Department of Computer Science

Johan Garcia

Implementing Java on a Non-Generic Platform

Master's Thesis

1999:4

Implementing Java on a Non-Generic Platform

Johan Garcia

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Johan Garcia

Approved, June 9, 1997

Opponent: Anna Brunström

Advisor: Donald Ross

Examiner: Anna Brunström

Abstract

Java offers a number of advantages for software development. These advantages can be beneficiary also to non-generic platforms such as telecommunication switches. This paper examines a possible implementation of a Java execution system on the APZ 212 telecommunication switch central processing system. An overview of the Java Virtual Machine Specification and of the APZ 212 is given. The problems and possibilities of implementing a Java execution system based on the Way-Ahead of Time compiling (WAT) execution model is presented.

The study shows that it is possible to implement a Java execution system on the APZ 212, but that several problem areas exist. The main problems arise in the fields of negative number representation and data sharing between objects.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Java is a concept that has attracted much attention recently. In addition to being a new programming language, Java also defines an execution environment that has strong provisions for interoperability and security. The Java programming language is designed to address most of the weaknesses of C and C++, but still be a general programming language. A general background on Java is provided in Appendix A. This dissertation studies the possibility of implementing a Java execution system on a non-generic platform. The platform examined in this dissertation is the processor system found in telecommunications switches manufactured by Ericsson, specifically the platform called APZ 212. This platform differs markedly from a conventional processor platform since it has no shared linear address space.

1.1 Dissertation structure

This dissertation comprises a general introduction (chapter 1) followed by a chapter describing the Java Virtual Machine and runtime system. Then a chapter describing the processor architecture/environment of the APZ follows. These chapters are relatively detailed in order to lay the foundation for the next chapter, a discussion on how an actual

implementation of the Java environment for the specific platform might be done. This chapter provides a synthesis of the two previous, and describe a possible mapping of a Java execution system onto the target hardware. The last chapter presents a summary of the previous chapters and states the conclusions that are drawn in relation to the research questions, see section 1.2. An appendix is provided to present the concepts behind Java.

1.2 Research questions

The problem domain studied in this dissertation is the possibility of implementing an execution system for Java on a non-generic computer platform, in this case the APZ 212. The research questions are:

- Is it possible to implement a practically useful Java environment for a non-generic hardware platform which has a hardware architecture that is markedly different from the hardware architecture the Java VM was targeted for?
- If so, what are the problems/possibilities inherent in the mapping of the execution model of Java to the specifics of the hardware platform, and what implementation strategy is best suited to make maximum use of the hardware.
- How is the execution performance of Java programs affected by the above factors and what is the level of performance to be expected in comparison with similar implementations using other hardware architectures.

1.3 Scientific Paradigm

The dominating scientific paradigm in this study is the system theoretic paradigm. The problem domain will be divided into parts which are studied separately. This study will cover both the APZ environment and the Java VM environment. These are disseminated

to find the possible structures and methods that can be used to overlay Java functionality on the APZ. The scientific basis for this study is literature studies. A further study of this topic would probably include aspects of the positivistic paradigm as further work would typically entail implementing an experimental Java execution system in order to measure the actual performance and relating this to the performance available on typical Java execution system implementations on generic platforms.

1.4 Purpose

The purpose of this study is to give an indication if Java can be viable development tool for developing telecommunications software that is to be executed by the processor in the exchange.

1.5 Restriction

This dissertation studies the applicability of a Java environment for telecommunications software development from a technological point of view. The focus will be if it is technologically feasible to create an execution system that is capable of using Java class files to control a processor in an exchange. No discussion of the Java language suitability *per se* for the domain of telecommunication application development will be presented.

1.6 Prerequisites

The reader is assumed to have general knowledge of the object-oriented paradigm. Thorough knowledge of the APZ structure or Java is not required since the technicalities relevant to the specific problem area are discussed in chapters 2-4. For readers who are are unfamiliar with the basic concepts behind Java, a brief introduction is present in appendix A.

1.7 Definition of terms

This study will use the following terms in the meaning stated below:

- **Java Virtual Machine** As stated in the book "The Java Virtual Machine Specification" [17], the Java Virtual Machine is an abstract design. In some literature the term Java Virtual Machine is confused with an *implementation* of the Java Virtual Machine. This study will refer to the Java Virtual Machine as being an abstract entity only.

- **Execution model** The execution model is the term used to discriminate between implementations that distribute the translation of source code to native code differently in time. For example are interpretation and compilation different execution models.

- **Java execution system** A Java execution system is an implementation of a Java Virtual Machine along with the support functionality needed to execute Java class files.

Chapter 2

Java Virtual Machine Description

This chapter gives an overview of the characteristics of the Java Virtual Machine (JVM) and goes into detail where appropriate with respect to the focus of this dissertation. This means that topics judged to be of secondary concern when implementing a Java execution system for the APZ will not be covered. Such topics include floating point support and some security considerations.

This chapter describes version 1.0.2 of the Java Virtual Machine (JVM) as defined by Lindholm and Yellin [17]. This was the current version at the time of writing (Spring 1997) and every future release of a JVM specification is expected to be backwards compatible with this version. Besides the definition several introductory presentations of the JVM are available [13, 16, 26].

2.1 Overview

This section gives an overview of some of the properties of the JVM in order to give an overall 'feel' for the JVM before going into details in the following sections.

The JVM:

- is a stack-oriented machine, i.e. an operand stack is used instead of registers for

arithmetic operations.

- has an address space, register size and stack width of 32 bits.
- can handle multiple threads, and each thread has its own runtime stack.
- has primitive data types ranging from 8 to 64 bits wide.
- has builtin support for exceptions.
- has a data structure referred to as the constant-pool which is used akin to a symbol-table.
- uses run-time resolution of class names for dynamic linking.

The typographical conventions used in this chapter are:

- instructions will be *italiziced.*
- types and Java source code will use the `typewriter` font.

2.2 Supported types

The JVM has two kinds of types:

- primitive types, used to hold atomic values.
- reference type, used to hold values that reference (points to) an object or interface.

2.2.1 Primitive types

The primitive types of the JVM are specified in table 2.1 and maps nicely to the types available in the Java language as specified in [11], with the exception of the `returnAddress`

type which is a data type internal to the JVM. The `returnAddress` type is a pointer to JVM instructions and is used by the JVM instructions *jsr*, *ret* and *jsr_w*.

As evident by looking at table 2.1, integral values are signed and can have a size ranging from 8 to 64 bits. The smaller types (`byte` and `short`) are however automatically extended to `int` before they are pushed on the operand stack or stored in a local variable since the stack as well as local variables are aligned on four-byte boundaries as described in section 2.3. Since the JVM instructions for arrays have support for arrays of byte and short this suggests that all integral values that are not stored in arrays would optimally be defined as `int` in order to minimize type conversions done by the JVM.

Nr. of bits	Type Name	Description
8	`byte`	Signed integral value
16	`short`	Signed integral value
32	`int`	Signed integral value
64	`long`	Signed integral value
16	`char`	Unicode character (unsigned)
32	`float`	IEEE 754 Float
64	`double`	IEEE 754 Float
32	`returnAddress`	pointer in JVM's address space

Table 2.1: JVM primitive types

As will be seen in section 2.4 a feature of the JVM instruction set is that the type of the operands are defined by the instruction, thus eliminating the need for a type-defining bit field as part of the instructions. This requires all type information regarding the primitive types to be known at compile time which also exempts the need for run-time type checking, yielding an improved performance.

Although there exists a boolean type in the Java Language, no such type is implemented in the JVM. Instead the JVM uses int values to express logical values following the C convention that any nonzero value is true. Reference types can also be used with the convention that any non-null value is true. Although a boolean type is not supported in the JVM instruction set it is possible to create arrays with the JVM instruction *newarray*

that are specified to hold boolean values. The JVM specification [17] leaves the choice of representing the boolean values inside the array as bytes or as packed bits in a byte to the implementer of the JVM. Insertion or retrieval of boolean values in the array is however specified to be performed using bytes for representing the boolean values.

2.2.2 Reference type

The reference type is used to refer to objects. The concept of references is similar to the concept of pointers as found in C and C++. There are however some major differences between the two:

- References may not be assigned absolute memory addresses.
- References are automatically dereferenced, no explicit dereferencing is necessary as in C.
- No arithmetic operations are allowed on references, pointers allow pointer arithmetic.

The above differences provide added security over pointers as implemented in C and add to the overall security that was a prime design goal of Java.

2.3 Run-Time Data Areas

The details of the Run-time data areas are not a part of the JVM specification. There are however some basic structures that are described in section 3.5 of Lindholm,Yellin [17]. These are the Java stacks, frames, heap, method area and constant pool. These will be described in the forthcoming sections, and a pictorial summary can be found in figure 2.1. The figure serves only an example of how the run-time data areas might be implemented. The functionality provided by the structures showed in the picture would however need to be present in some way in every implementation of a JVM.

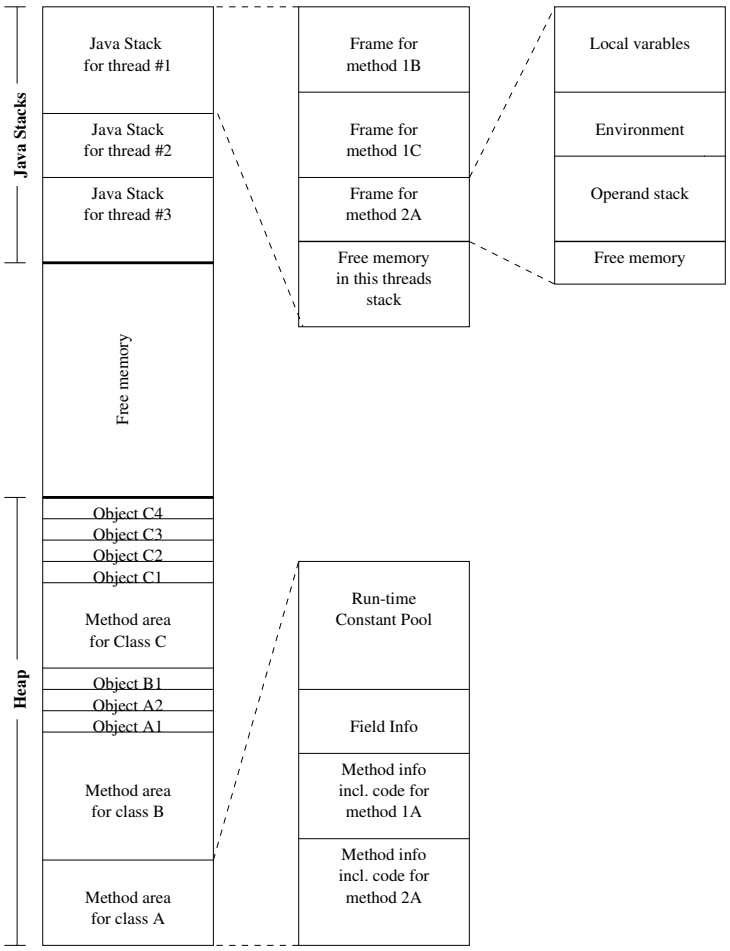

Figure 2.1: Run-time Data Areas

2.3.1 Java stacks

The JVM is capable of handling multiple parallel threads of execution. In order to record the state of each thread, every thread has an associated Java stack. This Java stack contains the frames, also called activation records, of the chain of method invocations that lead to the current method. The exact content of the frames is implementation specific but they typically contain storage for local variables, a frame info part containing various implementation specific data and an operand stack. The frame info part (sometimes called environment) may for example hold information regarding the caller's frame location and

state data. A slide showing the layout of Sun's frame info structure can be found in Yellin, Lindholm [37]. Since Java stacks only handle whole stack frames at a time, a JVM implementation may allocate Java stacks discontinuously and may also dynamically change the size of the stack.

2.3.2 Heap

The heap is the JVM dynamic allocation area used by all threads to store objects. Since the JVM may dynamically load classes, the method areas (see next section) may also be allocated on the heap. Heapsize may change dynamically and heap memory need not be continuous. Garbage collection is used for the heap, but choosing which specific garbage collection algorithm to use is left to the implementer. Several considerations have to be made when choosing a garbage collection algorithm. Further discussion of these issues can be found in Venners [25]. A discussion of how to minimize the need for garbage collection in Java programs can be found in McManis [19].

2.3.3 Method area

The method area[1] is the place where the model JVM implementation stores a run-time representation of the information found in classfiles. This information includes the constant pool, field specifications and method specifications. All threads share one representation of every loaded class. As is shown in section 2.5.4, the bytecodes(the actual code for the JVM) are stored in the method info section of the method area. The layout of and handling of this data-structure is left to the implementer of the JVM.

In order to bind the symbolic references found in the constant pool, the entries in the constant pool have to be resolved i.e the symbolic references are bound to the corresponding physical structure in the JVM. If this structure is a class or an interface and it is not yet

[1]Why Sun decided to call this area 'method area' in lieu of 'class area' is unknown, but 'class area' would seem more appropriate judging by the contents.

loaded, the JVM loads and initializes it. This loading and initialization causes loading of the loaded class' superclass(recursively) and execution of its static initializers. The exact time when this resolution is to be performed is not specified, it can be done the first time a constant pool entry is actually referenced(late), or all constant pool entries can be resolved when a class is loaded(early). Details of the constant pool resolution process can be found in chapter 5 of Lindholm, Yellin [17] and of initialization in section 2.16.4 of the same.

2.4 Instruction set

This section describes the instruction set of the JVM. The instruction set is specified in chapter five of Lindholm, Yellin [17]. A briefer overview can be found in Case [5][2]. This section will begin with some characteristics and then provide an overview of the instruction set presenting more details for the more complex instructions.

The instruction set of the JVM has some noteworthy characteristics:

- Instruction opcodes are stored in one byte. The term **bytecode** is commonly used to describe an instruction's numerical value.
- The opcode can be followed by zero or more operands. No alignment is required for the instruction stream, with the exception of the instructions *lookupswitch* and *tableswitch*.
- Arithmetic instructions and stack instructions are type specified, i.e. the type of the operand is given by the opcode.
- Java VM instructions signal runtime errors by throwing exceptions. These exceptions can then be handled in a similar manner as the exceptions thrown from within the program.

[2]This article gives a good overview but contains a technical inaccuracy since the author claims that classfile bytecodes are stored in the constant pool area, whereas they are in fact stored in the method area.

- Some instructions index into a high-level data structure, the constant pool. See section 2.5.3.

2.4.1 Instruction set overview

Typed instructions generally have versions for the `int`, `float`, `long` and `double` data types. `Byte` and `short` are sign-extended to `int` before they are stored on the stack or in a local variable, and the `int` version of instructions is used to operate on them. The instruction set is non-orthogonal, since all typed instructions do not have versions for all types. The `int` type has most supporting variants and can be considered to be the 'standard' type.

The instructions can be grouped as follows:

1. Push constant

   Instructions that push constant values onto the operand stack. The values can either be given explicitly by the instruction (i.e. *iconst_0*) or by an operand indexing into the constant pool. See Venners [26].

2. Load/Store from/to local variable

   Instructions that move values to/from local variables from/to the stack. The local variable index can be given either explicitly by the instruction (i.e. *istore_1*) or by an operand indexing the local variable. See Venners [26].

3. Stack management

   Instructions that manipulate the stack, copying, moving or removing values on the stack. Examples: *dup, swap*. These instructions are type-independent (although type size matters).

4. Arithmetic

   Instructions that perform the common arithmetic functions (+, -, *, /, mod, negate) on `int`, `long`, `float` and `double` respectively. See Venners [27, 28].

5. Logical

   Instructions that perform shifts, logical and, or, xor functions on `int` and `long`. See Venners [28].

6. Type conversion

   Instructions that convert between `int`, `long`, `float` and `double`. Also instructions that narrow `int` values to `byte`, `char` and `short` values (which still are stored internally as 4 byte values). See Venners [26].

7. Control transfer and compare

   Instructions that perform unconditional or conditional jumps. Tests for conditional jumping are most plentiful for `int`, with `long`, `float` and `double` requiring a two phase compare and branch on result instead of just branch on compare. See Venners [32].

8. Table jumping

   Instructions *lookupswitch* and *tableswitch* provide for a comparison of a key value against a set of match values and jump if a match is found. See section 2.4.2 and Venners [32].

9. Array management

   Instructions for allocating, storing and retrieval of arrays. Array instructions support the `byte`, `short`, `char`, `int`, `long`, `float`, `double` and `objectref` types. Arrays containing boolean values can also be created, these use `byte` type for storage and retrieval. See section 2.4.3 and Venners [29].

10. Object related

    These instructions handle different aspects of object operations and may be further subdivided:

(a) Method invocation and return

Instructions that invoke methods and return from methods

(b) Field manipulation

Instructions that store and retrieve field values for object or class fields.

(c) Miscellaneous object handling

Instructions *new, instanceof* and *checkcast* which create a new object, checks inheritance and checks object type respectively.

See section 2.4.4 and Venners [29].

11. Monitors and Exception handling

Instructions *monitorenter* and *monitorexit* provide locking for synchronization. Instruction *athrow* throws an exception. See section 2.4.5 and Venners [30].

2.4.2 Table jumping instructions

The JVM instruction set has two instructions for supporting switch-case type of statements:

lookupswitch compares the key value on the stack against all match values in the instruction, and jumps to the associated branch offset. If a match is not found, a jump to the default branch offset is made.

tableswitch compares the value on the stack against the low and high range values of the instruction, and jumps to the [value-low] branch offset if the value is inside the range. If the value is outside the range it jumps to the default branch offset. Every value inside the range low-high must have a branch offset.

