Architecture for Object-Oriented Programming Languages

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

ABSTRACT

In this paper we investigate the overheads of object-oriented operations, such as virtual method dispatch and field access, in the context of an embedded processor for real-time systems. As an example we use a Java processor that implements those operations in microcode similar to the way those operations are compiled to a RISC processor. As this processor is a soft-core, implemented in an FPGA, an optimization of those operations is a valuable option. Significant application speedup is possible by providing an architecture for object-oriented programming languages. We also evaluate the hardware cost of this optimization with respect to the application speedup.

1. INTRODUCTION

Object oriented (OO) languages, such as Java and C#, are the dominant languages for desktop and server programming. However, in embedded systems C is still the common choice. This conservatism in the embedded systems domain is not just the availability of a large code base in C. The main reason is the pressure for efficiency – with respect to memory consumption and processor resources. Java, as a popular example of an OO language, uses just-in-time compilation on the target to achieve an acceptable performance and still provides the platform independent class files. In an embedded system a compiler on the target is usually not an option due to the large memory usage. Therefore, the Java virtual machine (JVM) in an embedded system is still implemented as an interpreter.

One solution for a high-performance JVM for embedded systems is a Java processor. A Java processor implements the bytecodes, the instruction set of the JVM, in hardware. In [10, 11] JOP, the Java optimized processor, is presented. JOP is intended to be a time-predictable Java processor for embedded hard real-time systems. Simple bytecodes are implemented in hardware, more complex, such as OO oriented bytecodes, by microcode sequences. Compared to other Java processors and solutions in the embedded domain JOP is a very small and high performance solution [9]. In this paper we evaluate the benefits from implementing OO related instructions on JOP[1].

Other Java processors such as aJile's JEMCore [1, 3], Sun's picoJava [8], and Komodo [6] are quite similar to JOP. Simple bytecodes are supported by the processor pipeline, more complex are implemented by the execution of microcode or with a software trap. The Cjip processor [2, 5] takes this approach to the extreme: All bytecode instructions are implemented by microcode to support multiple instruction sets for Java, C, C++ and assembler.

In [15] support for object access in a Java processor is considered. The proposal deals to a great extent with a new cache architecture for objects. However, no implementation or estimation of the implementation complexity is given. The jHISC project [14] proposes a high-level instruction set architecture for Java. This project is closely related to the proposed approach. However, the resulting design is probably not very well balanced. The processor consumes 15500 LCs compared to about 3000 LCs for JOP. The maximum frequency in a Xilinx Virtex FPGA is 30 MHz compared to 100 MHz for JOP. According to [14] the prototype can only run simple programs and the performance is estimated with a simulation.

The rest of the paper is organized as follows: Section 2 gives an overview of OO instructions in Java as defined by the JVM specification [7]. In Section 3 we investigate the hardware implementation of OO instructions in a quantitative approach and evaluate the results in Section 4 by implementing array instructions in hardware on JOP. Section 5 concludes the paper and gives directions for future development.

2. OO INSTRUCTIONS

The JVM specification [7] defines bytecodes for OO instructions. Those instructions fall into four categories:

- Object and array creation
- Method invocation
- Field access
- Array access

All those instructions are related to either an object[2] or a class. That means that the instructions operate on a reference to an object or class. Therefore, the object and class structure layout of the runtime system influences the complexity of the instruction. When e.g., the JVM uses a compacting garbage collector (GC) the object references are usually implemented by an indirection through

[1]JOP is open-source and all sources, including the changes proposed in this paper, are available at `http://www.jopdesign.com/`

[2]Arrays are considered objects in Java

JTRES '07 September 26-28, 2007 Vienna, Austria

a handle. In that case the movement of objects by the GC is simplified, but the object access involves an additional memory load.

