrast to the active stack, non-empty inactive stacks have all their values stored within memory. On top of the actual stack content, an additional frame word as also constructed by \texttt{ENTER} stores the necessary information to revive the topmost method frame of the stack. Any further context information of the JVM implementation is saved prior to the \texttt{SWITCH} invocation and restored thereafter as needed. From the perspective of the stack module, this is regular stack content.

The rather long execution time of the \texttt{SWITCH} operation of 4 cycles is due to the many memory accesses needed in its implementation. If valid, it must flush the \texttt{NOS} value to data memory and store the frame word behind it. It then internally exchanges the active stack pointer \texttt{SP} with the \texttt{TOS} content. If the new \texttt{SP} identifies a non-empty stack, its topmost method frame and the \texttt{NOS} register are restored from memory. This completes the context switch for the stack module. Note that the handle to the previously active stack, which is to be returned, has been moved to the top of stack by the stack pointer exchange already.

The \texttt{KILL} operation, finally, is the only operation, which may take an indefinite amount of time as it unwinds the stack identified by the passed handle block by block to relink it into the free block list. Such an expensive rewinding is, however, an exceptional case as a Java Thread usually ceases execution by returning from its \texttt{run()} method so that a single method frame and thus only a single block belong to the stack to be destructed. In this case the \texttt{KILL} operation finishes after 2 cycles. Otherwise an extra cycle is required for each additional block to relink.

---

**TABLE IV**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{NEWREF}</td>
<td>Allocate a new object on the heap.</td>
</tr>
<tr>
<td>\texttt{STREF}</td>
<td>Activate the provided object reference for use in the following calls.</td>
</tr>
<tr>
<td>\texttt{STOFS}</td>
<td>Register offset into the active object and read the value at this offset.</td>
</tr>
<tr>
<td>\texttt{STVAL}</td>
<td>Store the provided value into the active object at the offset registered by a previous \texttt{STOFS}.</td>
</tr>
</tbody>
</table>

---

**C. Memory Manager**

1) \textit{Features and Requirements}: The memory manager administrates the Java heap by allocating objects, performing read and write operations on them and, finally, by freeing memory used by unreferenced objects. This module encapsulates the complete object management. The central core only acts upon references identifying objects and applies offsets into these objects to store and retrieve data.

The memory manager provides the small set of operations listed in Tab. IV. All of them are executed concurrently to the CPU operation. Since all outgoing data is stored in an output register that is permanently accessible to the CPU, no operation explicitly providing a reading access needs to be implemented inside the memory manager although, of course, the microarchitecture knows microcode instructions transferring the contents of this register onto the core stack. All operations of the memory manager are executed in constant time but may take several cycles. This may incur extra wait cycles in the pipelined execution of the core but their worst-case number is known in advance. Thus, execution under real-time constraints is still available.

2) \textit{Implementation}: The implementation of the memory manager is mainly defined by the GC technique to be used. The main task of the GC is to scan the stack and the heap for referenced objects. Its access to these memory components must be interleaved with the normal execution of the mutator, i.e. the central core, and also requires a few atomic primitives for the modification...
of administrative data structures.

If the garbage collection is not to jeopardize the real-time properties of the processor system, it must allow the mutator execution to be performed in predictable time. A trivial approach to meet this condition is the employment of a stop-the-world GC implemented as another real-time thread, which performs a complete heap scan within its assigned time slot. The worst-case execution time of such a GC can be calculated in dependence of the heap size. Although this yields a fixed bound, it would typically be prohibitively large requiring a very long scheduling period. The resulting guaranteed response time of the system would be unacceptable for many applications.

A solution to this problem is either an incremental or a concurrent GC approach. As to allow the parallel execution of the GC to all Java threads, we decided to follow trace of Wise et al. [36] (and preceding pioneers especially from the LISP community) and actually implemented a concurrent mark-and-sweep collector directly in hardware. Its direct integration into the memory manager minimizes the distance to the memory for fastest possible access.

As listed in Fig. 7, the memory manager consists of three main components: the reference manager (refman), the segment manager (segman) and the garbage collector (gc). Further, the actual access to the main memory is performed through the memory access manager (memaccman), which also provides basic atomic primitives needed for memory management. Only the reference manager and the segment manager utilize the access manager directly. The GC merely relies on some high-level operations provided by the other components.

In order to allow the easy movement of objects in memory without requiring the update of all pointers, we decided to make the references used by the computing core indirections. They are only resolved inter-nally within the memory manager through a fixed-size reference table. Each of its entries contains the current address of the associated object within the physical memory, the size of this object and a few status bits. The reference items are additionally organized within linked lists. For example, all the free entries (without a valid pointer into physical memory) form the free list. When performing an allocation, the reference manager removes a free reference item, assigns a contiguous free region of physical memory to it and stores the reference identifier, which essentially is its number within the reference table, into the register accessible to the CPU. To simplify the GC, the reference manager also provides some high-level operations such as scanning an object, moving it to another region in memory, marking it garbage and disposing of it.