Both of the above instructions use only int values for comparison, and both have zero to three bytes of padding after the bytecode. See figure 2.2

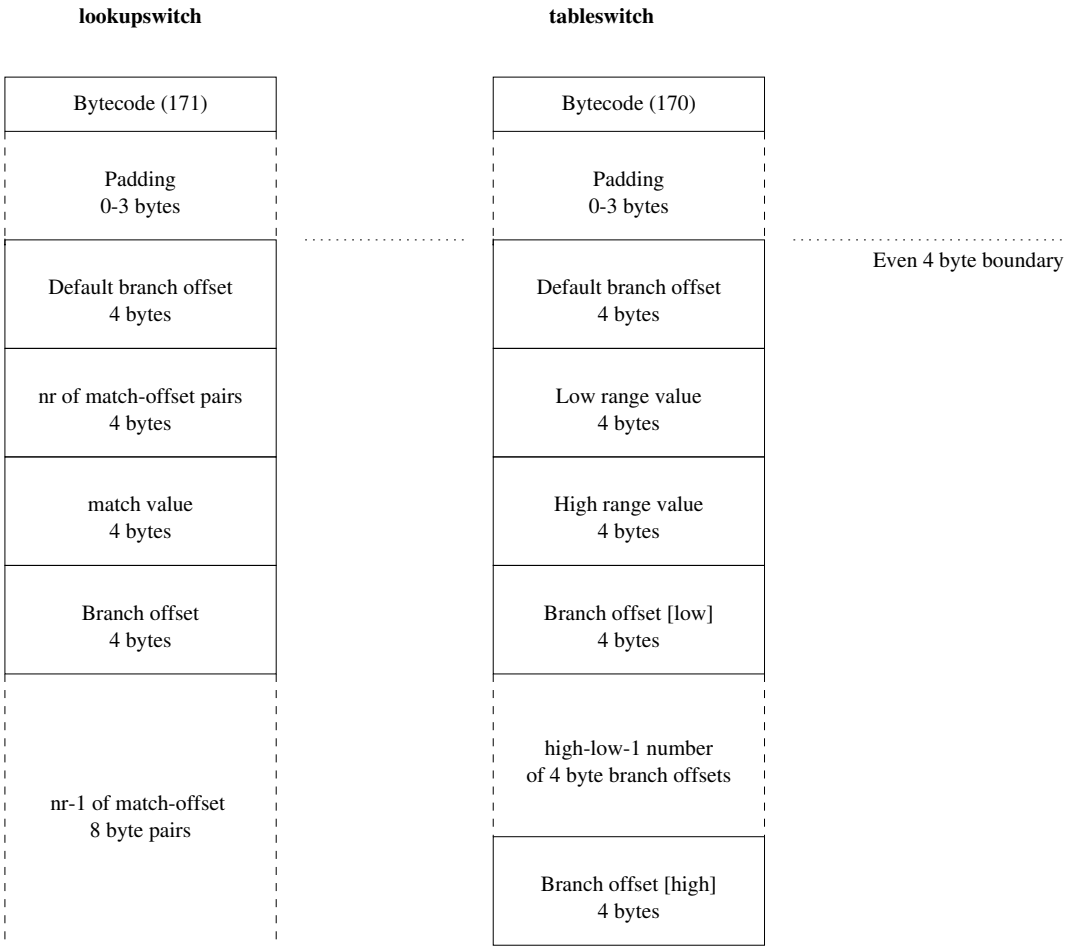

Figure 2.2: Lookupswitch and tableswitch

2.4.3 Array management instructions

The JVM instruction set supports creation and handling of arrays:

newarray Creates an array of a specified JVM primitive type or boolean with length of array value popped from the stack.

anewarray Creates an array of objects of a specified class with length of array value popped from the stack. Only an array of references is allocated, not the objects themselves.

multianewarray Creates a multidimensional array of objects of a specified class with specified number of dimensions. The size of each dimension is popped from the stack.

All the array creation instructions return an arrayref on the stack.
In addition to the above instructions there also exist instructions for data storage and retrieval:

Xaload pops index and arrayref from stack and pushes arrayref[index] value.

Xastore pops value, index and arrayref from stack and assigns arrayref[index] = value.

X in the above instructions may be any letter of { b, c, s, i, l, f, d, a } signifying types `byte`, `char`, `short`, `int`, `long`, `float`, `double` and `objectref`. Lastly, there is also one support instruction:

arraylength pops arrayref from stack and the pushes the length of the array.

2.4.4 Object related instructions

The JVM has several object related instructions that are coupled to the object model of the Java Language. The features of the object model are summarily described in appendix A. Additional information can be found in chapter seven of Anuff [1] or chapter five of Niemeyer, Peck [21].

Method invocation and return

invokevirtual invokes an instance method based on the class of the instance (dynamic). This instruction specifies a constant pool index to a method and pops an object reference and the arguments.

invokestatic invokes a class method. (non dynamic). This instruction specifies a constant pool index to a method and pops the arguments.

invokespecial invokes a superclass, private or instance initialization method. (non dynamic) . This instruction specifies a constant pool index to a method and pops an object reference and the arguments.

invokeinterface invokes an interface method (dynamic). This instruction specifies a constant pool index to a method and the number of arguments. It pops an object reference and the arguments.

If an instruction is marked as (dynamic) the method to be run is decided at run-time based on the class of the object. This is a consequence of the method overriding in the Java Language as exemplified below.

Method overriding example

This simple example of method overriding uses a vehicle metaphor as illustration.

```
class Vehicle
{
  void move() {...}
}

class Car extends Vehicle    //Car inherits Vehicle
{
  void move() {...}
}

class DoSomething ()
{
  Car volvo = new Car();     //Makes new Car object
  Vehicle transport = volvo; //Assign to Vehicle var

   transport.move()           //Accesses Car move()
}
```

This code illustrates the fact that it is the runtime type of an object that decides which method is to be selected. This can be seen as the method run in the example above is dependent on the runtime class of the object in the variable transport, not by the type of the variable. This mechanism can be achieved by multiple lookups as illustrated in fig 2.3. As seen in the figure, the relative location of the method move() is the same in the subclass Car as in the superclass Vehicle. This allows the MethodIdx offset to be used for both Vehicle and, as in this case, Car classes. The internal structure of the JVM including

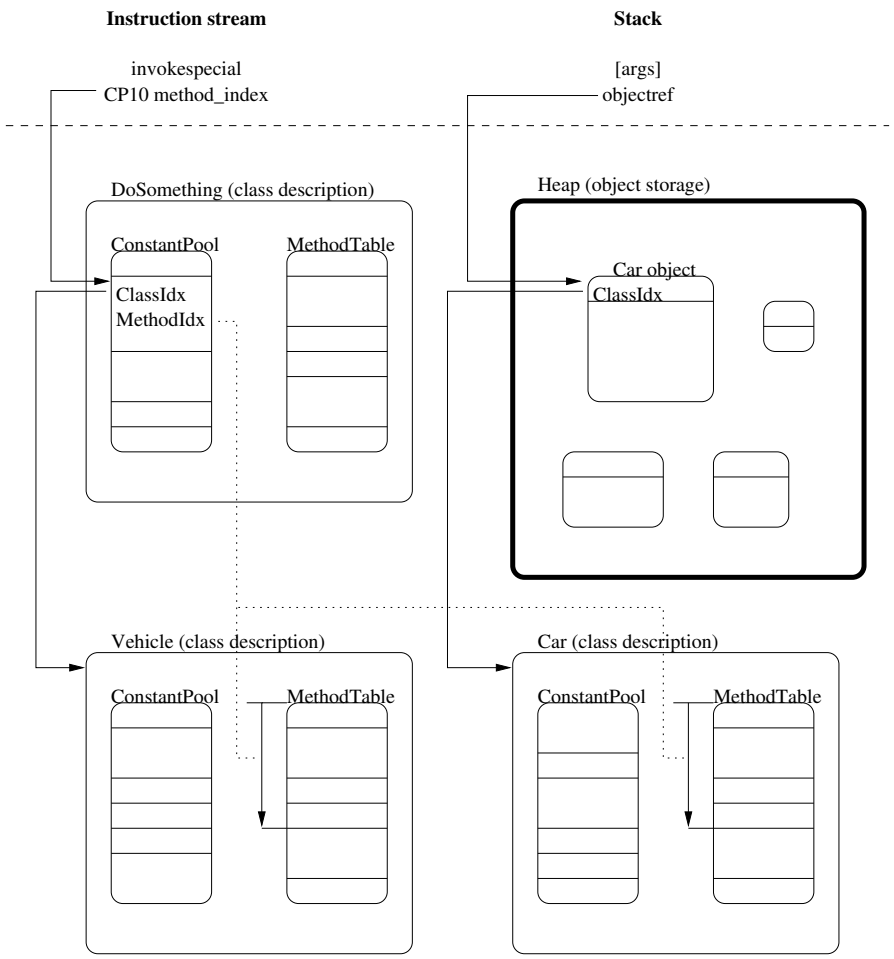

Figure 2.3: *invokevirtual* dynamic method selection

the runtime representation of the constant pool is implementation specific and is not given in any specification, so the structures are only examples. Each of the method invocation instructions creates a new frame on the stack of the executing thread (see section 2.3).

Complementing the method invocation instructions are the return instructions: *return, ireturn, lreturn, freturn, dreturn, areturn* which return from methods with values of type `void`, `int`, `long`, `float`, `double` and `objectref` respectively. On return the stack frame of the calling method is reinstated and the stack frame of the invoked method, including the operand stack, is discarded.

Field manipulation

Instructions that store and retrieve field values for object or class fields exist in the following variants:

putfield specifies a constant pool index to a field. Pops a value and `objectref`. The value is stored in the object's designated field.

getfield specifies a constant pool index to a field. Pops an `objectref`. The value of the object's designated field is pushed onto the stack.

putstatic specifies a constant pool index to a field. Pops a value. The value is stored in the class' static field.

getstatic specifies a constant pool index to a field. The value of the class' static field is pushed onto the stack.

As can be derived from the above *putfield* and *getfield* operates on instance variables, whereas *putstatic* and *getstatic* operates on class variables.

Miscellaneous object handling

new specifies a constant pool index to a class. Memory is allocated and thus is the class instantiated as a new object on the heap. An objectref is pushed on the stack. To complete the creation of a new object the instance initialization init method must be called before the object is used.

instanceof and *checkcast* both specifies a constant pool index to a class or interface. Both pops an `objectref` and check if the object is the same class or a subclass of the specified class (respectively implements the interface). *instanceof* pushes 1 onto the stack if this is true and 0 else. *checkcast* pushes the `objectref` back if true and throws en exception else. See Venners [29].

2.4.5 Monitors and Exception handling instructions

monitorenter and *monitorexit* both pop an `objectref` and can be used to ensure that only one thread uses an object at any time. These instructions set and clear the monitor flag associated with every object.

athrow pops an `objectref` that must be an object of class Throwable or a subclass. The current method's exception table is searched for a matching handler. If none is found, the stack is unwound and the caller of the current method has its exception table checked for a matching handler. This is repeated until a matching handler is found or, if the calling chain is exhausted and no handler is found, the thread becomes terminated.

2.5 Class file format

A Java class file is the representation of a compiled Java language class. It contains Java bytecodes for the methods of the class as well as specifications for the fields of the class and additional information to support the Java execution system. In order to understand the information that an implementation of a Java execution system has at its disposal, knowledge of the data structures of the classfile is essential. The format of the class file is defined in chapter 4 of Lindholm, Yellin [17]. A brief description of the class file format can be found in Venners [24]. An introduction to Java Classloaders, the mechanism of the runtime system that loads class files, can be found in McManis [18]. This section will give a fairly detailed overview of the class file layout, although focusing more on comprehensibility rather than minute detail.

Naming conventions used in this section adheres as much as possible to the one found in Lindholm, Yellin. One exception is made for the sake of brevity: Constant pool entry types originally named **CONSTANT_xxxx** are referred to as **C_xxxx** throughout this section.

2.5.1 Introduction

The Java class file layout can be separated into its main parts as follows:

magic Four bytes (0xCAFEBABE) used to ascertain that this file indeed is a class file.

minor_version Two bytes containing the minor Java version number of the compiler that produced the class file.

major_version Two bytes containing the major Java version number of the compiler that produced the class file.

constant_pool_count Two bytes containing the number of entries in the constant pool.

constant_pool[ ] A table of variable length structures representing various constants such as class names, literal strings, field names and types, referenced by other parts of the class file. The constant pool will be discussed in greater detail in section 2.5.3.

access_flags Two bytes containing flags for the class defined in the class file. The access flags are specified in table 2.2

this_class Two bytes containing an index to a constant pool C_Class entry leading to the fully qualified name of the class in the class file.

super_class Two bytes containing an index to a constant pool C_Class entry leading to the fully qualified name of the superclass of the class in the class file.

interfaces_count Two bytes containing the number of interfaces.

interfaces[ ] An array of indexes to a constant pool C_Class entry leading to the names of each of the Interfaces this class implements.

fields_count Two bytes containing the number of fields.

fields[] A table of variable length field_info structures representing the class and instance variables of the class. The field_info structure gives details of the accessability of the variable, name and type for all variables as well as value for static variables.

methods_count Two bytes containing the number of methods.

methods[ ] A table of variable length method_info structures representing the class and instance methods of the class. Only methods that are explicitly defined by this class are included, methods that are inherited are not stored here. The method_info structure contains information such as name, a descriptor detailing the return type and argument list, a table of exceptions caught by the method and the bytecode sequence that implements the method. The method_info structure is described in section 2.5.4.

attributes_count Two bytes containing the number of attributes (zero or more attributes are allowed in the attribute section of the class file)

attributes[ ] A table of variable length attribute structures. The current JVM spec (1.0.2) only defines one attribute, the SourceFile attribute, which gives the name of the sourcefile from which this class file was compiled. It is possible for compiler writers and JVM implementers to define their own, proprietary, attributes since the JVM is specified to ignore attributes it cannot understand.

As evident by the above description the class file is somewhat complex. A more graphical summary of the structures in the class file can be found at the end of this chapter in figure 2.5 and should prove to be helpful when studying the structure of the class file .

2.5.2 Access flags

The access flags are used to describe characteristics of classes, interfaces, fields and methods. Table 2.2 lists the flags. This table is an adaption of tables 4.1, 4.3 and 4.4 in

Lindholm,Yellin [17]. The abbreviations in the Used by column are C=Class, I=Interface, F=Field, M=Method.

Flag Name	Value	Description	Used by
ACC_PUBLIC	0x0001	May be accessed outside its package	C, I, F, M
ACC_PRIVATE	0x0002	Usable only in the defining class	F, M
ACC_PROTECTED	0x0004	May be accessed within subclasses	F, M
ACC_STATIC	0x0008	Is static	F, M
ACC_FINAL	0x0010	No overriding is allowed	C, F, M
ACC_SYNCHRONIZED	0x0020	Wrap use in monitor lock	C*, I*, M
ACC_VOLATILE	0x0040	Cannot be cached	F
ACC_TRANSIENT	0x0080	Non persistant	F
ACC_NATIVE	0x0100	Method not implemented in Java	M
ACC_INTERFACE	0x0200	Is an interface	I
ACC_ABSTRACT	0x0400	No implementation is provided	C, I, M

Table 2.2: Access and modifier flags

* For Classes and Interfaces the value 0x0020 represents the flag ACC_SUPER which indicates that superclass methods are to be treated specially in the JVM instruction *invoke-special*. The reason for this dual meaning for this value cannot be found in the literature.

2.5.3 Constant Pool

The constant pool is a table of variable length entries containing literal information as well as entries linking methods and fields to a particular class. The type of any entry is given by the first byte of that entry, the tag byte, which contains an unsigned value. The values inside the parentheses in the tag byte field in figure 2.4 give the tag value for every type of entry. The entries in the constant pool can be characterized as either primitive, containing a literal value or composite, containing one ore more references to other constant pool entries. References to the entries in the constant pool can hence be found both inside as well as outside of the constant pool. Figure 2.4 shows the different types of entries and their internal references.