Java is a safe language with runtime checks to avoid hard to find pointer errors such as in C. Each reference to an object is symbolic. That means that no *addresses* to data structures are available at the JVM level. No *efficient* pointer arithmetic is possible. Furthermore, each usage of a reference is checked at runtime to be not null. A null reference has to raise an exception.

2.1 Object and Array Creation

Class instances and arrays are created with single bytecodes such as new or newarray. The objects are allocated on a heap and are *deleted* by a garbage collector (GC) when not referenced anymore. The creation bytecodes interact closely with the GC and are therefore usually implemented by software on a Java processor.

2.2 Method Invocation

Methods in Java come in two flavors: static methods that belong to the class and virtual methods that belong to an object. A virtual method is the main vehicle to implement polymorphism. The receiver method depends on the object type. Therefore, the target method is selected at runtime. To avoid searching the receiver method in the class hierarchy the common implementation of method lookup is through a virtual method dispatch table. The table for a class (object type) contains references to all inherited methods and overridden or additional methods. Additional to the method address some auxiliary information, such as number of parameters and number of local variables, is necessary for an efficient method invoke. The JVM defines four bytecodes for method invocation:

invokestatic: A class method (declared static) is invoked. As the target does not depend on an object, the method reference can be resolved at load/link time.

invokevirtual: An object reference is resolved and the corresponding method is invoked. The resolution is usually done with a dispatch table per class containing all implemented and inherited methods. With this dispatch table, the resolution can be performed in constant time.

invokeinterface: An interface allows Java to emulate multiple inheritance. A class can implement several interfaces, and different classes (that have no inheritance relation) can implement the same interface. This flexibility results in a more complex resolution process. One method of resolution is a search through the class hierarchy that results in a variable, and possibly lengthy, execution time. A constant time resolution is possible by assigning every interface method a unique number. Each class that implements an interface needs its own table with unique positions for each interface method of the *whole* application.

invokespecial: Invokes an instance method with special handling for superclass, private, and instance initialization. This bytecode catches many different cases. This results in expensive checks for common private instance methods.

2.3 Field Access

Fields, similar to methods, come in two flavors: static or class fields that belong to a class and object (or class instance) fields. Static fields are accessed by the bytecode getstatic and putstatic. As the addresses for the static fields are defined during class load and link time the access is usually faster than for object fields.

```
public int test(int cnt) {

    int a = 0;
    int i;

    for (i=0; i<cnt; ++i) {
        a += arr[i&0x3ff];
    }
    return a;
}

public int overhead(int cnt) {

    int a = 0;
    int i;

    for (i=0; i<cnt; ++i) {
        a += abc&0x3ff;
    }
    return a;
}
```

Figure 1: Micro benchmark to measure iaload

Object fields are inherited in the same way as methods. A similar *trick* such as the method dispatch table can be applied to object fields: objects contain the inherited fields at the same position as in the super class. Additional fields are added at higher positions. This position can be determined at class load/link time and no runtime resolution is necessary.

2.4 Array Access

Arrays are first class objects in Java. An array is allocated on the heap and can only be deleted by the GC. As with objects a reference indirection through a handle is usually needed for a compacting GC. Furthermore, besides the usual null pointer check, a JVM has to perform array bounds checks at runtime. All those operations result in a way slower array access as compared to a C style array.

2.5 Execution Time

We have measured the execution time of some simple bytecodes and OO bytecodes on two Java processors. As we cannot directly measure the execution time of a single bytecode[3] we use an indirect approach. We measure the execution time of two different bytecode sequences in a loop. One sequence contains the bytecode to measure, the other sequence not. The difference between the two execution times (divided by the loop count) is the execution time of the bytecode under test. However, as almost all bytecodes manipulate the stack we have to include at least a second bytecode to compensate this change.

Figure 1 shows an example to measure the execution time of iaload. The execution time of overhead() is subtracted from the execution time of test(). Figure 2 lists the bytecodes of the code within the loop for method test() and method overhead(). The difference of the two methods are the bytecodes iload_3 and iaload. That means we can only measure the execution time for both bytecodes.