The Java heap is divided into several equally-sized segments, inside which objects are placed. As shown in Fig. 8, the segment manager sets up all the corresponding data structures in a doubly-linked list during initialization. The free pointer denotes the head of the free segments. All other free segments are located left of it. All allocated segments, i.e. segments containing allocated memory, are situated to the right of the free pointer. One of these segments is the selected allocation segment, where all allocation of new objects takes place through a monotonic fast constant-time bump-pointer allocation. If the space left within a segment becomes smaller than the maximum object size after an allocation, the next free segment will substitute the current allocation segment. This exchange is performed transparently and independent of the core operation - typically concurrently to the initialization of the object just allocated. Whenever the allocation segment is replaced or the GC requests a new target segment, the free pointer is moved one position to the left. Note that the first allocated segment is not automatically the allocation segment because also the
GC can request a free segment as target for object movement.

For each segment, a list is maintained that links all the reference items whose associated objects are currently contained within it. The root of this list is stored in the administrative segment descriptor, which also holds useful segment occupation statistics such as the number of allocated data words within the segment and the number of dead data words, i.e. those that are occupied by objects identified as unreachable by the GC. The latter is supported by the segment manager through the provision of operations for requesting a new target segment, freeing a segment and accessing and manipulating the segment descriptor. When the GC requests the freeing of a segment, it is simply relinked as the leftmost element of the doubly-linked segment list moving it also to the left of the free pointer.

The process of garbage collection starts with the gathering of the roots of the graph of the reachable objects by scanning the stack and the microcode variables of the core. The transitive closure of all objects reachable from this root set is then constructed through a heap scan. While an extra bit could be provided for the identification of references on the stack and within the microcode variables, this approach was not feasible for the heap as the used memory only stores 32-bit words. Here, the identification of references is based on a signature inside the otherwise unused topmost bits of the references. The disadvantage of this approach is that the safe identification of reachable references requires a conservative classification of the encountered values so that certain numeric values will be falsely identified as references. Although the choice of this signature outside the heavily-used low-magnitude numerical range reduces the number of misinterpretations, this issue is worth to be solved in the future by a classification directed by the class layout rather than by the encountered values (cf. [37]).

The scanning of the stack, the microcode variables and the heap is performed concurrently to the regular execution of the core. This execution must be monitored by the garbage collector in order not to miss a live reference due to some race condition. As the architecture of the core requires all data transfer to go through the stack, this monitoring can be restricted to the topmost stack value whose situation within a register is a great simplification. Whenever a reference appears there in the process of the scan, it is conservatively marked alive by the GC.

Having identified all live references, a set of segments not containing the allocation segment is selected for cleaning. This selection must be done in an intelligent manner, and further research is required on this issue. The goal is to choose segments with an appreciable gain in free memory. They will then be emptied by removing all referenced objects to another target segment. Subsequently, the whole memory space of the segments is free so that they can be relinked as free segments left to the free pointer within the segment list. Note that all the GC operations are interruptible in order to enable a timely access of the core to the memory permanently. Nonetheless, it must be considered that the available memory bandwidth is shared among the object accesses of the core and the integrated garbage collection.

Although the implemented integrated garbage collection is fast and truly concurrent, it cannot guarantee safe real-time performance in all cases. Applications exhausting the available memory resources or depending on an instant collection of large data structures for immediate re-allocation will fail. While the problem of
resource exhaustion can be contained to a single task or even a thread by quota accounting, the allocation characteristic of an application is a problem hard to grasp. The statistical evaluation of some typical Java applications is available in the literature [31], [32].

We also implemented a minimal baseline version of the memory manager, which only implements the very simple bump-pointer allocation within the flat memory without ever reclaiming unused space. While this variant is certainly not suitable for field application, it allows to determine the overhead of the garbage collection in terms of additional memory access latency and chip area. A few figures are presented below at the end of this section.

D. Method Cache

1) Requirements: To execute the microcode subroutines in constant time, no pipeline stall may be introduced due to the fetching of Java bytecode. This is typically achieved, by a Harvard architecture, which provides separate memories and buses for transmitting code and data. The code memory itself must provide a high throughput so that one Java bytecode can be fetched each cycle. Also, the latency should be low for fast execution of branch instructions.

On our current prototyping platform, there is only one external memory. This restricts us to a von-Neumann architecture. Due to the concurrent data and code accesses, the bandwidth of the memory is too small, especially in the case of concurrent garbage collection. To circumvent this problem, we use the concept of a method cache as it can be found in JOP [7]. Our method cache stores only the current method to execute and achieves real-time properties.

2) Implementation: As our method cache only stores one single method, no tag memory and comparators typically found in caches are required. The cache is explicitly filled during the switching to another method by the invoke* and *return bytecodes. No data access is required during this caching, and the GC is paused so that the full memory bandwidth is available for caching. Thus, the execution time of the indicated bytecodes depends on the size of the method to call or to return to. Because the size of a method is known in advance, the execution time of these bytecodes is known in advance so that the real-time requirement can be fulfilled. With using of invokevirtual and invokevirtual, the called method be unknown, so that the caching time is unknown too. But, in this case, also the normal execution time of the called method is unknown. Thus, their is no new uncertainty.