The different types of constant pool entries are:

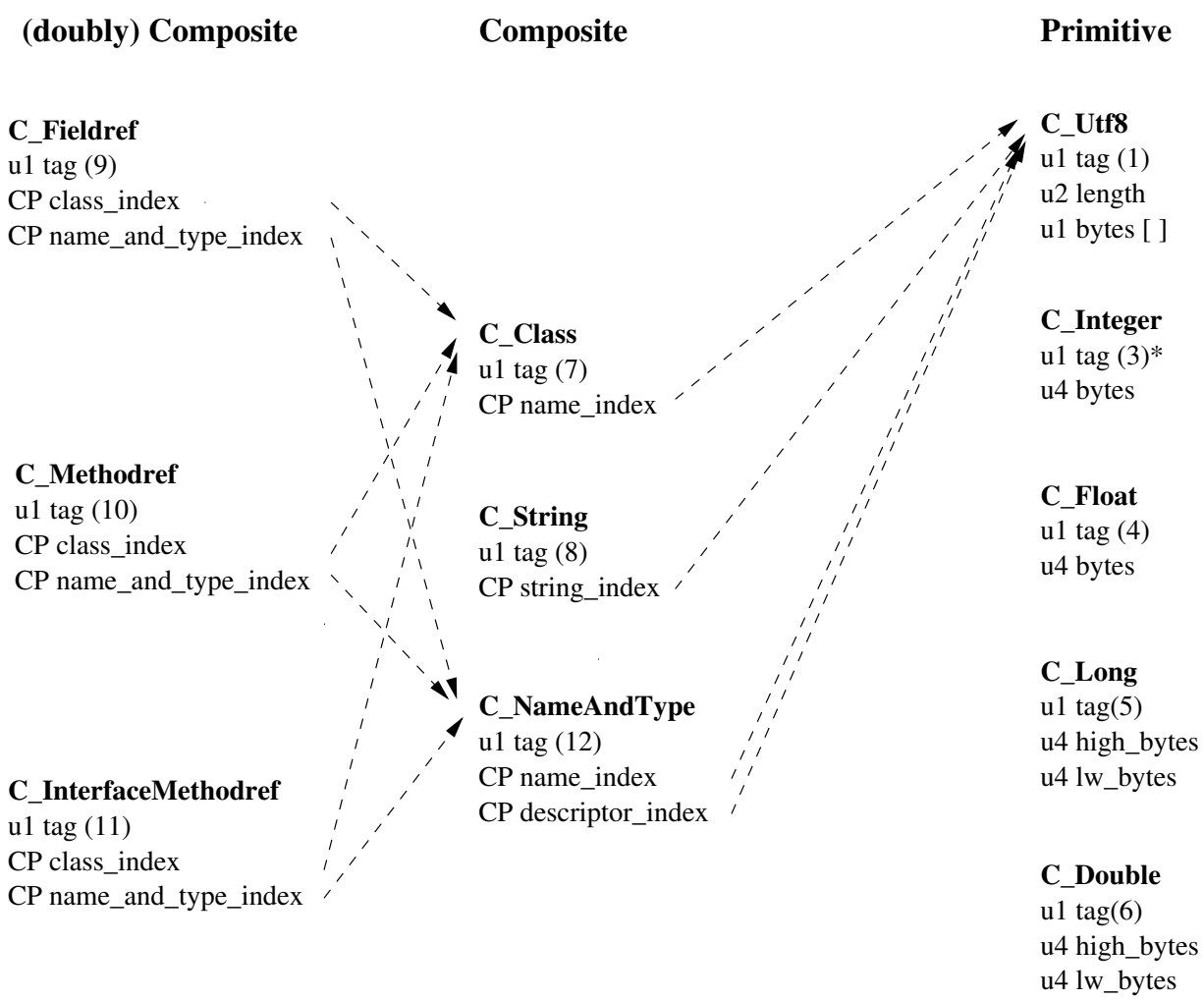

Figure 2.4: Constant pool entry types

Primitive Entries

C_Utf8 Holds a literal string in Unicode UTF-8 format. Literal strings is much used by the JVM since the JVM uses symbolic references to classes, fields and methods in order to achieve dynamic (late) binding.

C_Integer Holds an integer value.

C_Float Holds a float values.

C_Long Holds a long value.

C_Double Holds a double value.

Composite Entries

C_Class Holds an index to a C_Utf8 entry representing a fully qualified Java class name.

C_String Holds an index to a C_Utf8 entry representing a sequence of characters to which a java.lang.String Object is to be initialized.

C_Fieldref Holds an index to a C_Class entry representing the class that a field belongs to. Also holds an index to a NameAndType entry giving the name and type description for the field.

C_Methodref Holds an index to a C_Class entry representing the class that a method belongs to. Also holds an index to a NameAndType entry giving the name as well as a return type and argument list description for the method.

C_IntefaceMethodref Holds an index to a C_Class entry representing the interface that a declares a method. Also holds an index to a NameAndType entry giving the name as well as a return type and argument list description for the method.

C_NameAndType Holds an index to a C_Utf8 entry representing a field or method name (not fully qualified). Also holds an index to a C_Utf8 entry that contains a description of either the type of a field or the return type and argument list of a method. The details of how to interpret this string is given in section 4.3 of Lindholm, Yellin [17].

For the curious: tag number 2 existed in JVM specification 1.0 Beta DRAFT as C_Unicode with exactly the same format as C_Utf8. Sun then decided to remove it from the specification.

2.5.4 Methods

Information about the methods of a class are stored in one variable length method_info entry for each method in the class file. The structure of the method_info structure is not very complicated, but since it contains two attributes, one Code_attribute and one Exceptions_attribute, the total structure is somewhat complex. A good overview of the method_info structure can be found in fig 2.5. This section will give some detail on the involved structures: method_info, Code_attribute and Exceptions_attribute.

method_info structure

The structure of a method_info entry is as follows:

access_flags Two bytes containing flags according to table 2.2 for specifying access rights and other properties of the method.

name_index Two bytes containing an index to a constant pool entry giving the simple, not fully qualified name, of the method.

descriptor_index Two bytes containing an index to a constant pool entry giving the return type and argument list of the method.

attributes_count Two bytes containing the number of attributes.

attributes[] A table containing variable length attributes. For the current version of the JVM (1.0.2) only Code_attributes and Exceptions attribute are defined. These must exist exactly once for every method.

Code_attribute

The Code_attribute contains fields with information that the JVM uses when it is to execute a method. These name of these fields are found in figure 2.5 under the method_info heading.

The most obvious of these are of course the bytecode array which contains the JVM instructions for the method. In addition to this, the Java compiler counts the maximum number of words that can be placed on the operand stack by this method as well as the number of local variables (including the method parameters) used. This means that the required size of a method activation record/stack frame (see section 2.3.1) can be statically determined.

Also stored in the Code_attribute is an exception table that gives the details of how exception handling is to be performed in the method, i.e which exceptions will be dealt with where and how.

The Code_attribute can in turn have attributes of its own, currently only two attributes containing debug information is defined as possible attributes.

Exceptions_attribute

The Exceptions_attribute contains information on which exceptions this method can throw. This attributes main component is an array of indexes to constant pool entries specifying the class of the exceptions that can be thrown from this method.

2.5.5 Class file summary

The structure of the class file is presented in a compact form in fig 2.5. A few comments on the figure:

- u1 means that the value is stored as an unsigned integer in one byte.
- u2 means that the value is stored as an unsigned integer in two bytes.
- u4 means that the value is stored as an unsigned integer in four bytes.
- CP refers to an u2 holding an index to a constant pool entry.

- (o) means that an attribute is optional and that other, proprietary, attributes are allowed after the optional attribute. Attributes without (o) must exist exactly once.

2.6 Summary

This chapter has described the major features of the Java Virtual Machine (JVM). It is a stack machine with a fairly set of primitive data types, all signed, and a reference type used to refer to objects. Pointers do not exist. In order to execute any code, an implementation of the JVM must have some run-time data areas. The exact layout of these runtime data areas are not specified, but any Java execution system will typically need the regular activation records (frames) based layout with a heap for object allocation. The instruction set reflects the fact the JVM is a stack machine. In addition to stack manipulation instructions and standard arithmetic and control instructions the instruction set also includes instructions for arrays, objects, monitors and exceptions. The bytecode to be executed in a Java execution system is stored in a somewhat complex .class file containing a considerable amount of extra information besides the bytecodes.

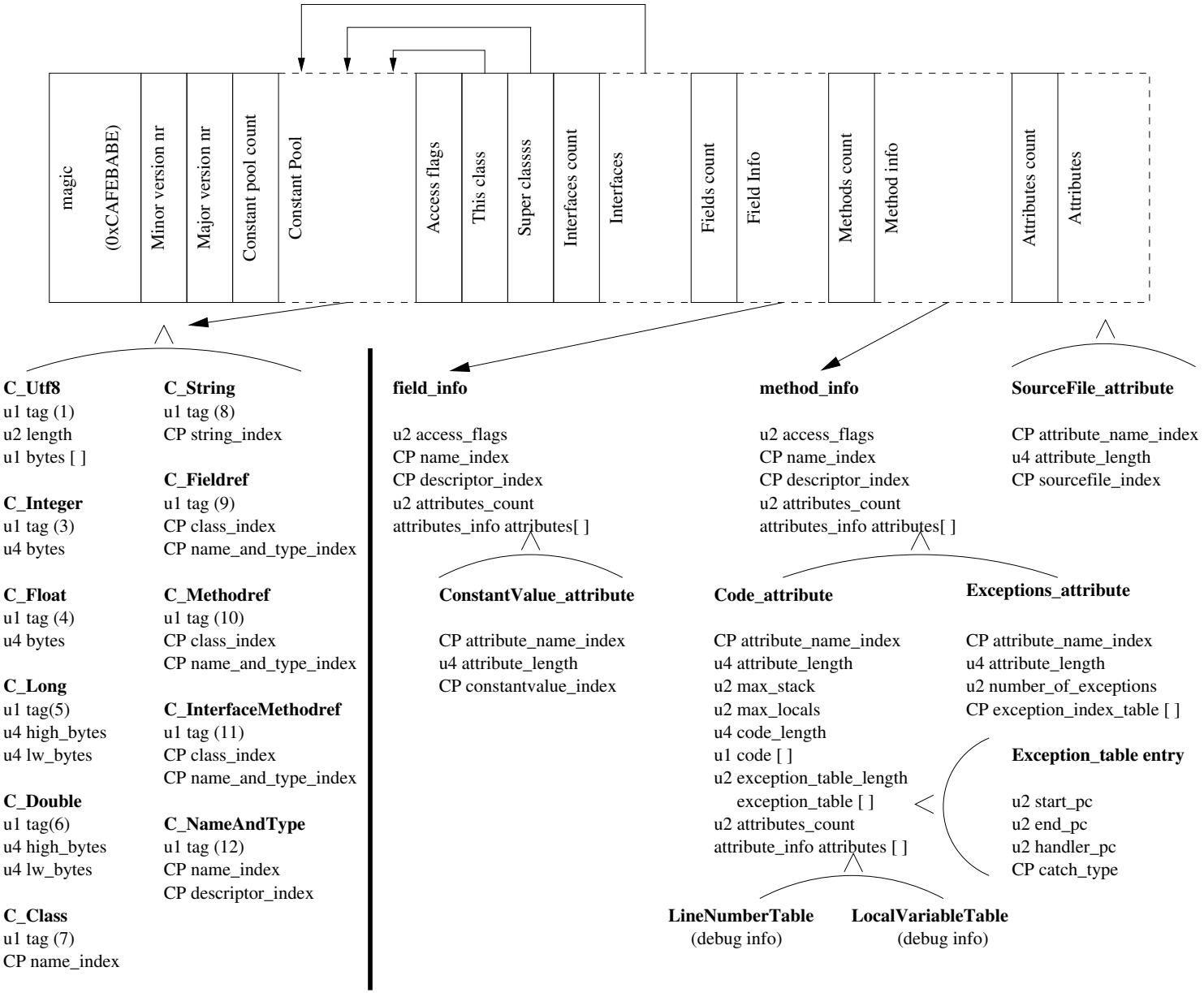

Figure 2.5: Java class file structure

Chapter 3

APZ 212 Description

3.1 Overview

This chapter describes the target system of this study, the data processing system APZ 212. After an introductory section which provides a context to the later sections, the part of most concern for this study, the Central Processor System is presented. First, a description of the logical structure is given followed by an overview of the hardware system. Finally, the instruction processing unit with its registers and instruction set is discussed.

Since this chapter contains many new terms which stem from Ericsson terminology, relevant terms are *italiziced* the first time they are used to show that they are previously not discussed.

3.2 Introduction

The APZ 212 system is the controlling part of some telecommunication switches manufactured by Ericsson. Since the APZ 212 is designed for this specific application, its architecture has by necessity to reflect the demands made on the system by the application. This leads to an architecture that has some differences compared to a generic

microprocessor-based system.

A switch in a telephone network typically needs to perform many similar time-constrained tasks in parallel with high reliability. An example would be a Central Subscriber Stage(CSS) which must handle a large number of uncomplicated subscriber operations. In order to efficiently perform multiple tasks, the switch is constructed with a *regional processor system* (RPS) containing many *regional processor*s (RP) which can handle routine tasks. More complicated tasks are performed by the *central processor system* (CPS). Since the regional processors are specialized in performing switch-related operations, they cannot be used efficiently for performing general program operations. This makes the regional processors unsuitable as a target for use by the Java execution system. The regional processor system is hence not examined further in this chapter.

The regional processors communicate with the central processor via *signals,* a central concept in the software structure that will be presented in section 3.3.3. The concept of signals has no connection to analog signals or DSP[1], it is more akin to the concept of messages in the object oriented world.

As shown in figure 3.1 the RPB and CPS are duplicated. This is done for reasons of service reliability, and this parallelism cannot be used for improving performance. Figure 3.1 also shows that the CPS can be divided into the following parts:

- *Central processing unit* (CPU). The CPU is further subdivided into
  - *Instruction processor* (IPU). The IPU is used for actual instruction execution.
  - *Signal processor* (SPU). The SPU handles job administration such as signal buffer handling, prioritizing signals and transfer of signals.
- *Program storage* (PS). The program storage is used to store the program code and tables related to the distribution of signals. The PS and the DRS will be further explained in section 3.4.1.

[1]Digital Signal Processing

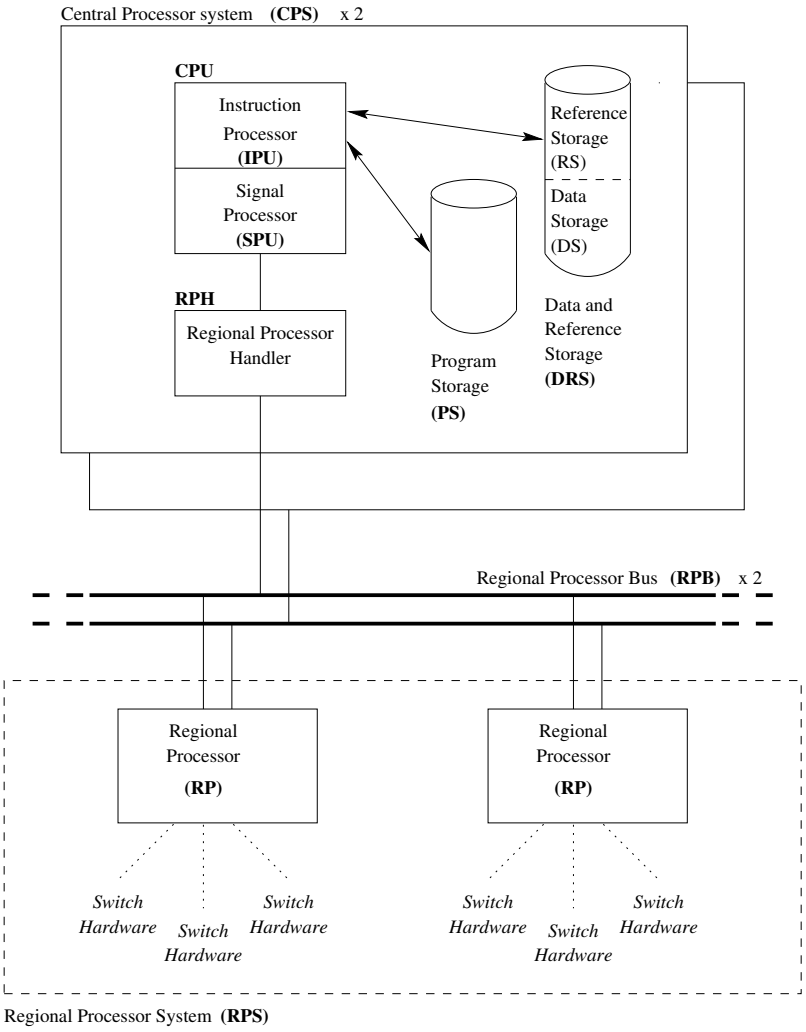

Figure 3.1: APZ structure

- *Data and reference storage* (DRS). This physical store is logically divided into
  - *Data storage* (DS). The DS is used to store data for variable blocks.
  - *Reference storage* (RS). The RS is used to store tables containing information about the system.
- *Regional Processor handler* (RPH). The RPH handles the interface to possibly multiple RP *busses*.

The system is designed so that program blocks as well as data are relocatable at run-time. No absolute addresses are used, instead all referencing is done by using tables.

3.3 Logical Structure

3.3.1 Program structure

This section presents the software structure of the CPS. The software structure of the CPS is part of the system hierarchy of the APZ as described section 1 of [8]. The CPS functionality required by a specific function block is provided by a CP program-unit.

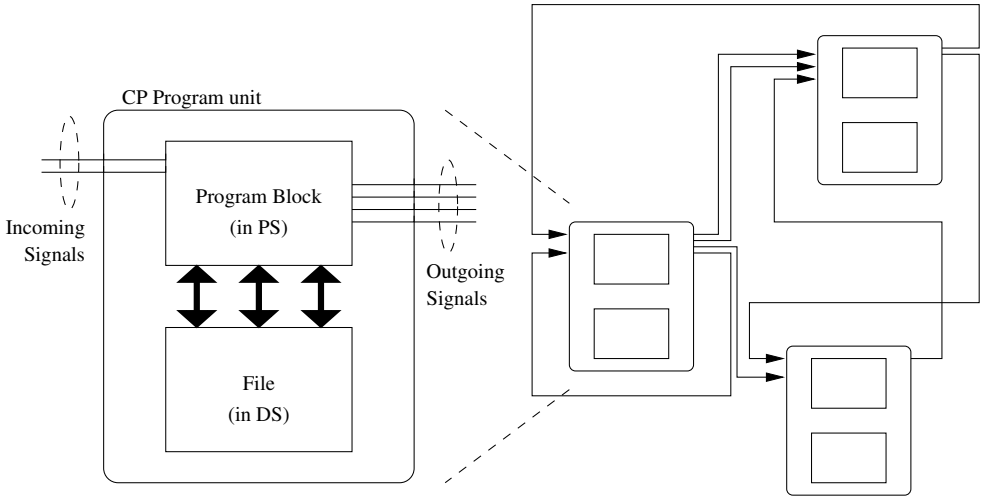

Figure 3.2: CP-program unit

A CP program-unit contains instructions(one program block) and data(one or more files). All software in the system is built up around program blocks, self-contained code parts which have access to their own data only and communicate with each other by means of signals. Signals are the chief method of inter-block communication and will be further examined in section 3.3.3. Figure 3.2 illustrates the logical structure of program blocks.