Furthermore, the benchmark framework is adaptive to provide meaningful results for different platforms. The loop bound (cnt) is

[3]It would be possible by instrumenting the bytecode and when a clock cycle counter is available that can be accessed at the bytecode level. This is possible in JOP, but not in the general case on a JVM.

```
test loop:
   9:   iload_2
   10:  getstatic   #2; //Field arr:[I
   13:  iload_3
   14:  sipush  1023
   17:  iand
   18:  iaload
   19:  iadd
   20:  istore_2

overhead loop:
   9:   iload_2
   10:  getstatic   #3; //Field abc:I
   13:  sipush  1023
   16:  iand
   17:  iadd
   18:  istore_2
```

Figure 2: Bytecode listing of the iaload benchmark

increased (exponentially) until the test method runs for more than 1 second. With this relative long measurement we can use System.currentTimeMillis() to measure execution times of short bytecode sequences.

Table 1 shows execution time measurements of simple bytecodes and OO-related bytecodes on JOP and on the aJile aJ100 [1]. As explained above, those measurements include additional bytecodes. The first row shows the execution time of an iload_3 and an iadd. For the JOP case both instructions execute in a single cycle. The measurement of iaload contains an additional iload_3 (as seen in Figure 2). Therefore, the iaload alone takes 35 cycles.

From the table we can see that OO bytecodes are quite expensive. Field and array access is in the range of 10 to 30 cycles. Invoke instructions are even more expensive and take around 100 clock cycles on JOP and the aJ100. In JOP the OO related bytecodes are implemented in microcode [10]. Similar to the way those bytecodes are implemented on a RISC processor without support for OO instructions.

3. A QUANTITATIVE APPROACH

Hennessy and Patterson's warning

> Virtually every practicing computer architect knows Amdahl's Law. Despite this, we almost all occasionally expend tremendous effort optimizing some feature before we measure its usage. Only when the overall speedup is disappointing do we recall that we should have measured first before we spent so much effort enhancing it!

was prominent in the 1st edition of [4] (1990) and is still included in the 4th edition (2006). We will follow their advice and first measure the potential speedup that can be gained by a hardware support for object-oriented instructions.

3.1 Amdahl's Law

Amdahl's Law gives us the tool to calculate the performance gain (*speedup*) of the complete system when part of the system is improved. The central message is that the overall speedup depends on the speedup of the enhanced feature *and* the fraction in time this feature is used.

The basic processor performance equation [4] gives us the CPU

Instruction	JOP	aJile
iload iadd	2	8
if_icmplt taken	6	18
if_icmplt n/taken	6	14
getfield	22	23
getstatic	15	15
iaload	36	13
invokevirtual	138	115
invokestatic	100	95
invokeinterface	144	153

Table 1: Bytecode benchmarks for JOP and the aJile aJ100 (Execution time in clock cycles)

time t_{exe} with the instruction count IC, the clock cycles per instruction CPI, and the clock cycle time t_{clock}:

$$t_{exe} = IC \times CPI \times t_{clock} \tag{1}$$

When we keep IC and the clock cycle time constant by our improvement we can get our speedup factor by just comparing the original CPI and the enhanced CPI.

With IC_i and CPI_i of instruction i the overall CPI is

$$CPI = \frac{\sum_{i=1}^{n} IC_i \times CPI_i}{IC} \tag{2}$$

We *just* need the individual CPI values and the instruction count or relative instruction frequency of each instruction. For the processor used in this example the CPI values are listed in [10]. We can measure IC_i by generating traces from benchmarks. However, (2) is a simplified version which does not include any pipeline stalls through data dependencies or cache misses.