This method cache, however, has also a big disadvantage. During thread-switching, the microcode switches to another method, which must be cached accordingly. Because it is not known, which method is currently active in the new thread, the caching time is only limited by the maximum size of a method, which equals to the size of the cache memory. Consequently, there is a large worst-case delay every time the scheduler is called.

One solution to this problem, is to enlarge the method cache, so that the current method of each thread can be stored. This approach omits the extra delay during thread-switching but requires extra on-chip memory. Only a low size of this memory would be acceptable, resulting in a low number of supported threads.

Due to this disadvantage, we are currently switching to another prototyping platform with a Harvard architecture. Class objects will then be split up, so that an application consumes the same amount of memory as before.

E. Integrated Devices Bus

The integrated devices bus connects the SHAP core with its secondary components. Many of these implement the communication with the outside world. Others realize a secondary interface to internal components as the statistics port to the memory manager. The bus is mastered and arbitrated exclusively by the CPU. All connected devices are slaves.

The devices bus supports full 32-bit wide data and addresses. While the upper part of the address selects the targeted device, the lower bits may be evaluated by the device to distinguish several ports or commands. The data bus has designated lines for both directions toward and from the devices as there is no support for on-chip tristatable logic on the targeted FPGA platform.

Every device further supplies the core with two status signals as applicable. It signals ready when it is able to receive data from the core, and it asserts available when data is available for reading from the selected port.

The CPU does not handle all bus devices directly. It rather interfaces to a single set of address, data and status lines. The activation of a device to drive these lines is performed through the device selection based on the supplied address.

The currently implemented range of devices includes a serial interface (RS232), a PS/2 keyboard controller, an LCD controller as well as the statistics interface to the memory manager. Due to the simple bus interface, the addition of further devices is straightforward.

F. Scheduler

SHAP’s thread scheduler is currently implemented in microcode and is regularly triggered by the timer interrupt. To enable an efficient integration with scheduling-related API functionality, it is further utilized by the closely-related implementations of the conventional blocking java.io package, of the monitorenter
and monitorexit instructions and some extended system bytecodes that are used in the implementations of methods like Thread.yield(), Thread.sleep() and Thread.start().

The scheduler maintains a single central variable in the JVM state that contains a reference to the Thread object of the currently running thread. This serves as the root to a doubly-linked list maintaining all active Thread instances. A newly-created Thread object is added to this list by the invocation of its start() method, more precise by the new bytecode thread_start; and it is removed after ceasing execution of its run() method by the also new bytecode thread_finish. The list is linked through two private fields of the Thread object not available to the public API. A further hidden instance field managed by the scheduler holds the stack handle while the Thread is not the currently executed. Finally, the state of a Thread instance, which is one of NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING or TERMINATED, is maintained in close cooperation with the Java portion of the Thread implementation.

The implemented scheduling strategy is a flat round-robin scheme. Prioritization of Threads is not yet implemented. The currently executed Thread is preempted when its assigned time slice has ended. While a Thread cannot defer its preemption, it can prematurely yield to its successor in the list through an unsatisfied blocking IO request, due to an unavailable monitor or by its explicit request through Thread.yield() or Thread.sleep().

Since the scheduling also constitutes a permanent administrative burden to the computing core, we opt at a direct hardware implementation in the same spirit as the transparent automatic memory management. Using a shared stack module, which contains most of the Threads states, this might even enable small multi-core systems with very efficient context transfers.

G. Preliminary Results

Currently, prototyping of the SHAP microarchitecture is done on a SPARTAN-3 Starter Kit Board. The actual configuration is:
- 8 KByte stack, up to 32 threads,
- 2 KByte method cache,
- memory manager with GC,
- memory controller for external 1 MByte SRAM,
- bus devices: UART, LCD, PS/2, memory statistics unit
- clock frequency of 50 Mhz,

and has a resource usage on the Spartan3 XC3S1000 of:

<table>
<thead>
<tr>
<th></th>
<th>w/ GC</th>
<th>w/o GC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>2387</td>
<td>1359</td>
</tr>
<tr>
<td>Block RAMs</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>18×18 multipliers</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>User I/O pins</td>
<td>95</td>
<td>95</td>
</tr>
</tbody>
</table>

We do not include a table with the execution time of specific bytecodes because this information is pointless without the execution frequency in a real-world application. Thus, we are working on getting a standard benchmark to run. Preferably the SPEC JVM98 to enable the comparison with the results of others.

VI. THE SHAP JVM IMPLEMENTATION

The SHAP Java Virtual Machine is completely implemented in microcode which runs on the SHAP microarchitecture. This section describes the organization of our JVM implementation and the data structures used by the microcode. But first, we give definitions for some terms used in this section.

A reference is a handle to a heap memory area, created and managed by the memory manager. One example is an object reference as it is known from the Java programming language.

An offset locates a specific value inside a reference.