3.3.2 Data structure

The data belonging to CP program-unit is organized in one or more files. These files contain one or more *records*. Each record contains one or more *variables*. Each variable can be indexed, i.e. be an *array variable* and it can be split into several subvariables. All data is accessed by means of tables, which means that data is relocatable in memory at run-time. Variables can have a number of different sizes according to table 3.1. The smaller variables in the table are packed in order to achieve maximum space efficiency.

Length (in bits)	Abbrev.	Name
1	B	Bit
2	T	Bit-pair
4	C	Character
8	H	Half-word
16	W	Word
32	D	Double-word
64	Q	Four-word
128	O	Eight-word

Table 3.1: Variable Sizes

The file is a logical unit, and a file need not be contiguously allocated in memory. A variable however, is a physical entity and is continually allocated in memory. The address calculations necessary to obtain the absolute address of a specific instance of a (sub)variable is performed by hardware (or microcode) and need not concern the programmer. An example of the above concepts is illustrated in fig 3.3.

Figure 3.3 illustrates a file with p number of records. This implies that there are also p instances of variables AVAR, BVAR, CVAR and DVAR respectively.

These variables are of different types:

- AVAR is a plain variable.
- BVAR is a composite variable which contains four subvariables. Subvariables can have the the sizes of 1,2,4,8 or 16 bits. All subvariables in a variable have the same

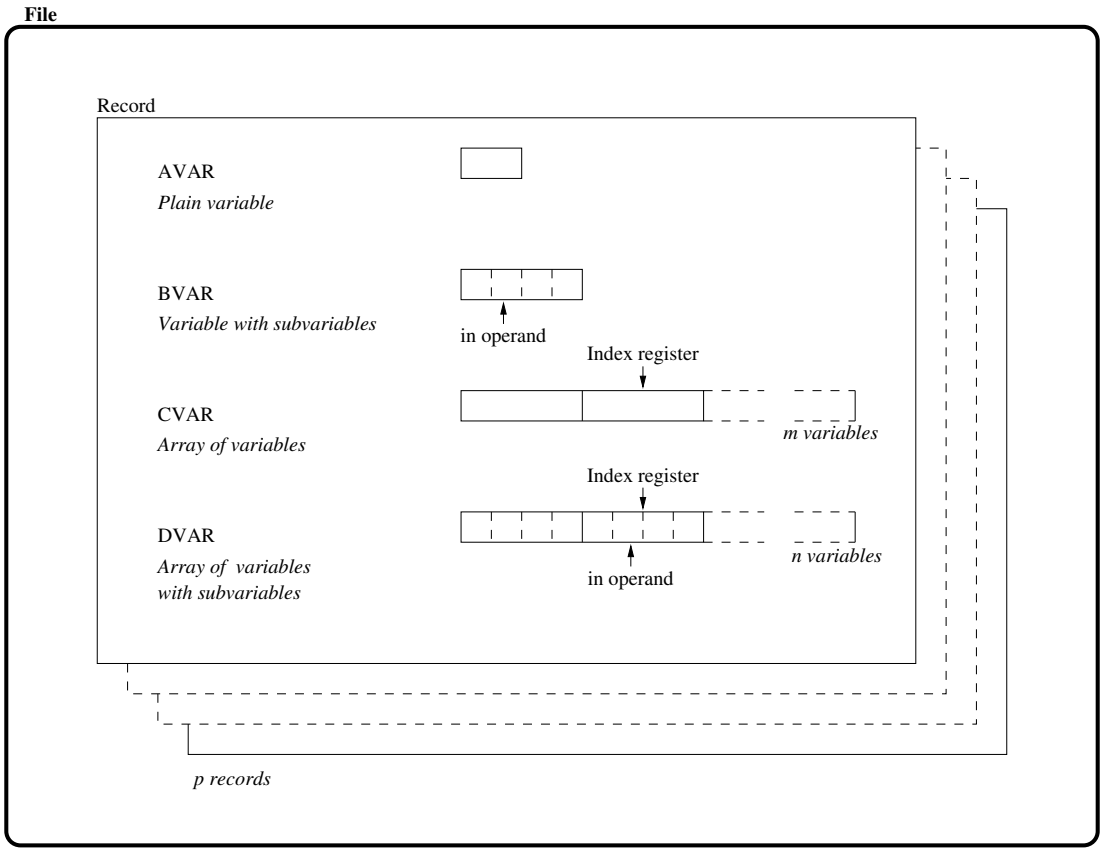

Figure 3.3: Logical Data Structure

size. The subvariable to be accessed is given as an operand value to the machine instruction processing the variable.

- CVAR is an array of variables. The indexed variable to be accessed is given by the *index register* (IR) in the processor.

- DVAR is an array of composite variables. The IR is used to determine which index variable and an operand is used to determine which subvariable part of the variable is to be accessed.

From the above we can draw the conclusion that a maximum of four determinants must be used to uniquely identify a value in a variable:

1. Variable name. Determines which variable to access.

2. *Pointer register* (PR0). Determines which instance of the variable (which record) to access.

3. Index register (IR). Determines which indexed variable to access if the variable is an array variable.

4. Operand value. An operand value given to the data processing instruction specifies which subvariable to access if the variable is composite.

3.3.3 Signals

Signals are the means by which program blocks communicate. Signals are sent from one program block to another, or from one program block to itself. Signals can carry data, the data is placed in the pointer register (PR0) and *signal data registers* (DR0-DR23) (see section 3.5.1 on registers). Signals have a format which specifies how many signal data registers is used.

When programming, signals are identified with a name, but this is mapped to a number and this number is then percolated through some tables as will be shown in section 3.4.4. This is done in order to achieve the same storage independence (re-allocatability) as the program blocks and data blocks.

There are several types of signals. One type carries information between program blocks in the central processor (a CP-CP signal). Other signals are concerned with the flow of information between central processor blocks and regional processor blocks. Since this study is limited to the central processor, all signals discussed henceforth will refer to the CP-CP signal type. In addition to the types of signals there are several more categorizations of signals.

Direct and delayed signals

Signals can be either *direct* or *delayed*. The delay can be either a fixed time or based on priorities and is implemented by buffers in the signal processing unit, SPU. Further details of delays and prioritizing can be found in section 2.3.1 and 3.4 of [8]. Direct signals will cause execution to continue in the program block that receives the signal. Execution in the sending block will be frozen until the receiving program block has finished executing. If execution is to be continued in the sending block, the signal must be sent as a delayed signal via job buffers in the SPU where it will be subject to prioritizing and delays.

Single and combined signals

A further categorization of signals is into *single signals* and *combined signals*:

Single signals are sent to a program block to trigger an activity in that block. A single signal never returns.

Combined signals can only be direct. They are used when then the receiving program block function will return some data. The return is performed with what is called a *backwards combined signal* and the execution is then continued in the sending block. The backwards combined signal can carry data just as the forward. Only the block number and address is stored on the link register stack when a combined signal is sent, no processor registers are preserved. Combined signals can also be referred to as being *linked*.

Unique and multiple signals

Signals can also be either *unique* or *multiple*:

Unique signals have only one program block as possible recipient.

Multiple signals have many program blocks as possible recipients. Each instance of a multiple signal can however only be received by a single program block. When sending a multiple signal, a block reference specifying the receiving program block is given in a processor register. A multiple signal can also be referred to as being *indirect*.

3.4 Hardware structure

This section will discuss the hardware structures and tables related to program blocks, data blocks and signals. First an overview of the memory layout is presented.

3.4.1 Memory overview

The memory subsystem of the APZ is divided into two physical parts: Program Storage (PS) and Data and Reference Storage (DRS). The DRS is logically subdivided into Reference Storage (RS) and Data Storage (DS). The Program storage is partly cached in order to improve performance. The cache resides in the Program Storage Cache Memory (PSCM). The memory contents of the different stores can in general be categorized as follows:

- Program Storage

  Contains program blocks (i.e. code) both for the Central Processor (CP) and Regional Processors (RP). Also contains two tables for multiple signals (see section 3.4.4) and a transport area used for temporary storage during reallocations.

- Program Storage Cache Memory

  Contains frequently accessed CP program blocks. Also contains two tables (GSDT-U and GSDT-M) used for global signal routing,(see section 3.4.4) and one (RTC) used to hold a reduced variant of the reference table.

- Reference Storage

  Contains OS information tables and free address tables for PS, RS and DS. Also contains various tables used to store information about the program blocks. The reference storage also contains a transport area used for temporary storage during reallocations.

- Data Storage

  The data storage contains data stored in variable blocks. Also contains a storage bank area used for dynamic allocation of storage and a transport area.

A graphical example of the memory layout is given in figure 3.5 at the end of this chapter. Both the PS and DRS are physically 32 bits wide, but addresses in PS program blocks are counted in 16-bit words from the program block start address.

3.4.2 Program blocks

Program blocks contain the actual code. Program blocks are either stored in the regular program storage (PS) or in the cached part (PSCM). Only CP program blocks can be stored in PSCM.

Structure of a program block

A program block consists of the following parts:

- Identification word.
- Product identity. Max 32 ISO characters used to identify the block.
- *Signal distribution table* (SDT). Maps *local signal number*s to an address in the program code.
- *Signal sending table* (SST). Maps local *signal sending pointer* to *global signal numbers.* Section 3.4.4 covers signals and the SDT and SST tables.

- Program code. The machine code to execute.
- *Correction area.* Is used for temporary corrections to the code. Signals may be redirected to code in the correction area for temporary corrections or testing purposes. The correction area is optional.

A program block is limited in size to 32768 16-bit words.

Reference table

The reference table is used to keep information about the program blocks. Among the information stored are:

- Program block name. Stored with max 7 ISO chars.
- Size. The size of the program block.
- Start address. The address at which the program block is stored. A flag indicates whether it is stored in PS or PSCM.
- *Base Address Table* start. The location of the Base address table. The base address table stores information about the variables used by the program block. See next section.
- The number of different incoming and outgoing signals.
- Various flags and fields used to support tracing.

The first entry in the reference table is used to point to the operating system area in the reference storage. The reference table can store a maximum of 4096 items, which sets the maximum number of program blocks to 4095.

3.4.3 Variable blocks

Variable blocks are the physical representation of a variable. As shown in section 3.3.2, a variable can exist in several instances (one per record). Each instance can then be an array variable. The array variable could be an array of variables split into subvariables.. This structure is mapped onto the one dimensional memory space used to store the variable in DS.

Base address table

There is one base address table for each program block. The base address table contains information about all the variables used by this program block. Each program block can only locate its own base address table, and thus it can only access its own variables. Among the information stored in the base address table are:

- Variable length in bits. Possible variable lengths are given in table 3.1.
- Number of indexed variables in an array variable. The number of indexed variables can be 2^n where n =1...15.
- Number of records in data file. Specifies the number of variable instances. A flag bit is used to specify that there are multiple instances.
- Start address. Where in DS the variable starts.
- Data file number.
- Variable category. This field is used to store information on which kind of variable it is and what should happen during restart. The following options are given:
  - Bit 0: Dynamic buffer (STATIC)
  - Bit 1: Is reloaded during restart with reload (RELOAD)

  - Bit 2: Cleared during restart (CLEAR)
  - Bit 3: Preserved during restart (DUMP)
  - Bit 4: Permanent data. Data are not transfered from old to new block during function transfer. (STATIC)
  - Bit 5: Spare
  - Bit 6: Register marked. Variable is r-Declared, length 32 bits. (R)

- Various flags and fields used to support tracing and double writing. Double writing indicates that when writing to a variable, not only the value at the calculated variable address will be changed, but also a value at an address corresponding to the calculated address plus a value in a special register. Double writing is used during reallocations.

The first entry in the base address table is used to store the current program block number and during reallocations. The maximum number of remaining entries in the base address table is 4095, which is the maximum number of variables per program block. Just as performance critical program blocks are stored in a cache memory (PSCM), likewise the base address table for these time critical program blocks are stored in a simplified format in a separate cache memory (BAS).

Storage Bank

The storage bank is used for dynamic allocation. It comprises of several data files that can be allocated and deallocated by sending signals to OS function blocks. There is a lack of background material on this subject.

3.4.4 Signals

This section will describe the way signals are processed and distributed. First the program block signal tables will be studied. Then the global tables related to signals are presented. Lastly, the signal distribution for multiple signals are discussed.

Signals are the means of interwork between programs. Signals are sent and received by program blocks and data transfer occurs via the signal data registers. Delayed signals have the data copied from the registers in order to preserve the data while the signal is waiting in the buffer. The amount of data transferred in a signal is specified so that only the necessary registers are copied. Besides the pointer register PR0, 0-8,12,16,20 or 24 signal data registers may be used to hold data when sending a signal.

When discussing the distribution of signals, two terms are useful: *local signal number* (LSN) and *global signal number* (GSN). The LSN is used for identifying an incoming signal within a program block. The GSN is a globally unique number that identifies a signal outside the program block.

Program block signal tables

The program block has two signal tables:

Signal Distribution table (SDT): used to map a LSN to a code address in the codespace of the program block.

Signal Sending table (SST): used to map the signal sending pointer (SSP) to a global signal number (GSN). The SSP is given as an operand to a signal sending instruction.

Global signal tables

The system has two global signal distribution tables. The global signal distribution tables are:

GSDT-U. Global Signal Distribution Table for Unique signals. Unique signals only have one possible recipient block and this table maps an incoming GSN to a block number and a LSN.

GSDT-M. Global Signal Distribution Table for Multiple signals. Multiple signals have several possible recipient blocks and this table maps a hash index to a block number

and a LSN. The hash index is constructed by applying the hash function: receiving block number XOR GSN.

Multiple signals collision tables

When using the GSDT-M two different signals may result in the same hash index when the hash function is applied. To solve this, two tables are used. The Collision table start address table (CTSAT) is indexed by the hash index and points to the start of collision list in the collision table. The collision list is searched for the corresponding receiving program block number and the LSN is then retrieved from the list.

Signal Distribution example

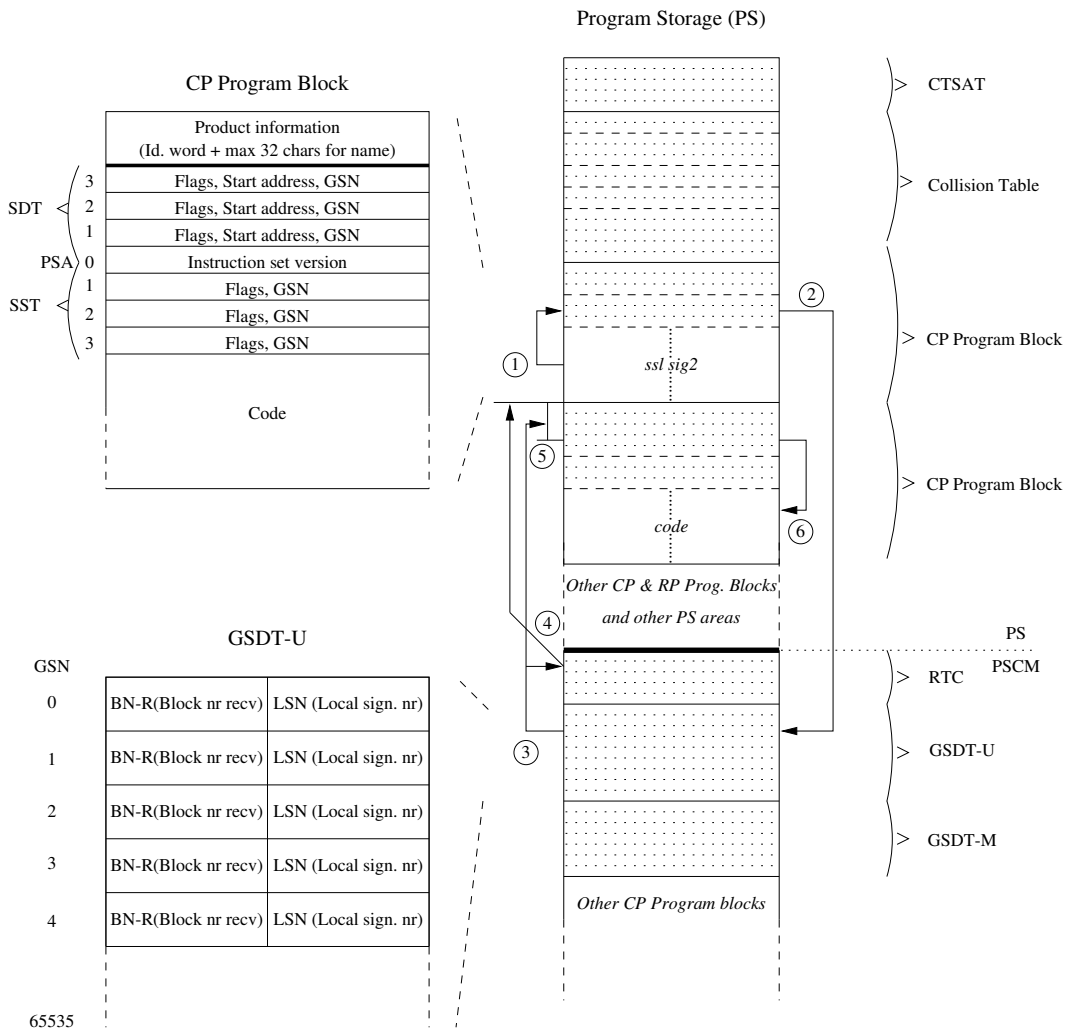

Figure 3.4: Signal Distribution example

Example of unique signal

1. An ssl instruction in the code sends a unique, linked and direct signal. The operand sig2 gives a signal sending pointer (SSP) which points to an entry in the signal sending table (SST).