With CPI_{stall} for cycles caused by pipeline stalls and CPI_{miss} for cache miss cycles CPI for a single scalar pipeline is

$$CPI = \frac{\sum_{i=1}^{n} IC_i \times CPI_i}{IC} + CPI_{stall} + CPI_{miss} \tag{3}$$

As CPI_{stall} and CPI_{miss} cannot be looked up in a data sheet or easily measured we take an indirect approach to estimate the overall speedup by architectural changes.

A simpler and more accurate approach is to artificially increase the execution time of a single instruction and measure the resulting execution time t_{new}. Using Amdahl's law with fraction enhanced f_i and speedup s_i of the enhanced feature (instruction) the overall application speedup s is

$$s = \frac{t_{old}}{t_{new}} = \frac{1}{(1 - f_i + \frac{f_i}{s_i})} \tag{4}$$

Hence we can evaluate f_i[4] for instruction i by

$$f_i = \frac{s_i}{1 - s_i}(\frac{1}{s} - 1) \tag{5}$$

This approach is very efficient when using an FPGA for the evaluation of the architecture. Increasing the execution time of an instruction is simple and we can run the benchmarks in real-time on the target. The same measurements within a VHDL simulation would just take too long and would not allow us to run the same amount of benchmarks.

[4]Note that f_i is the execution time fraction for bytecode i and not the instruction frequency.

Benchmark	JOP	aJile
Kfl	17120	14148
UdpIp	6781	6415
Lift	13574	–

Table 2: Application benchmarks results for JOP and the aJile aJ100 (in iteration/s)

Instruction	s_i	s	f_i
invoke	0.5	0.81	23.8%
getfield	0.5	0.89	11.9%
putfield	0.5	0.99	1.1%
getstatic	0.5	0.93	7.4%
putstatic	0.5	0.98	1.8%
xaload	0.5	0.89	12.1%
xastore	0.5	0.92	9.0%
all	0.5	0.59	68.6%

Table 3: Execution time fractions of OO bytecodes

3.2 Measurements

We use the embedded Java benchmark suite JavaBenchEmbedded[5] V1.1 for our measurement as described in [9]. The benchmark suite contains three application benchmarks: *Kfl*, *UdpIp*, and *Lift*. *Kfl* and *Lift* are two real-time applications in use in an industrial environment. The benchmarks are adapted by a simulation of the environment, i.e. sensor inputs, to generate a realistic execution profile. *UdpIp* is small UDP client-server application that tests an embedded TCP/IP stack, written completely in Java. We would prefer to use a standard benchmark, such as SPECjvm98 [13]. However, most of the SPECjvm98 benchmarks cannot run within the 1 MB memory we have available in our embedded system. Furthermore, all of them need a file system which is not available in many embedded systems.

The benchmark measures iterations per second, which means a higher value represents a better performance. Table 2 shows the results for a 100 MHz JOP version with 4 KB instruction cache and the aJile aJ100 [1] at 103 MHz. Both systems use 1 MB SRAM, with 15 ns and 10 ns access time respectively, as their main memory.

In our measurement we increase the execution time of a bytecode artificially by a factor of 2. Therefore, the speedup s_i for this bytecode is 0.5. All three benchmarks are combined by the geometric mean. The result is compared to the original, unaltered version. With the resulting application speedup s we calculate the fraction of the execution time f_i for instruction i with (5).

Table 3 summarizes the results. Column s gives the application speedup ($\frac{t_{old}}{t_{new}}$) and f_i the execution time fraction.

The row with *invoke* includes all four variants of invoke bytecodes (invokevirtual, invokestatic, invokeinterface, and invokespecial). getfield and putfield are the bytecodes to access object fields, getstatic and putstatic to access static fields (i.e. class fields). The rows *xaload* and *xastore* include all versions of array access (iaload, saload,...).

We also measured the slowdown when all object related bytecodes are slowed down by the factor of 2. The result is shown in the last row *all*. It shows that all OO instructions account for about 69% of the execution time in the current implementation of JOP for our three application benchmarks.