Here, a pointer is not a memory address. Instead it points to a specific data structure by indices and offsets. For example, a method pointer is composed of a class number and an offset. The class number is resolved to a reference to a class object, and the offset locates the method inside the class object.

All fixed information used by the JVM, e.g., methods and constant pools, is generated by the SHAP linker. The linker reads in a collection of Java .class files, links them and write outs a .shap file. Dynamic class loading is currently not implemented in the SHAP linker, but the JVM would already support it.

A. Primitive Types

The primitive types supported by the SHAP JVM are the integral types byte, short, int and char as well as the boolean type as they are defined in the Java programming language. The returnAddress type is not available as the bytecodes jsr, jsr, w and ret are not implemented, cf. Sec. IV. Also, support for floating point arithmetic and long 64-bit data types is missing.

Each value of a primitive type is stored inside a 32-bit word onto the stack, inside a static or instance field, or in an array.

B. Reference Types and Values

There are three kinds of reference types which must be handled by a JVM: class types, array types and interface types. The respective values are references to
C. Representation of Objects and Arrays

Each object (and array) which is created by the JVM stores a reference to the class object, this object is an instance of, and the instance fields of this object in the order defined in Fig. 11. The fields are numbered from 1 to \( n \) to enable direct field addressing while skipping the header. The order of the fields is defined by the SHAP linker. To enable casting along the class hierarchy, the linker places the new fields introduced by a class after the fields inherited from the extended class. This procedure is applied iteratively until the common base class of all classes \texttt{java.lang.Object}—\texttt{Object} for short—is reached, thus, collecting all instance fields of a class. Hence, the object representation always starts with the fields of \texttt{Object}.

Arrays are a special form of an object, as they can be casted to \texttt{Object}. Thus, the runtime objects start with the fields of \texttt{Object} followed by the length and the components of the array (Fig. 12). The maximum length of an array equals to the maximum object size defined by the memory manager minus the header size. For the associated class objects, we have introduced nine special classes which extend \texttt{Object}:

- \texttt{[I}—array of integers,
- \texttt{[S}—array of shorts,
- \texttt{[B}—array of bytes,
- \texttt{[J}—array of long integers (future release),
- \texttt{[C}—array of characters,
- \texttt{[Z}—array of booleans,
- \texttt{[F}—array of floats (future release),
- \texttt{[D}—array of doubles (future release),
- \texttt{[L}—array of objects (references).

These classes are required for type checking through the Java bytecodes \texttt{checkcast} and \texttt{instanceof}. Note, currently, we do not distinguish between different types of object arrays \texttt{[L}. This security leak will be fixed in the future.

One optimization would be to pack multiple byte, short or character array components into one 32-bit word to save memory space. We do not take this approach, so that we can use the same microcode subroutine for array reading and writing for all array types except long and double type (which are not implemented yet).

D. Runtime Class Objects

Another “special” form of objects are the class objects. As defined by the standard Java API, an instance of
Java.lang.Class, for short Class, is created for each class loaded into the JVM. Through this instance, basic information like the class name as well as enhanced reflection information only supported in complex API implementations is available. As the JVM must also deal with internal information about a class, we decided to store both parts inside the same object to save references of the memory manager. Thus, the class object, as depicted in Fig. 13, stores at first the fields of Class and afterwards the internal class structure. The class structure starts at the offset classStructOffset inside the class object. The offset is common to all class objects and is initialized during JVM initialization.

The class structure holds all the internal data about a class (Fig. 14). These are specifically:

1) the offset of the constant pool (cpOffset),
2) the size of an object instance of this class including the header,
3) the offset of the instanceof-mask (iomOffset), if this isn’t an abstract class,
4) the virtual method table (VMT),
5) one or several interface method tables (IMTs), if this class implements Java interfaces,
6) the static fields of the class, which are initialized by the linker to their default values,
7) the implemented methods of this class, and
8) the runtime constant pool associated with this class.

All offsets are absolute from the beginning of the enclosing class object.

The parts VMT, IMT, instanceof-mask, and the runtime constant pool of the class structure are now described in more detail. Handling of methods is explained in Sec. VI-E.

1) VMT: The virtual method table stores the pointers to the method implementors. Each pointer consists of the class number of the implementing class and the offset of the implementing method inside the class object. Resolving of class numbers is described in connection with the runtime CP in Sec. VI-D.4.

The order of the entries is defined by the SHAP linker. Along one branch of the class hierarchy, a virtual method has always the same index into the VMT regardless of the actual class. This is the standard approach, of implementing virtual methods in object oriented languages. The offset of the VMT entry in the class object, which corresponds to the index, is inserted as bytecode immediate for the Java bytecode invokevirtual.

2) IMT: The call of an interface method is indrected thrice. At first, the interface index from the belonging reference addresses the global IMT lookup table (see Sec. VI-B and returns the IMT to use: class number plus offset into class structure. The IMT (Fig. 15) is addressed by the method index, provide by the bytecode immediate of invokevirtual, resulting in an offset pointing into the VMT of the the same class. Finally, using the VMT entry the implemented method is resolved, as it would be done for virtual method calls.