2. The signal Global Signal Number (GSN) is fetched from the SST and used to access

an entry in the Global Signal Distribution Table for unique signals (GSDT-U).

3. From the GSDT-U a block number and a local signal number(LSN) is fetched.

4. The block number is used to find the receiving program blocks start address in the reference table cache (RTC).

5. The program block start address from the RTC and the Local signal number from the GSDT-U is used to find the correct entry in the signal distribution table(SDT) of the program block.

6. The SDT entry holds an code offset from the start of the block. Transfer of execution is made to this code, which is the code which is to respond to this signal.

3.5 Instruction processor (IPU)

3.5.1 Registers

The processor has the following registers:

- Index register (IR). Selects which variable in an array variable is to be accessed.
- Pointer register (PR0). Selects which record is to be used when accessing variables.
- Extra pointer register (PR1).
- Comparison register (CR). Used by comparing instructions (conditional jumps).
- Signal transmission register 1 (SR1).
- Signal transmission register 2 (SR2).
- Signal data registers (DR0-DR23). Used for data transfer in signals.

- Arithmetic registers (AR0-AR3). Used for arithmetic operations.
- Other process registers (WR0-WR29). General use registers.

These 64 registers are 32 bits wide for the APZ system studied (APZ212). Numeric values are stored in an unsigned format.

There are 24 registers(DR0-DR23) which can be used to transport information in signals. The pointer register also retains its data, the other registers are not guaranteed to hold their value when the signal reaches its destination.

3.5.2 Instruction Set

The instruction set of the APZ is adapted to its specific architecture. Instructions for dealing with the unusual data structure is provided for example. This section is based on the information in [3]. Below is a functional grouping of the assembler instruction with some comments:

1. Reading and writing from/to store.

   These instructions support the data structure that the APZ uses. For example, instructions for reading a variable instance from an array variable in a specified record are provided. Some of these instructions provide transfer of more than one value.

2. Register instructions.

   These instructions are used for moving data between registers and for loading constant values to registers.

3. Arithmetic instructions

   Addition, subtraction, multiplication and division instructions are available. Addition and subtraction can be performed register to register, constant to register or

constant to store. Multiplication and division can only be performed register to register.

4. Logical instructions

   These instructions are used for shifts, rotations, logical and, or, xor functions.

5. Local control transfer

   Unconditional jumps, subroutine(linked) jumps and conditional jumps are provided. Also zero-start indexed table jump instructions are available with both a register or a stored variable used as index value.

6. Signal transmission instructions

   Several signal sending instructions exist although many are not relevant for this study; the signals that deal with regional processors and time queues for example. The signal sending instructions of most interest are the linked, which store the calling block and a return address on a link register stack. The end of program instruction is used to return to the return address of the calling block.

7. Search instructions

   These instructions search for set bits in a register or for a value in a variable which exists in several instances. A string comparison instruction is also present.

8. OS-instructions

   Many OS instructions exist. These are instructions for handling the system tables, direct writing/reading in stores and other system functions.

9. Macro instructions

   This instruction category represents instructions that are made available by the assembler, and the instructions in this category range from simple aliases for other instructions, label defining instructions which generate no code, to instructions which extract information from system tables.

3.6 Summary

After an introduction describing the distributed nature of a telecommunications switch, this chapter has presented the logical and hardware-level structure of the central processor system of APZ 212. Logically, programs are divided into a number of program units which contains both program code (in a program block) and data (in 'files'). The files can contain several 'records' which each can contain several variables, arrays of variables or variables with subvariables. Communication between program blocks is performed by sending signals, which exists in several variations. On the hardware implementation level, the Program Storage and the Data and Reference Storage provide storage for program blocks and files, respectively. Both storages uses a number of tables to provide reallocatability and flexibility. The signal sending mechanism, which also is a heavy user of tables, can transport up to 24 32-bit values in the signal data registers. The assembler instruction set used for the APZ provides, in addition to standard arithmetic and control instructions , a number of instructions supporting the hardware features, such as signal sending instructions.

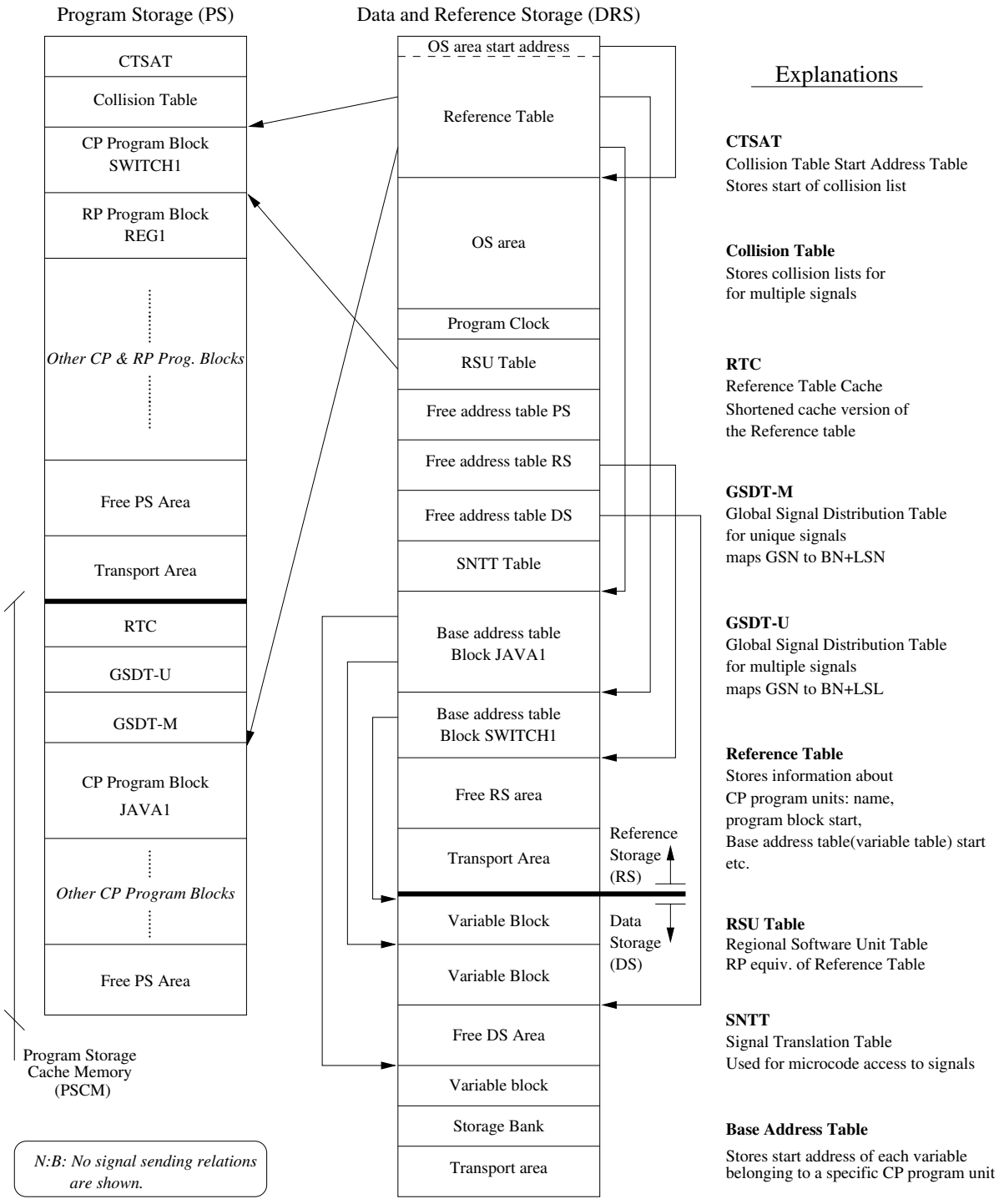

Figure 3.5: Memory Layout

Chapter 4

A mapping of the Java Virtual Machine to APZ 212

This chapter studies the possible mappings from the Java Virtual Machine specification to the APZ. This chapter is divided into problem subgroups that are discussed individually. The subgroups are:

1. Execution model.
   Possible layouts of the execution system is presented. Whether to use an interpreting or compiling execution system is discussed.

2. Object handling.
   How to implement the structures needed to cater for the object model of Java is discussed. Specifically the following areas are covered:

   (a) Method Inheritance, both for instance and static (class) methods.

   (b) Variable Inheritance, both for instance and static (class) variables.

   (c) Dynamic Method selection, how to select the correct method based on object type.

3. Memory handling.

   How the memory model that is implicit in the Java VM can be mapped on the APZ's non-standard memory architecture as well as methods for garbage collection are discussed.

4. Activation records.

   Discusses how the functionality of the Java Stacks can be mapped to the APZ where no stack or heap is present.

5. Multithreading.

   Covers the implementation of multithreading and the adjacent topic of method synchronization. Also the yielding made necessary by the hardware architecture is discussed.

6. Exception handling.

   How to implement the functionality for exception based error handling is discussed.

7. Number representation.

   Different strategies for adopting the unsigned format of the APZ to the signed format of the JVM is covered.

8. Library support.

   What standard Java packages need to be included and how they can map to the APZ system is discussed.

9. Instruction mapping.

   The similarities and how to overcome the dissimilarities between the JVM instruction set and the APZ instruction set is covered.

Some of the above problem subgroups present multiple solutions to a problem. Where the selection of one solution is not essential for the continued presentation, the selection is deferred to the next chapter.

4.1 Execution model

Since one prime design goal of Java was to achieve portability, Java uses an execution model based on platform independent bytecodes stored in a classfile as explained in section 2.5. By basing the APZ's Java execution system on bytecodes instead of a Java sourcecode several benefits are gained:

- Ability to use state-of-the-art development tools for producing the bytecodes.
- Enhanced ability to buy components from other vendors.

The bytecodes must by some means be transformed into native instructions for the specific hardware platform that the Java runtime system run on. This can be done in several ways, mainly differentiating in the time when the translation from bytecodes to native instructions occur as illustrated in fig 4.1. Three variants will be discussed in this section along with the consequences for a possible APZ implementation.

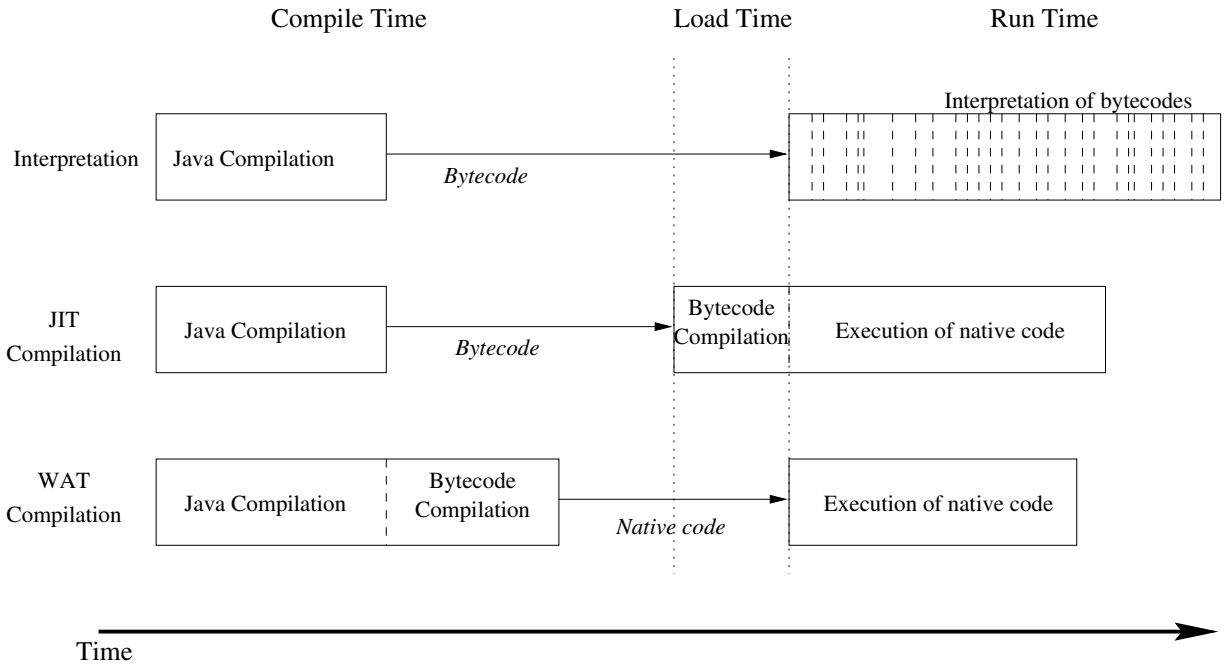

Figure 4.1: Execution models

4.1.1 Interpretation

The first implementations of a Java runtime system used the interpreting model for execution of bytecodes. Under this model the program is stored as bytecodes in memory. As the program executes, the interpreter evaluates the bytecode to be performed and executes the native instructions specified for the bytecode. This means that every bytecode is evaluated every time it is to be executed.
Advantages:

- Easy to implement. An implementation based on an interpreter is relatively easy to construct. This is true on generic platforms that use standard memory access schemes. As seen below this is not true for the APZ.
- Dynamic loading of classes is possible. Some applications dynamically load classes at runtime. This is easily accustomed for in an interpreted environment.
- Platform independence. The same bytecode-based software can be executed on any platform which has a Java execution system.

Disadvantages:

- Speed. The interpreting model is slow since it incurs a large amount of redundancy in the bytecode to native code translation process. In some applications this is not critical, but for the applications domain targeted by this study this speed decrease is clearly unsuitable.

An example of an interpreting execution system is Sun's JDK 1.1 execution system.

4.1.2 Just In Time (JIT) Compiling

Just in time(JIT) compiling was conceived when the speed of execution for interpretation based execution systems proved to be inadequate for some applications. The JIT model

uses select compilation just before runtime to improve the performance. As a JIT based execution system loads an application it compiles some or all classes based on some analysis. As compilation occurs during load time, JIT compilers typically tries to minimize the time for compilation resulting in minimal optimizations. Also most kinds of global optimizations are impossible to perform because of the per-class compilation scheme.
Advantages:

- Faster than interpretation.
- Dynamic loading of classes is possible.
- Platform independence. Since compilation is performed transparently on the local system, the platform independent bytecodes are still used for distribution.

Disadvantages:

- More complex than an interpreting model. A JIT compiler contains both a compiler and an interpretator (in most cases).

Examples of JIT compilers can be found in the Java execution systems of Netscape Navigator and Internet Explorer. A possible evolution of JIT compilers is to let them compile/recompile classes as an background process, for example when waiting for user input.

4.1.3 Way Ahead of Time(WAT) Compiling

Way ahead of time (WAT) compiling is similar to the 'ordinary' concept of compiling. Current WAT compilers convert Java bytecodes to C language statements which is compiled to native instructions. The use of C as an intermediate representation is because the extensive optimizations made possible by modern C compilers. A WAT compiler for the APZ architecture cannot reap similar benefit because the lack of optimizing C compilers for the APZ environment, instead a WAT compiler is proposed to generate native instructions

directly from the bytecodes. It is still possible to perform substantial optimizations if deemed necessary.

Advantages:

- Speed of execution. No interpretation overhead is present and the level of optimization can be adapted to the need of the application.

Disadvantages:

- Dynamic loading of classes is not possible.
- Platform independence is lost to some degree. When using APZ-specific binaries instead of bytecode 'binaries' the platform independence is lost. The bytecodes used to generate the APZ-specific binaries are however still platform independent.

Examples of Java WAT compilers which generate C-code are Toba [22] and Harissa [20].

4.1.4 Recommended execution model

Applications for telecommunication switches typically differs somewhat from regular Java applications. Software for telecommunication switches does not need to retain platform independence to the same degree as applications targeted for distribution over a heterogenous network such as the Internet. Speed is more relevant since telecommunications applications are in greater need of efficient execution than the normal applications. Using WAT compilation seems to be the most appropriate execution model since it provides the highest execution speed and the drawbacks are manageable.

4.2 Object handling

The structures and mechanism used to support the object model of Java as presented in section A.3.2 is discussed in this section. How to represent objects, implement variable and method inheritance and the overall runtime framework are covered.

4.2.1 Runtime framework

The runtime framework is based around an *executive program block* which performs the bookkeeping related to the execution system. This includes functions for the following:

- Object creation and allocation.
- Garbage collection.
- Dynamic method selection.
- Exception handling.

The detailed functioning of the executive block is explained in the individual sections below.

4.2.2 Object representation

The representation of Java objects in the APZ runtime system are proposed to be as follows:

- A class corresponds to a function block (i.e. a program block with zero or more associated files).
- The method code is represented in the program block where each method is executed by sending a direct, combined(linked) signal.
- Instance variables are stored in a file associated with the program block. If the number of objects exceed the number of records in the file, a dynamic buffer can be allocated to the block in order to accommodate more object instances.
- Inheritance of instance variables is achieved by allocating space for the inherited variables in the subclass.