[5]Available at `http://www.jopdesign.com/`. The benchmark suit also contains the micro benchmarks for bytecode measurements as performed in Section 2.5.

Instruction	s_i	f_i	s
invoke	5	23.8%	1.24
getfield	2	11.9%	1.06
putfield	2	1.1%	1.01
getstatic	2	7.4%	1.04
putstatic	2	1.8%	1.01
xaload	2.33	12.1%	1.07
xastore	2.33	9.0%	1.05
all		67.0%	1.73

Table 4: Speedup estimates by enhanced OO bytecodes

The invoke instructions dominate the execution time of the OO related instructions. The speedup s of the applications is about 0.81 when the execution time for the invoke instructions are doubled ($s_i = 0.5$), resulting in f_i of 24%. This can be explained by two factors:

1. Methods in Java are usually very short and the execution frequency $\frac{IC_{invoke}}{IC}$ of the invoke instructions are relative high
2. The invoke instruction for Java is quite expensive. It involves saving the context on a stack frame, looking up the receiver method in a method table, looking up the number of arguments and local variables, and preparing the new stack frame. In the current implementation of JOP and similar Java processors, such as the aJile [1], the invoke takes around 100 cycles (see Table 1). In JOP this instruction is coded in microcode.

The next best candidate for optimization are the array access bytecodes (*xaload* and *xastore*) that account together for 21% of the execution time in the current implementation. What surprises a little bit is that array store operations account for almost the same execution time fraction as the array load operations. We expected that array loads would be more common than array stores.

The relation between load and store instructions (get and put in JVM bytecode terminology) is as expected. Loads are generally more common than stores and the execution time of the two instructions is similar (e.g. getstatic needs 14 cycles and putstatic 15 cycles on JOP with a memory with one wait-state).

From Table 3 we can see that the invoke instruction clearly dominates our execution time. It is therefore the prime candidate for optimization.

3.3 Estimations

Based on the measurement we can estimate the speedup by improving OO instruction through hardware support. It has to be noted that the estimated cycles in this section for enhanced operations are rough guesses as we do not know the details before we actually implement the enhancements. Actual cycle counts for the implementation of the array access are given in Section 4.

Knowing the execution time fraction f_i of instruction i we can estimate the speedup gained by improvement through hardware support. In the microcode for the invoke instruction a lot of bit manipulation is performed to extract information from the method information fields. This part can be easily speed up by hardware. Lookup of the method information can benefit moderately from hardware support. We estimate that the invoke instruction can be executed in about 20 cycles in hardware compared to 100 cycles

in microcode, resulting in a speedup s_i of 5. The estimation of 20 cycles in hardware is based on the inspection of the microcode that currently implements the invoke instruction.

The array access takes 35 cycles. Most of the cycles are spent for the null pointer check and the array bounds check. Performing this checks and the indirect memory access in hardware can probably be performed in 15 cycles ($s_i = 2.33$). For the object and class field access we assume a cut of the execution time by 2.

According to Amdahl's law we calculate the overall speedup s for each category. The results are shown in Table 4. The highest speedup of 1.24 can be gained by an enhancement of the invoke instruction. This instruction has a high individual speedup s_i of 5 and consumes 24% of the execution time in the original version. The next best candidates are the array access instructions.

The last row (*all*) shows the estimated speedup when we enhance all OO related bytecodes. It is calculated as follows:

$$s = \frac{1}{(1 - \sum f_i + \sum \frac{f_i}{s_i})} \quad (6)$$

4. EVALUATION

We evaluate the suggested OO hardware on JOP by implementing the array bytecodes in hardware. We have chosen the array bytecodes as they consume a relevant portion of the execution time, but are easier to implement than an invoke instruction. The implementation of array instructions also involves quite similar operations (e.g., null pointer check, pointer indirection) that are needed for the field access. Therefore, they provide the basis for the implementation of those instructions.