The indirection over the VMT enables reuse of IMTs. An IMT is only generated and stored in the class structure for each interface a class implements directly and not transitively. Each subclass of such a class reuses the already existing IMTs, and a possible new implementation of an interface method will be resolved through the VMT.

3)InstanceOf-Mask: The instanceof-mask indicates super classes and implemented interfaces of this class. For each of these as well as the class itself a bit in the bit field is set. The class number of a class specifies the bit position inside this field. The field is as long as the highest set bit. The field is stored in a word array. Bit 0 is located at word 0, bit 0. The layout of the instanceof-mask in the class structure is included in Fig. 14. It contains the class number of this class, the length in words of the field and the field itself.

The instanceof-mask is used by the Java bytecodes checkcast, instanceof, and athrow.

To test, if a source class S extends/implements a given target class/interface T, two steps are necessary. At first, we read the class number classNum of T from the class structure. Then, this number is used as an index into the bit field of the instanceof-mask of S.
Thus, we have to test, if

\[
\text{the bit } = \text{classNum mod 32 inside the word } = \text{classNum/32 is set.}
\]

The word address may greater or equal than the length of the bit field. In this case the requested bit is not set.

4) Runtime Constant Pool: The runtime constant pool (CP) consists of a number of 32-bit entries each one of the types given in Fig. 16. The CP is indexed by a bytecode immediate to which the \text{cpOffset} has to be added to get the absolute address inside the class object.

The three kinds of pointers, class pointers, static field pointers, method pointers for static/special invocation, all point into another class structure holding the desired information. A simple approach would be to store the reference of the class object together with the offset. But, as the constant pool is generated by the SHAP linker, the reference is unknown. Thus, we decided to use a runtime class table (Fig. 17) which stores the references to class
objects. This table is indexed by the already assigned class numbers (cf. Sec. VI-D.3). Another solution would be to resolve and overwrite the class numbers once the structure and all referenced one have been loaded by the class loader. This approach may be taken in the future.

A different entry kind is used for virtual method calls. It contains the number of arguments of the method to invoke to find the this-reference on the stack. As well as the offset of the VMT-entry to use.

String constants in the CP are implemented using references to instances of java.lang.String. The string objects are always placed at the end of the constant pool because the reference can only be initialized at runtime by the class loader, see Sec. VI-I.

E. Methods, Frames and Exceptions

The layout of a method structure inside the class structure is given in Fig. 18. The structure contains a header, the bytecode and the associated exception table. The header specifies:

1) the number of arguments of this method including the this-reference if applicable,
2) the number of local variables without arguments,
3) the length of the method in 32-bit words, this also implies that the bytecode is padded up to the next 32-bit boundary,
4) the number of exception table entries,
5) the static flag, indicating if this is a static method,
6) the sync flag, if the method is synchronized.

A method frame is provided by the stack module. This module already implements the creation of new frames during method invocation as well as their destruction when returning to the caller, as already described in
Sec. V-B.3. This module also handles the variable pointer VP taking care of the passing of method arguments to the new frame, too. Furthermore, the frame pointer FP locates linking information, which is used by the stack module to leave the current frame and to return to the previous one.

The stack module allows FP-relative addressing, so the JVM can store additional frame variables in the following words on the stack. Currently, these are:

1) the method pointer of the method to return to,
2) the program counter to return to,
3) the monitor’s reference locked by the current method, if this is a synchronized method, otherwise null.

Beside the above frame variables, the JVM stores further frame data in microcode variables to get a fast access to frequently used values. These need not to be stored on the stack, as they can be reconstructed from the above frame variables. The additional frame data are:

- the method pointer of the current method,
- the reference to the class object of the current method,
- the offset of the runtime CP inside the class object.

The exception table lists all exceptions to be caught within a given code block and associates it with the defined exception handler. The table is always searched from top to bottom as defined by the Java compiler. For each entry, it is checked whether the Java bytecode program counter belongs to the code range specified by the values from and to (exclusive) and whether the exception thrown is an instance of the exception type to be caught as specified by its class number. The instance-of check is equal to the evaluation of the instance-of mask as described in Sec. VI-D.3. If both checks pass, the control flow branches to the handler specified by target, a bytecode offset within the current method. The special value −1 for the class number defines a finally-block and matches all exception types.

If the end of the exception table is reached, the exception is not caught inside the current method. Thus, the control flow leaves the method frame and returns to the previous one on the stack. Also, the monitor lock is released if the left method is declared synchronized. After that, the exception table of the now active method is scanned. This procedure continues until an appropriate exception handler is found. The termination of this search is guaranteed as described just below.

F. Threads and Locks

Multiple threads are managed through instances of the well-known Java class java.lang.Thread—Thread for short—or a subclass of it. These objects are created explicitly by the Java program. A new thread is considered for scheduling when its start() method has been invoked. The actual scheduling is done by the scheduler described in Sec. V-F, which utilizes some of the instance fields of Thread. The separate stack for the new thread is provided by the stack module. Switching between stacks as well as stack destruction is also accomplished by the stack module. Refer to Sec. V-B.3 for details.