- Inheritance of class variables is performed by sharing the superclass' storage location for the class variable.
- Inheritance of methods is discussed later, see section 4.2.3.

To exemplify the above the following fragment of code is used as an illustration:

```
class A {
  int varA;
  int varB;
  static int classvarA;

  int setA (int newA) {
    varA=newA;
    return varA;
  }
}

class B extends A {
  int varC;

  int setA (int newA) {
    varC=varA=newA;
    return varC;
  }
}
```

The code defines the two classes A and B. Class B inherits the variables varA and varB from class A. Class B also inherits the method setA, which it however overrides with its

own method setA. Both class A and class B have access to the class variable classvarA which is common to all instances of either class A or class B (or any other subclass of A). The strategies available to class B for accessing classvarA are discussed in the paragraph on data sharing in section 4.2.3.

The above Java code is compiled to bytecodes and then a WAT compiler is used to produce APZ native code. The above code is illustrated in fig 4.2 which shows how the classes could be represented in memory. The figure shows how class variables are separately allocated from instance variables.

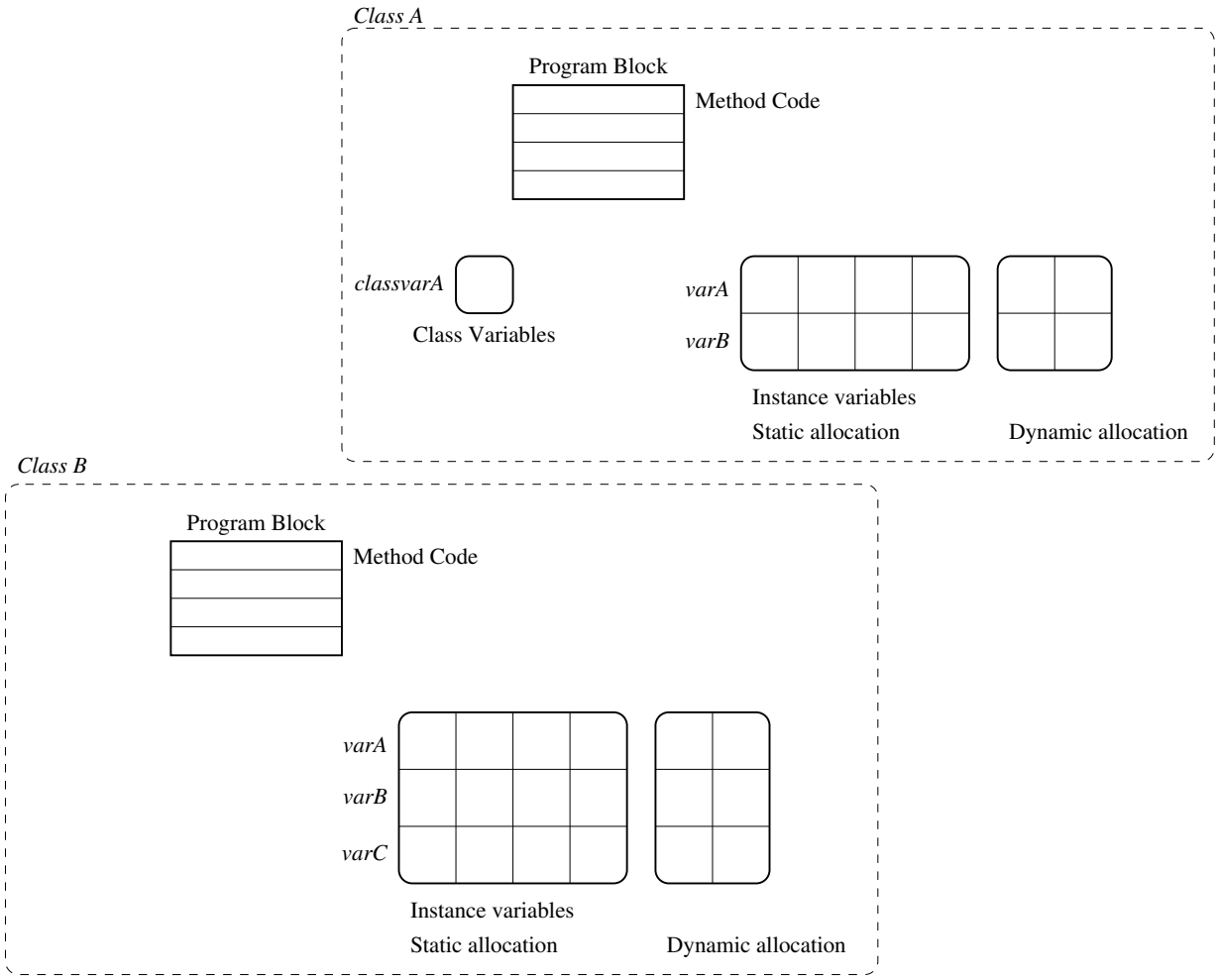

Figure 4.2: Object representation

Each object is uniquely identified by using 32 bits. The 32 bits are split into two 16 bit areas, with the 16 most significant bits representing the class of the object and the 16 least significant bits representing the instance of the object. This representation allows for a maximum of 65536 different classes and 65536 instances of objects for each class.

4.2.3 Method inheritance

In order to support method inheritance there must be some mechanism by which either code or data could be shared between different program blocks. The APZ is designed to disallow sharing of data between program blocks, so how to implement method inheritance becomes a problem. The possible strategies to achieve method inheritance can be divided into two groups:

1. Code copy.
   This implies that each inherited method is copied into each subclass program block. This leads to extensive duplication of code but also leads to immediate access to inherited variables for the inherited methods.

2. Data sharing.
   This implies that inherited variables are operated on by inherited methods that are physically a part of the program block of the superclass and as such are blocked from accessing the variables of a subclass program block by the APZ system. Different schemes exist for achieving the needed data sharing:

   (a) Patch in Base Address Table(BAT). This scheme involves sending a reference to the data that is to be accessed with the call and then manipulating the BAT so that the program block has access to the data. This however requires hardware changes to be possible and reduces the system robustness.

   (b) Copy the BAT. This scheme copies the BAT entries for the inherited variables

so that the superclass also has BAT entries for the the inherited variables in the subclasses. This needs support by the Operating System function blocks.

(c) By using inspectors. This scheme uses signals to access the inherited variables. This requires the signal numbers for the signals used to access the variables to be sent as parameters in addition to the original method calls parameters. This scheme uses a large amount of signal numbers.

(d) Signal sending without BAT change. This scheme changes the program block without changing the BAT reference. This means that the inherited method code in the superclass has access to the inherited variables in the subclass. Minor hardware changes are required to make this scheme possible.

4.2.4 Dynamic method selection

As shown in section 2.4.4 the Java virtual machine uses virtual methods to dynamically call the correct method based on the type of the object making the call. Since each object is identified with 16 bits representing the class of the object and 16 bits representing the instance of the object, every object reference also holds type information for the object. Dynamic method selection is done by sending a method invocation request signal to the executive block containing the object reference, method index and method parameters in addition to the local variables and stackstate which is already present in the signal dataregisters and which are to be retained in an activation record. The executive block creates a new activation record and sends a multiple, linked signal indicating the receiving class by using the object reference class identification part.

Using dynamic method selection is slightly more complex and takes more time than ordinary static method invocation. One method for minimizing the use of dynamic method selection is Class Hierarchy Analysis (CHA). Results reported by Muller, et al [20] show that between 18 to 40 percent of the dynamic method calls could be replaced by static calls as a result of CHA. A more complete description of CHA can be found in Dean, et al [9].

4.3 Memory handling

The Java virtual machine typically uses a heap to allocate objects, and a garbage collection routine to reuse memory that becomes unused. The APZ memory structure has no heap, so the required functionality must be provided by other means. As can be seen in figure 4.2 the objects are allocated in two parts of the memory, one statically allocated in a file and one dynamically allocated in the storage bank. Each object type must have an anticipated number of maximum number of concurrently existing objects associated with it. This number can either be calculated by performing code analysis on the Java code or by manually providing meta-information about the classes. The exact mechanisms for this needs to be investigated further to examine to which extent it is possible to by code analysis generate an estimate of the maximum number of concurrent objects of any given class. A discussion on meta architecture for Java can be found in Kleinöder, Golm [14].

If the statically allocated object space runs out for an object, it is possible for the executive block to create a dynamic buffer to hold further instances of the object. This can be regarded as a 'safety valve' if the estimation of the maximum number of objects should prove to be incorrect. Since all allocations are handled by the executive block, continuous monitoring of the available free store for each class can be performed. Dynamic buffers can also be used as a mechanism for objects which temporarily exist in large quantities and where it would be wasteful to statically allocate space for that large number.

4.3.1 Garbage collection

As shown in section 4.2.2, the storage of objects are based on per-class structures and not a general heap. Many garbage collection algorithms are difficult to implement for this structure, especially since they cannot have direct access to all memory. Garbage collection routines are forced to use one or more of the data sharing mechanisms described in section 4.2.3.

A thorough investigation of all aspects of using garbage collection on the APZ is out of the scope of this study and therefore only a basic scheme for handling garbage collection will be presented. This scheme may have possibilities for execution speed optimizations, but still has the necessary functionality for use in the particular environment and is suitable to real-time applications.

4.3.2 Proposed garbage collection scheme

The garbage collection system proposed for the execution system is based on reference counting. This means that each object instance has an associated value indicating the number of references to it. Every time a reference to the object is created, i.e. when a variable is assigned a reference to the object, the reference count is incremented. Conversely, when a reference to an object is removed (for example by assigning another value to a referring variable) the reference count is decremented. When the reference count is zero any finalizers are run and the object space is inserted into the free list.

The reference counting scheme has advantages:

- Easy to implement. This scheme is relatively easy to implement in the APZ environment.
- It is incremental. The garbage collection work is interleaved in the normal work of the executing program.
- Real-time bounds. The execution of this garbage collection can be made to conform to real-time requirements.

Some disadvantages of reference counting also exist:

- Performance. The total time spent on garbage collection is larger than many other garbage collection schemes.

- Efficiency. Objects which are referring to each other creates cyclic dependencies. These cyclic dependencies cannot be identified by reference counting and therefore two object referring to each other cannot be reclaimed although no other references to either of them exist.

The performance problem can to some extent be relieved by using deferred reference counting. When using deferred reference counting the transient local variables are handled by a separate stage, relieving the need of reference increment/decrement each time a local reference variable is assigned.

The problem of efficiency can be resolved by a using a complimentary garbage collection routine that can be run in the background. This complementary routine traces the references of all objects and resets the reference count of all objects it cannot reach.

A good overview of garbage collection issues and a further explanation of reference counting an tracing is found in Wilson [34]. Discussions of real-time garbage collection can be found in Armstrong, Virding [2] and Wilson, Johnstone [35].

4.4 Activation records

As shown is section 2.3.1, the Java VM uses what is called Java stacks to store activation records for each thread. An activation record contains data relating to a specific invocation of a particular method. The state of a Java stack thus reflects the calling chain that led to the current method.

An activation record typically contains space for the method's local variables, some environment information and an operand stack. The operand stack is used by the stack oriented bytecode instructions. The proposed implementation of the Java execution system translates the bytecodes into native instructions before runtime and can at the same time make a static evaluation of the stack and replace all stack operations with operations on the signal data registers (DR0-DR23) in the APZ register file. As the maximum stack

depth for each method is calculated by the Java compiler and stored in the class file (see section 2.5.4), the required number of registers is known. The need of an operand stack is hence excluded.

The local variables are also stored in the signal data registers. Since both the operand stack and local variables are located in the signal data registers, they are retained intact during a signal sending. This makes it possible for the executive block to store this information when it receives a method invocation request signal. The executive block stores all signal data registers each time it receives a method invocation request. By using this scheme the sum of local variables, stack depth and method parameters cannot be larger than 22, which is the number of signal data registers minus the two registers needed for object reference and global signal number.

The activation records are stored in a file with each thread having it's own linked list. Each activation record includes information on which thread it belongs to, the previous activation record in the thread and the signal register contents. An index to the current method in each thread are stored in a separate variable. The free entries are also organized in a linked list with tail of the free list stored in a variable. All linked lists are singly linked and have insertion and removal done at the tail only, which gives them the logical function of a stack. This method of handling the activation record is efficient and has a time complexity of O(1) which makes it well suited for this implementation.

An example of activation records is given in figure 4.3.

Example of activation record handling:

1. Assume that there are two threads that have been running for some time. The activation record file now has the appearance shown in state 1.

2. Thread 2 is executing and returns from three methods without invocating any new. Thread 1 then returns from one method leaving the activation record file in state 2.

3. Thread 2 now executes and invokes a method which in turn invokes a new method.

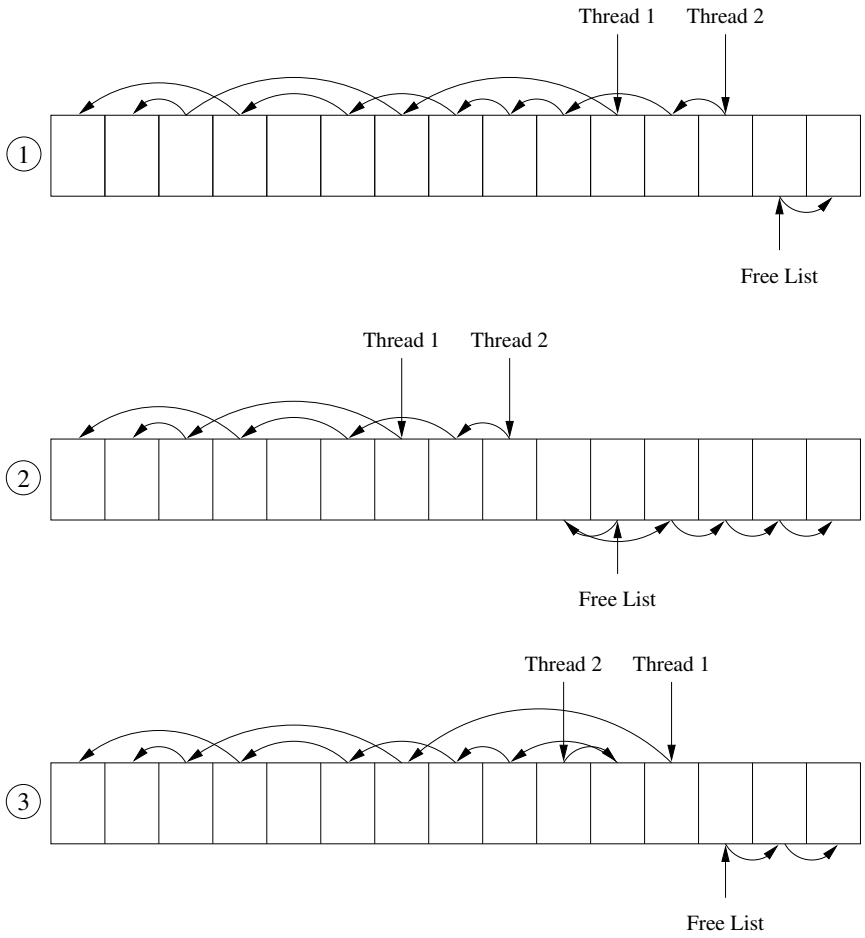

Figure 4.3: Activation records file

Thread 1 then executes and invokes one method leaving the activation record file in state 3.

4.5 Multithreading

Since all method calls are handled via the executive program block, it is also possible for the executive block to perform the switching between the different threads. This implies that change of threads can only be done when a method is called (i.e when a signal is sent) which maps well to the APZ restriction of not interrupting the execution inside a program

block.

The executive block can also yield to other programs outside of the Java execution system by temporarily storing an incoming method invocation request signal and sending a delayed (buffered) signal to itself. Before the buffered signal is processed, other signals in the queue are processed. When the buffered signal is received the executive block continues execution by sending the appropriate signal as specified by the method invocation request signal.

Methods that have long execution times without any method invocations need to have buffered signal calls to themselves inserted. This is a prerequisite of the APZ system since it has no preemptive multitasking. By sending a buffered signal, other tasks in the APZ job queues can be performed.

4.6 Exceptions

Java exceptions are normally handled by the VM by looking up the the program counter value where the exception occurred in the exception table. If a handler for the exception type is found in the exception table, the exception handler begins to execute.

In the APZ environment no access to the program counter is given so some other mechanism for exception handling has to be devised. A solution based on the following is proposed:

- The selection of which handler to use is based on an *exception state.* Every code part which are covered by an exception handler (i.e. are inside a `try` statement) has an exception state.

- When translating the bytecode to native instructions, the exception table is examined and at each exception handling boundary instructions to change the exception state are inserted.

- At the translation of bytecodes, an exception table for each method is set up. The exception table specifies which types of exceptions can be handled at different exception states, and which signals to send when handling an exception(i.e. which exception handling code to execute).

- When an exception occurs, a signal is sent to the executive block which performs the exception table searching and invokes the matching exception handler.

- Since exception handlers can be nested some mechanism to determine the scope of exception handlers must be provided. This is done by representing the exception states with bits in a 32 bit number. The states are thus numbered 2^0,2^1...2^{31} as they are found from the beginning of the method. By scanning for the most significant bit that holds a one, the innermost exception handler is found. If the innermost exception handler does not handle exceptions of the thrown type, the scanning continues for the next handler.