4.1 Array Load and Store

Arrays in Java are very similar to objects – actually the Java language definition calls them objects. Objects (and arrays) are allocated on the heap and are candidates for garbage collection (GC). For a compacting GC the arrays and objects have to be moved in the heap. To simplify the GC [12] an array/object reference does not point directly to the data. It points to a handle that contains the *real* pointer to the data. In case of a move only this handle and not all references have to be updated. When an object reference is used it has to be checked to be not *null* at runtime.

Another feature that makes Java a safer language is array bounds check. Those checks during runtime also add to the overhead. An additional memory access is necessary to get the size of the array on each access. Array access in Java is quite expensive when compared to C.

According to the JVM specification [7] following operations have to be performed for an array access:

- Check the reference against *null*
- Check array bounds
- Read or write the array data

Besides the checks an array load instruction needs three memory accesses:

1. Load the pointer to the array from the handle
2. Load the array size for the bound check
3. Load the *real* data

An array store needs two load operations and one store operation.

Instruction	s_i	s
xaload	3.5	1.10
xastore	3.2	1.06
xaload and xastore		1.18

Table 5: Speedup with array access hardware

The original implementation performs the memory operations and checks sequentially in microcode. The execution time in clock cycles for the array load and store bytecodes are

$$\begin{aligned} n_{load} &= 32 + 3r_{ws} \\ n_{store} &= 35 + 2r_{ws} + w_{ws} \end{aligned}$$

where r_{ws} and w_{ws} are the wait states for the external memory. On JOP at 100 MHz with 15 ns SRAM a memory access takes 2 clock cycles, meaning a single wait state. Therefore the array load instruction takes 35 clock cycles and the store 38.

4.2 Hardware Implementation

In our hardware implementation we still need to access the memory three times. However, we perform the checks concurrent to the memory access. The null pointer check can be performed while the real pointer to the array (the handle indirection) is loaded. When the original reference is null the load of a wrong pointer does not hurt. The data is just discarded.

The array size is part of the handle and not part of the data. It is at the address $reference + 1$. Therefore we can start this memory load even before we know the outcome of the first load (the pointer). During the size load the effective address is calculated (addition of the array index to the pointer). The index is also checked to be not negative at the same time (lower bound check).

When the array size is known the index can be checked for the upper bound. Then the actual array access is allowed. For the array load we can even perform the bounds check in parallel to the value load. That means we start the array load before we know the array size. In case of an array bound exception the wrong data is just discarded. This optimization is of course not possible on a write.

The resulting execution times for the array operations are

$$\begin{aligned} n_{load} &= 7 + 3r_{ws} \\ n_{store} &= 9 + 2r_{ws} + w_{ws} \end{aligned}$$

With r_{ws} and w_{ws} of 1 an array load takes 10 clock cycles and an array store 12 clock cycles. Therefore the speedup s_i is 3.5 and 3.17.

4.3 Measurements

During the change of the hardware we performed the benchmarks as described in Section 3.2. Table 5 shows the results. A hardware implementation of the array load instructions results in a 10% speedup and the enhanced store instruction in 6% speedup. Both values are higher than the estimates in Section 3.3. In the estimations we were conservative with the possible individual instruction speedups s_i. Implementing both instruction types in hardware gives an application speedup of 18%.

Table 6 shows the performance of the individual benchmarks in the original version and the enhanced version of JOP. The last column gives the resulting speedup s. The last row gives the geometric mean of the benchmark results and the speedup of this averages.

We can see that different applications benefit to a different extent from the enhanced array instructions. The *UpdIp* benchmark

Benchmark	original	enhanced	*s*
Kfl	17120	18347	1.07
UdpIp	6781	8520	1.26
Lift	13574	16425	1.21
geo. mean	11637	13693	1.18

Table 6: Individual benchmark speedup

	original	enhanced	ratio
JOP	2691	2906	1.08
jopcpu	2296	2485	1.08
mem.int.	89	304	3.42

Table 7: Processor and memory module size in logic cells (LCs)

contains a TCP/IP stack where a lot of buffer manipulation is performed (e.g. CRC calculation). Those buffer manipulations benefit from faster array access. At the other end of the spectrum is the *Kfl* benchmark. It is a control application written in a conservative programming style. It uses mostly static variables and performs a lot of control flow decisions and less array accesses.