The first method executed in the new thread, is its virtual run() method. As the scheduler is implemented in microcode, we need some kind of call-back from microcode to Java code. A simple solution would be to pass the VMT-entry offset of Thread.run() to the JVM during initialization (Sec. VI-H). Instead, we introduced a new static method private static Thread.runme(Thread target)
and initialize a method pointer to it during JVM initialization. This method pointer is stored inside the JVM’s constant pool, see Sec. VI-G. Although this method primarily calls target.run(), this approach has two more advantages. Firstly, it contains a try-catch-anything block with a default handler to ensure that all exceptions will be caught inside this thread avoiding a “stack underflow”. Secondly, it automatically liquidates the thread of control after its run() method returns or an otherwise uncaught exception is caught by withdrawing it from scheduling.

All three kinds of thread synchronization via locks are implemented on a per object basis:

- with a synchronized block, which locks on the specified object,
- synchronized static methods, which lock on the class object of the defining class,
- synchronized non-static methods, which lock on the implicit this argument of the specific invocation.

A private instance field has been added to the base class Object to store the information about the instance locking state. This field is also utilized for the support of the wait-notify-scheme required for an implementation of the API provided through java.lang.Object.

After the startup of the microarchitecture, the stack module already provides an initial stack, of course, without an associate instance of Thread. The microcode startup routine (JVM initialization) as well as the bootstrap class loader are executed using this stack. After loading the initial set of classes, see Sec. VI-I, the bootstrap class loader must create a new thread for the application.

G. Runtime Constant Pool of the JVM

The JVM requires a set of variables and constants to accomplish its operation. Due to their frequent usage, some of them are stored inside microcode variables for fast access and have already been introduced in the previous sections.
method pointer to Thread.runme()

method pointer to
NullPointerException.cthrow()

method pointer to
ClassCastException.cthrow()

method pointer to
ArrayIndexOutOfBoundsException.cthrow()

0x0000 class number of [Z
0x0000 class number of [C
reserved for [F
reserved for [D
0x0000 class number of [B
0x0000 class number of [S
0x0000 class number of [I
reserved for [J
0x0000 class number of [L
0x00000000 (reserved)
0x00000000 (reserved)

method pointer to
VirtualMachineError.cthrow()

max # of classes
classStructOffset

JVM constant pool (see Fig. 19)

bootstrap class object size

class structure of bootstrap class (loader)
(see Fig. 14)

method pointer to Bootstrap.main()

initialization data for classes
to load by bootstrap class loader

layout defined by bootstrap class loader

Fig. 19. Layout of JVM Constant Pool

Fig. 20. Layout of SHAP Programming-File

Not so frequently used constants are stored inside a separate memory object, called the JVM constant pool, whose layout is depicted in Fig. 19. Inside it, the layout of the method pointers equals those from the runtime CP in Sec. VI-D.4.

Some of the entries are already known: Thread.runme() from Sec. VI-F, and the class numbers of the array classes used in Sec. VI-C.

The remaining constants are method pointers to static methods, which will construct and raise the indicated exceptions. These runtime exceptions are thrown as specified by the runtime semantics of the Java bytecodes or when the virtual machine encounters an error. So, the VirtualMachineError is, for example, thrown whenever an unsupported bytecode is to be executed.

H. JVM Initialization

The startup of the JVM is performed through microcode, which reads the initialization data from a programming device, e.g., the UART. The data is generated by the linker and packed together with the class initialization data into a SHAP programming file, as depicted in Fig. 20.

At first, the maximum number of classes is given, which must not exceed the maximum object size specified by the memory manager. This number may be greater than the actual number of classes loaded by the bootstrap class loader to enable dynamic class loading. The classStructOffset specifies the offset of the class structure inside the enclosing class object, see Sec. VI-D. Then follows the JVM constant pool (see Sec. VI-G), which is copied into the runtime object.

Finally, the bootstrap class loader follows. This is the first Java class loaded and executed by the JVM as described in Sec. VI-I. The layout of the method pointer to Bootstrap.main() follows that of a standard constant pool entry for static method calls as described in Sec. VI-D.4.

The bootstrap class loader is also responsible for the remaining part of the SHAP programming file: the initialization data for the classes to load as described in the next subsection.

I. Bootstrapping

To save microcode, loading of the initial Java classes into the memory from the programming device, is also done by Java bytecode. The corresponding bootstrap method is located in the Bootstrap class, which is the only class loaded by microcode. The initial class set loaded by the bootstrap process can be a complete
API as well as the minimal set of classes to provide the system class loader.

At first, the bootstrap class loader reads class structures from the programming device until the end marker \(0xFFFFFFFF\) is reached, and creates the appropriate class objects as defined in Sec. VI-D. It also creates the string objects for the string constants in the constant pool (see below).