- If no handler for the thrown exception type is found within the exception table of the executing method, the calling chain is traversed backwards to the caller of the current method, and the caller's exception table is examined for handlers. The exception state of the caller has been saved in the activation record so the executive block can search its exception table correctly.

The above proposal has a maximum limit of 32 different exception handlers for each method. Although no such limitation is present in the Java VM specification the need of more than 32 different exception handlers in one method is highly unlikely.

4.7 Number representation

The Java VM specification [17] specifies that the VM uses two's-complement signed integers. The APZ uses unsigned integers. This creates a problem when doing arithmetic and

comparison operations. Three possible solutions are:

1. Disallowing negative numbers.

   This is easy since it forces the APZ integer representation onto the Java execution system. It does however induce an inconsistency in the Java implementation that could lead to program malfunction if code is executed which is dependent on the default behavior for negative numbers.

2. Using a sign bit.

   By using the most significant bit as an indicator of a positive number the sign of the value can be represented. This approach allows the full use of negative numbers with the normal Java semantics even though the internal representation is different from the representation specified in the Java VM specification. By using the most significant bit and letting a one signify a positive number the native APZ comparison operations could be used without any complementing code. Arithmetic operations would however be significantly complicated since they would need to handle multiple sign cases and also overflow conditions.

3. Using two's-complement.

   This is a common method to handle representation of negative numbers. When using this representation the APZ arithmetic instructions used for addition and subtraction can be directly used. Multiplication an division instructions,as well as comparison instructions, need to be extended by sign handling code.

The selection of which number representation to use implies a tradeoff involving speed, reliability and complexity. For the purpose at hand, the most reasonable choice seems to be to use two's-complement.

4.8 Library support

This section describes the API packages that are part of a standard Java distribution and their relevance in the context of application development for the APZ. A good overview of the API packages can be found in Flanagan [10].
The packages are:

- **java.applet**. This package contains a class used when implementing an applet (i.e. an application executable in a browser). This package is irrelevant for APZ applications.
- **java.awt & java.awt.image.** These packages provide GUI and image manipulation functionality. These packages are irrelevant.
- **java.io.** This package provides I/O functionality. Depending on the application this package could be useful. Implementing this package requires mapping of the native methods to the APZ functions for I/O.
- **java.lang.** This package contains many classes that are central to Java usability and most are required in order to make a viable Java execution environment. Some of these classes are:
  - **java.lang.Object** This is the root class in Java. All classes are subclasses of this class.
  - **java.lang.System**. This class provides methods that interface to system functions.
  - Wrapper classes for primitive types. Makes it possible to handle primitive values as objects.
- **java.net**. This package interfaces to TCP/IP network libraries. Unless the APZ is to have any direct connectivity with TCP/IP this package can be ignored.

- **java.util**. This package is a utility package containing several useful classes such as Hashtable, Date and Random. These classes should be implemented as the provide functionality used by other packages.

4.9 Instruction Mapping

This section discusses how the instruction set of the Java VM can be mapped to the instruction set of the APZ. The categorization used here is the same as the one used in section 2.4 where the JVM instruction set is presented. The instruction set of the APZ is presented in section 3.5.2.

The instructions groups are presented with an explanation on what their function are in the JVM and how this functionality is achieved with APZ instructions.

1. Push constant

   JVM: Instructions that push constant values onto the operand stack.

   APZ: Load the value into the corresponding register.

2. Load/Store from/to local variable

   JVM: Instructions that move values to/from local variables to/from local variables.

   APZ: Move to/from register representing local variable from/to register representing the stack.

3. Stack management

   JVM: Instructions that manipulate the stack, copying, moving or removing values on the stack.

   APZ: Manipulation of registers representing the stack.

4. Arithmetic

   JVM: Instructions that perform the common arithmetic functions (+, -, *, /, mod, negate) on `int`, `long`, `float` and `double` respectively.

APZ: Support for arithmetic operations on `int` only. Negative number representation differs, see 4.7.

5. Logical

   JVM: Instructions that perform shifts, logical and, or, xor functions on `int` and `long`.

   APZ: Corresponding functions exist for `int`.

6. Type conversion

   JVM: Instructions that convert between `int`, `long`, `float` and `double`.

   APZ: No conversion needed, only `int` type available.

7. Control transfer and compare

   JVM: Instructions that perform unconditional or conditional jumps.

   APZ: Required compares can easily be constructed.

8. Table jumping

   JVM: Instructions *lookupswitch* and *tableswitch* provide for a comparison of a key value against a set of match values and jump if a match is found.

   APZ: The instruction *lookupswitch* can be transformed to a native table jump instruction, *tableswitch* has to be constructed by performing multiple comparisons.

9. Array management

   JVM: Instructions for allocating, storing and retrieval of arrays. Array instructions support the `byte`, `short`, `char`, `int`, `long`, `float`, `double` and `objectref` types.

   APZ: Array management are provided by sending signals to the executive block.

10. Object related

    JVM: These instructions handle different aspects of object operations.

    APZ: All these instructions are transformed to suit the object handling model of the APZ. See section 4.2.

11. Monitors and Exception handling

    JVM: Instructions *monitorenter* and *monitorexit* provide locking for synchronization. Instruction *athrow* throws an exception.

    APZ: Synchronization is provided by the executive block which handles all method invocations. An exception is dealt with according to section 4.6.

4.10 Summary

This chapter has proposed Way ahead of time compilation as being the most appropriate execution model. Object handling is proposed to be performed by a runtime framework whose principal component is an executive program block that performs functions such as object creation, dynamic method selection, exception handling and garbage collection. A combination of statically and dynamically allocated memory are suggested for object storage, supervised by the executive program block. Activation records are stored in single linked lists in a file. The task switching needed to support multithreading can be obtained by sending buffered signals since the hardware lacks any pre-emption. The problem posed by the difference in number representation between the JVM and the APZ was discussed. An instruction set mapping between the JVM and the APZ assembler instructions were provided, and no overwhelming difficulties in performing the mapping was found.

Chapter 5

Conclusions

5.1 What has been done?

This work examined the feasibility of implementing a Java execution system on a non-generic platform, in this case the APZ 212. The Java Virtual Machine was presented in chapter 2 and the APZ 212 in chapter 3. The information in these chapters was synthezied into chapter 4 which presented several possible alternatives for method inheritance, number representation and execution model. Chapter 4 also presented the mechanisms developed for handling object representation, dynamic method selection, memory handling, activation records, multithreading and exceptions for a Java execution system implementation on the APZ. An overview of the library support and instruction mapping was also given. A large fraction of the work needed to complete this work was collecting, reading and re-reading the source material, which was typically lacking the structure needed to penetrate this somewhat complex subject.

5.2 Results

This section presents answers to the research questions as summarized below and gives an overview of the details of the implementation proposed in chapter 4.

5.2.1 Research questions

Question 1

Is it possible to implement a practically useful Java environment for a non-generic hardware platform which has a hardware architecture that is markedly different from the hardware architecture the Java VM was targeted for?

Answer 1

This work shows that no insurmountable difficulties exist for implementing a practically useful Java execution environment for the APZ 212 platform, but there are difficulties.

Question 2

If so, what are the problems/possibilities inherent in the mapping of the execution model of Java to the specifics of the hardware platform, and what implementation strategy is best suited to make maximum use of the hardware.

Answer 2

1. Problems/Possibilities

   (a) Problems.

   Problems arise in the areas where the architectural discrepancies are the greatest between the platform upon which the Java VM was designed to run and the APZ 212. These are:

i. Memory handling.

The memory handling of the APZ is markedly different from the generic computer platform upon which the JVM was designed to be run. The APZ physically separates code and data, and by default disallows any sharing of data between different program blocks.

ii. Negative number representation.

The JVM is specified to use signed integers, and the APZ is based on unsigned integers. This problem and possible solutions are discussed in section 4.7.

(b) Possibilities.

The APZ architecture also has some features that can be used to effectively implement some functionality needed for the JVM. These are:

i. Signal sending

Signal sending is a fast hardware assisted mechanism which allows for example, non-complex method invocations to be performed quickly.

ii. Powerful addressing modes.

The composite data structures used for storing data are directly accessible with APZ instructions.

2. Implementation strategy

The implementation strategy found to be best suited for the implementation of a Java execution system on the APZ 212 is Way Ahead Compiling (WAT) of the bytecodes, i.e. the translation of bytecodes to native instructions is performed before runtime and outside of the APZ system. The proposed implementation also entails the use of an executive program block to perform the services that requires special treatment due to the architecture specific constraints of the APZ. The executive program block thus creates a degree of abstraction used to lessen the architectural differences.

Question 3

How is the execution performance of Java programs affected by the above factors and what is the level of performance to be expected in comparison with similar implementations using other hardware architectures.

Answer 3

Since no actual implementation of a Java execution system has been performed on the APZ, no empirical results exist. However, some assumptions can be made about the performance. The factors that affect performance are listed below and their influence relative to an interpreted implementation on a generic platform is discussed. Such a generic implementation can for example be Sun's JDK on a PC.

Positive:

- The bytecodes are compiled, not interpreted. This yields a significant speed advantage.
- Some functions such as method invocations should be faster due to the effective implementations of signal sending in the APZ.

Negative:

- Some variable accesses may require signal sending due to the memory protection of the APZ.
- The proposed garbage collection algorithm is slower than some of the algorithms available on machines with generic memory systems.

How these factors distribute is varied depending on the characteristics of the application being run. It is impossible to make a precise estimate of how the execution speed is affected by the above factors. However it is likely that it will be within an order of magnitude relative to a generic computer platform with a similar (WAT) execution system and with similar hardware performance. The implications of the architectural differences are hard to ascertain, but probably the restrictions set up by the memory handling of the APZ has enough negative side-effects as to result in an overall slower system.

5.2.2 Proposed implementation

This section presents an overview of the proposed implementation of a Java execution system. A more detailed description of the different issues are found in chapter 4.

The proposed implementation is as follows:

- The execution model is based on Way Ahead Compiling (WAT) of the bytecodes, i.e. the translation of bytecodes to native instructions is performed before runtime and outside of the APZ system. The main advantage of this model besides speed is that it is suited to the architecture of the APZ in that the translation and optimization of bytecodes are done outside of the APZ, in an environment better suited to perform such work than the APZ.
- An executive program block performs the runtime functions of the execution systems. This includes:
  - Memory management task such as object allocation and garbage collection.
  - Method invocation. Both dynamic and static method invocation and the associated processing of activation records.
  - Exception handling. Finding the correct handler for an exception is done by searching the exceptions tables and, if necessary, unwinding the activation records on the call stack of the affected thread.

  - Thread switching. This is done by inserting buffered signals between method invocations or inside lengthy methods.

- Classes are represented as program blocks with object instances stored in a variable file associated to the program block. The variable file is statically sized and a dynamic buffer in the storage area is used in case the number of objects exceeds the size of the static allocation.

- The implementation is dependent on some type of data sharing mechanism to be present. Several mechanisms are presented in section 4.2.3, and the selection of which to use is dependent on decisions outside of the scope of this study, primarily to which degree the APZ are allowed to change in order to accommodate the required data sharing. Mechanisms are proposed that require hardware changes, software changes and no changes to the APZ.

- The Java virtual machine is specified to operate on signed integers whereas the APZ is based on unsigned integers only. Three possible solutions are discussed in section 4.7. The proposed solution is to use two's complement to represent negative numbers. This requires multiplication, division and comparison instructions of the APZ to be embedded with additional sign handling code.

5.3 Conclusions

It is possible to implement a Java execution system on a non-generic architecture such as the APZ 212 platform. The main problems are related to the memory architecture and the use of unsigned numbers in the APZ. The problems related to memory architecture has several possible solutions, differentiating in the degree of change induced to the APZ 212. The problem of unsigned numbers also have several possible solutions.

5.4 Related work

Many implementations of Java execution systems exist, but no known implementation for a machine similar to the APZ is known to exist. There are however some WAT based execution systems in development for generic platforms. These are Harissa [20], developed at University of Rennes, and Toba [22], developed at University of Arizona. Harissa includes a bytecode interpreter in the runtime library, which enables it to dynamically load classes. Toba has no such ability, but are able to handle threads, which Harissa can not. They both have techniques that are applicable to a Java execution system implementation on the APZ. Examples of useful techniques are static stack evaluation and class hierarchy analysis.

5.5 Further research

This study proposes an overall framework for the implementation of a Java execution environment for the APZ. Several areas needs to be investigated in further detail in order to achieve a optimum implementation. These areas are:

- Garbage collection (section 4.3.1). The implementation details of the reference counting mechanism needs to be further examined.

- Object dynamics (section 4.2.2). The possibility of determining the maximum number of concurrent objects for all classes needs to be investigated. Algorithms for examining the bytecode needs to be developed and possibly a meta-notation for specification by the programmer needs to be created.

- Task switching (section 4.5). The mechanisms of inserting buffered signals into lengthy methods with few method invocations need to be further examined.

5.6 Future work

Future work to be made on this subject would, besides the above mentioned, include:

1. Construction of a class file loader and analyzer.
2. Designing and writing the executive program block, probably in ASA assembler code.
3. Adding a bytecode compiler to the class file loader & analyzer which is capable of interfacing to the constructed executive program block.

References

[1] Anuff, E.; *Java Sourcebook*; John Wiley & Sons; New York; 1996.

[2] Armstrong, J., Virding., R.; One Pass Real-Time Generational Mark-Sweep Garbage Collection, Ellemtel Telecommunications System Laboratories

[3] ASA210C, Assembler Instruction Summary, Internal Document, Ericsson, 11/1551-ANZ 211 51 Uen. 1993.

[4] Bank, D.; The Java Saga, *Hotwired,* Issue 3.12, December 1995. http://www.hotwired.com/wired/3.12/features/java.saga.html

[5] Case, B.; Implementing the Java Virtual Machine, *Microprocessor Repor*t, Nr 4 1996.

[6] Case, B.; Java Virtual Machine Should Stay Virtual, *Microprocessor Report*, Nr 5 1996.

[7] Christenson, B., Mitchell, J. D.; That First Gulp of Java, *Linux Journal*, October 1996, pg 17-19

[8] CPS Priciples, Internal Document, Ericsson, 2/1551-ANZ 211 60 Uen. 1995.

[9] Dean, J., Grove, D., Chambers, C.; Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag. http://www.cs.washington.edu/research/projects/cecil/www/Papers /hierarchy.html

[10] Flanagan, D.; *Java in a Nutshell*; O'Reilly & Associates, Sebastopol, Calif. 1996.

[11] Gosling, J., Joy, B., Steele, G.; *The Java Language Specification*; Addison-Wesley, Reading; Mass. 1996.

[12] Gosling, J., McGilton, H., The Java Language Environment - A White Paper, Sun Microsystems Computer Company, Mountain View CA, 1996. http://java.sun.com/doc/white_papers.html

[13] Inside the Java Virtual Machine, *Unix Review*, January 1997.

[14] Kleinöder, J., Golm, M.; MetaJava: An efficient Run-Time Meta Architecture for Java. *Proceedings of the International Workshop on Object Orientation in Operatin Systems - IWOOOS '96,* Seattle, IEEE, 1996

[15] Kramer, D.; The Java Platform - A White Paper, JavaSoft, Mountain View CA, 1996. http: //java.sun.com/doc/white_papers.html

[16] Lentczner, M.; Java's Virtual world, *Microprocessor Report*, Nr 4 1996.

[17] Lindholm, T., Yellin, F.; *The Java Virtual Machine*; Addison-Wesley, Reading; Mass, 1996.

[18] McManis, C.; The basics of Java class loaders, *Java World Magazine,* October 1996. http://www.javaworld.com/javaworld/jw-10-1996/jw-10-indepth.html

[19] McManis, C.; Not using garbage collection; Minimize heap thrasing in your Java programs, *Java World Magazine,* September 1996. http://www.javaworld.com/javaworld/jw-09-1996/jw-09-indepth.html

[20] Muller, G., Moura B., Bellard F., Consel C.; Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code, Technical Report, IRISA / INRIA - University of Rennes, http://www.irisa.fr/compose/harissa/harissa.html

[21] Niemeyer, P., Peck, J.; *Exploring Java*; O'Reilly & Associates, Sebastopol, Calif. 1996.

[22] Proebsting T. A., Townsend, G., Bridges, P., Hartman, J. H., Newsham, N., Watterson, S. A.; Toba: Java For Applications - A Way Ahead of Time (WAT) Compiler. Technical Report TR97-01, University of Arizona, 1997, http://www.cs.arizona.edu/sumatra/toba

[23] Venners, B.; Under The Hood: The lean, mean virtual machine, *JavaWorld Magazine,* June 1996. http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.html

[24] Venners, B.; Under The Hood: The Java class file lifestyle, *JavaWorld Magazine,* July 1996. http://www.javaworld.com/javaworld/jw-07-1996/jw-07-classfile.html

[25] Venners, B.; Under The Hood: Java's garbage collected heap, *JavaWorld Magazine,* August 1996. http://www.javaworld.com/ javaworld/jw-08-1996/jw-08-gc.html

[26] Venners, B.; Under The Hood: Bytecode basics, *JavaWorld Magazine,* September 1996. http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html

[27] Venners, B.; Under The Hood: Floating-point arithmetic, *JavaWorld Magazine,* October 1996. http://www.javaworld.com/javaworld/jw-10-1996/jw-10-hood.html

[28] Venners, B.; Under The Hood: Logic an integer arithmetic, *JavaWorld Magazine,* November 1996. http://www.javaworld.com/javaworld/jw-11-1996/jw-11-hood.html

[29] Venners, B.; Under The Hood: Objects and arrays, *JavaWorld Magazine,* December 1996. http://www.javaworld.com/javaworld/jw-12-1996/jw-12-hood.html

[30] Venners, B.; Under The Hood: How the virtual machine handles exceptions, *JavaWorld Magazine,* January 1997. http://www.javaworld.com/ javaworld/jw-01-1997/jw-01-hood.html

[31] VENNERS, B.; Under The Hood: Try-finally clauses defined and demonstrated, *JavaWorld Magazin*e, September 1996. http:// www.javaworld.com/javaworld/jw-02-1996/jw-02-hood.html

[32] VENNERS, B.; Under The Hood: Control flow, *JavaWorld Magazine,* March 1997. http://www.javaworld.com/javaworld/jw-03-1997/jw-03-hood.html

[33] WAYNER, P.; Sun Gambles on Java Chips, *BYTE*, November 1996

[34] WILSON, P. R.; Uniprocessor garbage collection Techniques, In Beckers, Y., Cohen, J., editors, *International Workshop on Memory Mangement*, number 637 in Lecture Notes in Computer Science, pages 1-42, St. Malo, France, September 1992, Springer-Verlag.