4.4 Hardware Cost

One question remains to be answered. Is the investment in additional hardware for OO instruction support worth the speedup? In an FPGA the hardware size is measured in number of logic cells (LC) and number of on-chip memories. A LC, in the FPGA used for the evaluation, contains a 4-bit lookup table and a single register. The implementation of the array instructions needs additional LCs and no additional on-chip memory.

Table 7 shows the resource consumption of the original and enhanced version of JOP. The row *JOP* includes the processor and some minimal peripheral devices. The row *jopcpu* shows the size of the processor core without the peripheral devices. From the last column we see that the implementation of the array instructions in hardware increases the chip size by 8% – a moderate cost for the performance gain of 18%.

The main change of JOP is just in the memory interface module as listed in the last row (*mem.int.*). The core pipeline of the processor is, besides additional exception signalling, not touched by the change. The maximum frequency of the design is not affected by the change in the memory module. The critical path is still in the core pipeline.

5. CONCLUSION

In this paper we argue for computer architectural support of object-oriented languages. Some operations, such as virtual method dispatch, object field access, and array access, are quite complex and result in a lower performance of OO programs than procedural programs. To enable an OO oriented programming style in embedded systems, where computing resources are usually restricted, we propose a processor architecture with support of OO operations.

We have measured the fraction of time spent for OO instructions in a Java processor with microcode implementation of those instructions. About 69% of the execution time is spent on those instructions. We estimated a possible whole application speedup of 1.7 when those instructions are supported by the micro-architecture.

For an evaluation of the proposed architecture we implemented array load and store bytecodes within JOP, a Java processor. The individual instruction speedup is larger than 3 and resulted in a application speedup of 1.18. The hardware cost for this micro-architecture change is about 8% of the whole processor. Therefore we got 18% speedup for just 8% more transistors.

As future work we plan to implement the field instructions and invoke instructions in hardware. We expect that the field access will be cheap in hardware based on the array access already implemented. Both operations can be implemented in the memory access module and need no change in the core pipeline. However, the invoke instructions interacts with the on-chip stack cache and the registers (e.g. stack pointer, variable pointer). The invoke implementation requires changes in the core pipeline and we assume a higher hardware cost. However, the estimated speedup of another 24% is a good argument to pay for the additional hardware.

6. REFERENCES

[1] aJile. aj-100 real-time low power Java processor. preliminary data sheet, 2000.

[2] T. R. Halfhill. Imsys hedges bets on Java. *Microprocessor Report*, August 2000.

[3] D. S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 53. IEEE Computer Society, 2001.

[4] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[5] Imsys. Im1101c (the cjip) technical reference manual / v0.25, 2004.

[6] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, 2003.

[7] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[8] J. M. O'Connor and M. Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.

[9] M. Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.

[10] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[11] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, doi:10.1016/j.sysarc.2007.06.001, 2007.

[12] M. Schoeberl and J. Vitek. Garbage collection for safety critical java. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, 2007.

[13] SPEC. The spec jvm98 benchmark suite. Available at http://www.spec.org/, August 1998.

[14] Y. Tan, C. Yau, K. Lo, W. Yu, P. Mok, and A. Fong. Design and implementation of a java processor. *Computers and Digital Techniques, IEE Proceedings-*, 153:20–30, 2006.

[15] N. Vijaykrishnan and N. Ranganathan. Supporting object accesses in a Java processor. *Computers and Digital Techniques, IEE Proceedings-*, 147(6):435–443, 2000.