In the beginning, there is no other class available so reading from the programming device must be directly done via native methods. String objects can only be constructed after the class object for \(\text{java.lang.String}\) is loaded. Two additional classes are needed for that: \(\text{Object}\) for object construction as well as \([\text{C}\) for storing the string contents. Furthermore, the classes required by the JVM constant pool must be included. This implies the order and the rules of class loading:

1) Read the class initializer of \(\text{Object}\) and create the class object. This class must not contain any string constants.

2) Read the class initializer of \([\text{C}\) and create the class object. This class does not contain any string constants by definition.

3) Read the class initializer of \(\text{String}\) and create the class object. This class may only contain string constants if the constructor does not use them. This is possible because the first part of the constant pool (the part without the string constants) is already available, e.g. to invoke the superclass constructor.

4) Read the class initializer and create the class objects for all classes required by the JVM constant pool as well as their superclasses transitively). It is important that the bootstrap class loader does not implicitly or explicitly throw any exceptions before the exception classes are loaded.

5) Load all other classes of the application/system class loader.

Class objects are created on the heap by creating an integer array first. At the beginning, an instance of the integer array class \(\text{I}\) shares the same fields as an instance of \text{Class} (class object) those of \text{Object}. But then follow the array length on the one hand, and the additional fields of \text{Class} on the other. This difference is respected during code building of the bootstrap class loader by the SHAP linker.

At first, each class initializer (layout included in Fig. 21) specifies the class number. At this position the special value \(0xFFFFFFFF\) is read if the end of the class initializer list is reached. Secondly, the initializer specifies the size of the array to create first. This array is filled, from an initial offset according to the difference described above, with the class structure provided next. The latter equals that of the runtime class structure in the class object excluding the string references. These are created at runtime using string initializers (included in Fig. 21). The number of string initializer is calculated from the array size and the class struct size without string constants. Finally, the class is inserted into the runtime class table (cf. Sec. VI-D.4) by calling the native method \text{setClass()} which also “casts” the array to a class object.

After reading and creating classes, the IMT table (see Sec. VI-B) is read and stored in a heap memory area. This implies, that interfaces must not be used before. The size of the runtime IMT table may be greater than the actual number of entries provided to enable dynamic class loading.

Finally, the bootstrap class loader creates a new thread object and starts the new thread. A new thread is required because the main thread, which runs the bootstrap method, cannot be properly initialized at startup. Currently, an instance of \text{Startup} is created which extends \text{Thread}.

The \text{Startup} class extends \text{Thread} and implements the \text{run()} method. This method is generated by the SHAP linker and:

1) sets all class names in the class objects,
2) invokes all class initializers,
3) and invokes all \text{main()} methods.

All of the required strings and methods are collected at link time.

\section{J. Resource Usage}

The JVM implementation requires 1660 microcode instructions which is 81% of the memory block provided by the prototyping platform. The available memory may be increased, but we try not to exceed this barrier, to hold a small memory footprint.

Currently, this microcode utilizes 14 out of 16 microcode variables, 32 out of 48 constants and 54 out of 64 branch entries. The upper bounds cannot be increased without a modification of the microcode instruction set.

\section{VII. Conclusion and Future Work}

This technical report presented a novel implementation of an embedded Java microarchitecture for secure, real-time, and multi-threaded applications. Due to its additional support for modern features of object-oriented languages, such as exception handling, automatic garbage collection and interfaces, it also establishes a general-purpose platform built without an underlying operating system. The associated implementation of the Java Virtual Machine is explained in detail. Especially, the layout and usage of runtime data structures is described as well as the boot sequence of the JVM.

New techniques have been implemented for specific real-time issues, such as an integrated stack and thread management for fast context switching, concurrent GC for
real-time threads and autonomous control flows through preemptive round-robin scheduling. Open issues are the further improvement of the memory management as well as the integration of dynamic class loading.
## Appendix