[35] WILSON, P. R., JOHNSTONE, M. S.; Real-Time Non-Copying Garbage Collection. Position paper ACM OOPSLA Workshop on Memory Management and Garbage Collection, 1993.

[36] YELLIN, F.; The Java Native Code API, JavaSoft, Mountain View CA, 1996. http://java.sun.com/doc/jit_interface.html

[37] YELLIN, F., LINDHOLM T.; Java Runtime Internals, Slides from a lecture at JavaOne Developers Conference, 1996, http://www.javasoft.com/javaone/javaone96/pres/Runtime.pdf

Appendix A

Background on Java

A.1 Introduction

This appendix presents an overview of the Java language. In order to understand the rationale of the language design, this appendix begins with a brief historical background of the development that was to lead to the Java language. The key concepts of the language is then discussed in order to give a feel for the language characteristics. More information on the history of Java is available in Bank [4].

A.2 History of Java

The history of Java began in 1991 with a programming language called Oak. This was a programming language originally designed to offer control interfaces for consumer electronics. The control interfaces were to be constructed using a device with a touch sensitive LCD matrix display showing a virtual video player for example. This was part of a project at Sun which was looking for new applications and to diversify Sun's business opportunities. In August 1992 a control device programmed using Oak was demonstrated and Oak was pushed as the programming language for creating user interfaces in devices such

as cellular phones, televisions, home and industrial automation systems. After the initial enthusiasm had died down due to the relatively high cost for the supporting hardware (chip & display), the project was redefined in 1993 to target on interactive TV. The goal was to supply set-top boxes, programmed in Oak, which were able to handle the vast flow of images, data and money transactions predicted to be circulating on the information super-highway. However, Sun lost the tender for supplying set-top boxes for the large Time Warner interactive TV trials in Florida that year. This caused the project,which had up till now supported Oak development, to collapse early 1994. At this time the web started to explode in use and a Sun co-founder, Bill Joy, saw the potential for Oak on the Internet and funding continued. Late 1994 an early version of the revised language was released and in January 1995 it was renamed to Java. After a while Java received a lot of publicity. Netscape licensed the technology in order to include Java into their browsers and the Java technology began to catch on...
From the above the design goals that sets Java apart from most other languages can be derived:

- Programs used to control consumer appliances must be stable. An average consumer will not accept not being able to use his TV because of a program glitch in the control device.
- Since Java originally was intended for many different devices Java had to be platform independent.
- Using Java to control set-top boxes on the information super-highway called for good networkability and security.
- In order to effectively use the network and use as little bandwidth as possible, Java should be object oriented to facilitate the distribution of classes to the user.
- In order to make the transition for current C++ programmers to Java as easy as

possible the syntax and semantics of base language constructs is much the same as C++.

In essence Java is an object-oriented, distributed, network-aware, portable language intended to simplify both programming for the Internet specifically as well as programming in general.

A.3 Key concepts in Java

Java bears a resemblance to C++ which is intentional as Java was designed to be easy to use for programmers who are fluent in C++. Java is, however, not just "a dialect" of C++ but a totally new language environment and a new concept for distributing applications over a network. The main points of Java will be discussed in the following chapters on portability, object-orientation, distribution, performance and safety. Further information can be obtained from multiple sources such as [1, 7, 10, 12, 15, 21].

A.3.1 Portability

Current languages

In languages such as C, C++ and Pascal[1] the sourcecode is passed through a compiler which produces machine code runnable on a specific processor running a specific operating system. This of course creates a problem when used in an environment where a heterogenous collection of platforms is used and separate versions of the source code/executable code must be maintained for every platform on which a program should be able to execute. The largest collection of interconnected computers is the Internet, and the Internet is a very heterogenous collection of computers which are unable to share code written for another platform without conversion.

[1] Although Pascal implementations based on P-code actually used the concept of a virtual machines long before Java.

Java compilation

Java differs from most current languages in that the execution of a program written in Java source code encompasses two steps (compilation and interpretation) instead of one (compilation or interpretation). This is the key to the solution of the above problem, to produce programs that do not have to be modified in order to run on different platforms. This is achieved by using a virtual machine as target for the source code compilation instead of any particular processor/operating system combination as shown in figure A.1. Illustrated by the figure is also the fact that the statement that execution of a Java source code program encompasses two steps, compilation and interpretation, is not entirely true. Java Virtual Machine implementations that use a compiling scheme for converting Java bytecodes into native machine instructions promises a considerable gain in performance as described in section A.3.5.

The result from a Java source code compilation is called a Java .class file and it contains program information and Java byte code, which in effect are machine code instructions for the Java virtual machine.

Java bytecode interpretation

The Java virtual machine, being a part of the Java execution system, is implemented on each platform that wishes to support Java. The .class file is executed by a platform's Java execution system. In this manner all the platform specific knowledge is located deep inside the Java execution system. The programmer writing in Java does not require to know anything about the processor, operating system or file system layout on the system where his program is to be run. This means that the compiled Java programs (.class) can be distributed and run on every platform that has a Java execution system implemented. The Java execution system loads the bytecodes and transforms them to machine code instructions for the host processor. This transformation is made either by interpreting the

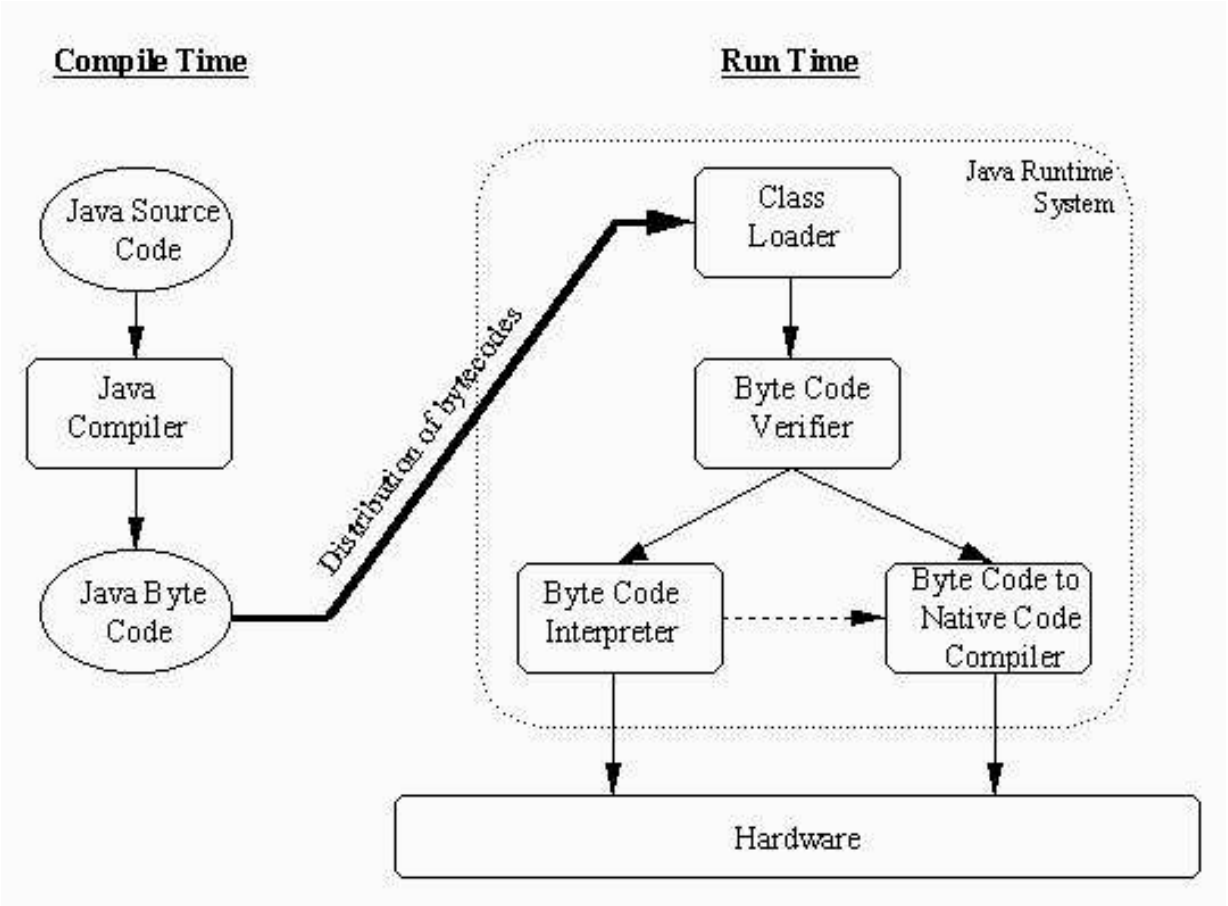

Figure A.1: Flow of Java Source

bytecodes or by compiling them as detailed in section A.3.5.

A.3.2 Object-orientation

Java is intrinsically designed to be an object oriented language. Java supports the key concepts of the object-oriented paradigm:

- Abstraction - Everything in Java (except primitive types) are seen as objects.
- Encapsulation - Hiding of implementation is done inside of classes.
- Inheritance - New classes can be defined as extensions of existing ones in order to

obtain code re-use and organization into class hierarchies.

- Polymorphism - The same message sent to different classes results in behavior that is dependent on the receiver's class.

- Dynamic (late) binding - Dynamic binding in Java provides the ability to send messages to an object without knowing its specific type at coding time. This provides for the use of non-local objects.

One point to make that distinguishes Java from C++ is that Java addresses the fragile superclass problem (a.k.a the constant recompilations problem). This problems arises as a consequence of the way C++ is usually implemented with static references to attributes (variables) and methods inside classes. If an attribute is added to a class it then displaces the relative location of the following attributes and methods. In order to provide for this change all classes that reference the class appended to have to be recompiled. This has proven to be a fault prone system even if a make utility is used to help manage the dependencies between classes.

Java instead uses symbolic references and resolves the reference once when the classes are loaded into the Java runtime system. The storage layout of classes in memory is not decided by the compiler but by the runtime system. This means that classes can grow incrementally and that adding new attributes and methods to existing classes does not make them unusable to previous programs. The cost for this is a small time penalty of a name lookup the first time a name is encountered, but it contributes to make Java a reliable programming environment.

Another difference is that Java supports only single inheritance between classes in contrast to C++ which also allows multiple inheritance. Multiple inheritance is however a mechanism that is quite complex and can give complex class hierarchies that are difficult to understand. Java instead uses the interface construct to provide multiple inheritance where needed, albeit in a simpler but still powerful way.

A.3.3 Distribution

Java was originally conceived as a language to be used in a networked environment and therefore has built in support for networking. Java can even dynamically load supporting classes to an application over a network at run time. The client/server model is further enhanced by Java which allows a server to not only supply the data to display, but also supply a program for displaying the data in the most meaningful way, helping the user to navigate through the data. The only requirement on the client side is to have a network connection and a Java execution system. Since large and/or demanding Java applets and applications can be cached and compiled to achieve a performance thought to be similar to an ordinary program, the corporate PC networks may in the future change into networks of cheap, JavaChip(see [6, 33])- based workstations. This could lessen the support costs and give additional corporate benefits.

A.3.4 Security

Programmer security

Programmer security is defined as the possibility for the programmer to make logical programming errors, i.e. to produce syntactically correct code that passes through the compiler without generating any error or warning messages, but still causes the program to perform in an unwanted way under certain conditions. The number of logical errors that is possible to produce are partly due to the design of the language. It is very hard to guarantee the absence of logical errors in a piece of code. Java, being based on C++, has however made some attempts to minimize some of its predecessors sources of logical errors:

- Pointers have been abolished, this means that it is no longer possible to have segmentation faults produced by incorrect pointers.

- A garbage collection system is used, relieving the programmer from caring about when to free a particular piece of memory. This means that it is no longer possible to by mistake free memory that still contains relevant data that is used somewhere later in the program.

- All references to arrays include bounds checking.

- The source code is made more context-free, i.e. a programmer trying to understand another programmer's work does not have to read a collection of header-files, #defines and typedef declarations before he can begin to analyze the actual code. The preprocessor and header files are removed from Java and the required functionality is provided by more appropriate means or is present inherently in the language.

- No possibility to mix different programming paradigms. Java forces everything to be an object, it is not possible to write a standalone function. This is possible in C++ and is a source for much bewilderment since it allows programs that are a cocktail of different programming paradigms: the object-oriented, the imperative, and the functional. In the best case this leads to programs which take the best of each paradigm, in most cases it probably just leads to confusion.

- Being a strongly typed language, Java compilers have the possibility to perform extensive compile-time checking.

User security

Java is also designed with user security in mind. Since Java is designed to run programs that can be downloaded from anywhere on the Internet, the user must be protected against malicious programs trying to do damage. This is achieved by several means, some of the most significant being implemented in the bytecode verifier, the class loader and the networking package.

Bytecode verifier

The bytecode verifier checks every piece of code that is to be run against illicit behavior such as:

- Forging references.
- Violation of access restrictions.
- Accessing objects in a way that is non-compliant with their type.

The bytecode verifier also performs checks to ensure program consistency. These checks include crosschecking type state information and bytecode operator analysis. When the checks are finished, the program is known to have no operand stack over- or under-flow, have correct types for all bytecode parameters and to only have legal object field accesses. Having performed these checks at load time relieves the bytecode interpreter/compiler from doing any such checks at runtime which will speed up its code execution.

Class loader

An executing Java program can dynamically load a required class, either from the local class storage or over a network. The class loader partitions the classes into different namespaces that are given different privileges. The class loader must guarantee that a class loaded externally cannot pass as being one of the locally stored trusted classes.

Networking package

The networking package handles all Java accesses using network protocols such as FTP and

HTTP. The networking package is configurable in order to comply with the users safety requirements. It can be set to disallow all network traffic, allow network access only to the host that the imported code came from, allow access only outside of a firewall or to allow all network access.

Another property of Java that renders intentional harmful programming more difficult is the fact that Java increases the level of abstraction for the programmer. The execution system hides all low-level detail and the programmer cannot make use of security deficiencies in an operating system or file system implementation because he has no access to them (and they may not be present on the users platform). Since Java is designed to be platform independent, Java programs have no way of using platform-specific security loop-holes. This said, it can be questioned if present implementations are totally secure. However, as Java implementations mature they have a increased potential for using the built-in provisions for security to provide the safest possible applications.

A.3.5 Performance

A Java program is first compiled by the Java sourcecode to bytecode compiler. This is normally done by the programmer. The bytecode is then distributed by a network or by other means to the user. The user then runs the bytecodes on his computer using an implementation of the Java execution system.

The execution system can be implemented either in a browser for running applets in a browser window, or as a standalone program for running standalone Java applications. In both cases the runtime system may use either an interpreter or a compiler to transform the bytecodes to machine code executable by the host processor.

Compiler variants

There are different approaches to the constructions of native machine code generators for Java. The below categorization is partly based on Yellin [36].

- Ahead-of-time compilers: A compiler that converts the Java source code into a "fat" class file, which contains both the Java byte codes and one or more native machine-code definitions for some of the methods. This compilation has to be done by the programmer because it requires access to the Java source code.

- Ahead-of-time recompilers: A compiler that converts the .class file containing the Java bytecodes to a "fat" class file as above.

- Just-in-time code generators: A JIT code generator generates native machine code for methods as they are running on the virtual machine. After the first call to a method processed in this manner, it will be executed by native machine code instead of Java byte code being interpreted.

- Flash (downloading) compilers: compilers that converts all bytecodes to native machine code instructions when they are loaded into the runtime system (either from local storage or across a network). The main difference between the JIT and a downloading compiler is that in a JIT system the virtual machine is still based on an interpreter, which calls a JIT compiler for the appropriate classes/methods. This leads to some overhead in the interpreted/ compiled code transitions as well as preventing global code optimization.

- Way ahead of time (WAT) compilers: This term first appears in Proebsting, et al. [20] and signifies a compiler that processes java class files and produces output in some intermediate language. Most common are to use C as an intermediate language as this allows for a high degree of optimization to be done by optimizing C-compilers.

To be noted is that the first and to some extent the second and last approach are contradictory to the Java vision: To create small, portable applications. Depending on the

demand of a particular application the most appropriate method may be chosen, but it is pointed out in the Java documentation that many applets which serve to enhance the interactivity are not speed-critical and therefore do not need any kind of compiling scheme. The question of which technique offers the best balance between speed of execution, time for start-up and download time is an aspect that differs for different applications/uses of Java.