**TABLE V**  
**Short Description of Microcode Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add</code></td>
<td>Add TOS to NOS.</td>
</tr>
<tr>
<td><code>and</code></td>
<td>Bitwise AND of NOS and TOS.</td>
</tr>
<tr>
<td><code>bne addr</code></td>
<td>Branch to address, if TOS ≠ 0. Leave TOS untouched.</td>
</tr>
<tr>
<td><code>beg addr</code></td>
<td>Branch to address, if TOS = 0. Leave TOS untouched.</td>
</tr>
<tr>
<td><code>bnez addr</code></td>
<td>Branch to address, if TOS ≠ 0. Leave TOS untouched.</td>
</tr>
<tr>
<td><code>bx addr</code></td>
<td>Branch to address, if TOS = 0. Leave TOS untouched.</td>
</tr>
<tr>
<td><code>bge addr</code></td>
<td>Branch to address, if TOS ≥ 0. Leave TOS untouched.</td>
</tr>
<tr>
<td><code>ble addr</code></td>
<td>Branch to address, if TOS ≤ 0. Leave TOS untouched.</td>
</tr>
<tr>
<td><code>blt addr</code></td>
<td>Branch to address, if TOS &lt; 0. Leave TOS untouched.</td>
</tr>
<tr>
<td><code>b addr</code></td>
<td>Branch always to address.</td>
</tr>
<tr>
<td><code>dup</code></td>
<td>Duplicate TOS.</td>
</tr>
<tr>
<td><code>enter</code></td>
<td>Enter new method frame. Number of locals and arguments is specified by TOS.</td>
</tr>
<tr>
<td><code>fbc</code></td>
<td>Fetch next Java bytecode and shift into internal register.</td>
</tr>
<tr>
<td><code>fbcidx</code></td>
<td>Fetch next Java bytecode and shift into internal register. Afterwards, load internal register into special IDX register.</td>
</tr>
<tr>
<td><code>fbcn</code></td>
<td>Fetch next Java bytecode and shift into internal register. Afterwards, put internal register as byte sign-extended onto stack.</td>
</tr>
<tr>
<td><code>fbcr</code></td>
<td>Fetch next Java bytecode and shift into internal register. Afterwards, put internal register as 16-bit short sign-extended onto stack.</td>
</tr>
<tr>
<td><code>fbcrb</code></td>
<td>Fetch next Java bytecode and shift into internal register. Afterwards, put internal register as byte onto stack.</td>
</tr>
<tr>
<td><code>fbcrs</code></td>
<td>Fetch next Java bytecode and shift into internal register. Afterwards, put internal register as 16-bit short onto stack.</td>
</tr>
<tr>
<td><code>incidx</code></td>
<td>Increment special IDX register.</td>
</tr>
<tr>
<td><code>jbr</code></td>
<td>Execute a Java bytecode branch.</td>
</tr>
<tr>
<td><code>ld addr</code></td>
<td>Load microcode variable.</td>
</tr>
<tr>
<td><code>leave</code></td>
<td>Leave method frame.</td>
</tr>
<tr>
<td><code>ldbw</code></td>
<td>Load stack variable relative to stack pointer.</td>
</tr>
<tr>
<td><code>ldfidx</code></td>
<td>Load method frame variable at index.</td>
</tr>
<tr>
<td><code>ldiod</code></td>
<td>Load data from I/O address (stioa).</td>
</tr>
<tr>
<td><code>ldios</code></td>
<td>Load status from I/O address (stioa).</td>
</tr>
<tr>
<td><code>ldjpc</code></td>
<td>Load JPC.</td>
</tr>
<tr>
<td><code>ldlv</code></td>
<td>Load local method variable.</td>
</tr>
<tr>
<td><code>ldref</code></td>
<td>Load object data (type: reference) at address set by stofs.</td>
</tr>
<tr>
<td><code>ldval</code></td>
<td>Load object data at address set by stofs.</td>
</tr>
<tr>
<td><code>mul</code></td>
<td>Multiply NOS by TOS.</td>
</tr>
<tr>
<td><code>newref</code></td>
<td>Allocate new object with size &lt;TOS&gt;.</td>
</tr>
<tr>
<td><code>nop</code></td>
<td>No operation.</td>
</tr>
<tr>
<td><code>pop</code></td>
<td>Pop TOS.</td>
</tr>
<tr>
<td><code>shl</code></td>
<td>Signed shift left NOS by TOS.</td>
</tr>
<tr>
<td><code>shr</code></td>
<td>Signed shift right NOS by TOS.</td>
</tr>
<tr>
<td><code>st addr</code></td>
<td>Store TOS into microcode variable.</td>
</tr>
<tr>
<td><code>stioa</code></td>
<td>Set I/O address.</td>
</tr>
<tr>
<td><code>stiod</code></td>
<td>Write TOS to previously set I/O address.</td>
</tr>
<tr>
<td><code>stjpc</code></td>
<td>Store TOS into JPC.</td>
</tr>
<tr>
<td><code>stlv</code></td>
<td>Store TOS into local method variable.</td>
</tr>
<tr>
<td><code>stofs</code></td>
<td>Set offset inside object-ref activated by stref.</td>
</tr>
<tr>
<td><code>stref</code></td>
<td>Activate object-ref &lt;TOS&gt;.</td>
</tr>
<tr>
<td><code>stval</code></td>
<td>Write TOS into object at address set by stofs.</td>
</tr>
<tr>
<td><code>sub</code></td>
<td>Subtract TOS from NOS.</td>
</tr>
<tr>
<td><code>tkill</code></td>
<td>Kill thread identified by TOS.</td>
</tr>
<tr>
<td><code>tswitch</code></td>
<td>Switch to thread identified by TOS. Create new thread if TOS = -1.</td>
</tr>
<tr>
<td><code>ushr</code></td>
<td>Unsigned shift right NOS by TOS.</td>
</tr>
<tr>
<td><code>wait</code></td>
<td>Wait for method cache completion.</td>
</tr>
<tr>
<td><code>xor</code></td>
<td>Bitwise XOR of NOS and TOS.</td>
</tr>
</tbody>
</table>
Fig. 22. Encoding of Microcode Instructions (1)
Fig. 23. Encoding of Microcode Instructions (2)
and Java Virtual Machine

REFERENCES


