USENIX Association

Proceedings of the Java™ Virtual Machine Research and Technology Symposium (JVM '01)

Monterey, California, USA
April 23–24, 2001

On the Software Virtual Machine for the Real Hardware Stack Machine

Takashi Aoki
Autonomous System Laboratory,
Fujitsu Laboratories Limited

Takeshi Eto
Semiconductor Group,
Fujitsu Limited

Abstract

Several technologies for Java[1] [1] program execution have been reported, e.g., Just-In-Time (JIT) compilation, pre-compilation engine, etc., to improve its running speed. Bytecode engine is another approach by taking advantage of the hardware acceleration.

This paper is concerned with the brief introduction to the picoJava-II core technology and its implementation at Fujitsu. Then, we will present our software mapping approach onto the hardware stack machine, focusing on Fujitsu's picoJava-II implementation, MB86799 [2]. Finally, we will report some benchmark results of Java virtual machine execution on the real stack machine and discuss yet unresolved problems.

1 Introduction

Our work aims at combining hardware bytecode stack machine with software Java virtual machine. This section explains how picoJava-II architecture works to help you understand our software development and porting work.

[1] Java, picoJava, JDK, J2ME, and PersonalJava are trademark or registered trademark of Sun Microsystems, Inc.

[2] MB86799 is an evaluation chip. Note that our project was not targeted on the specific end-user product, such as mobile phones or PDA's

1.1 picoJava-II Architecture

Sun's picoJava-II core is a Java bytecode execution engine. First of all, we briefly describe the architecture of picoJava-II core for the better understanding of our software approach.

McGhan et al. [2] reported the core technology of the picoJava architecture. Sun published the Programmer's Reference Manual[3]. Data sheets and other technical documentation are available through Sun Community Source Licensing (SCSL) program[4] as well.

1.1.1 Registers

The picoJava-II architecture has registers for specific purposes as depicted in Table 1. Among these, it is notable that the stack registers, the constant pool register, the monitor caching registers, and the garbage collection register are designed to contribute to the performance improvement of Java program execution. Their access can be realized by simple register read/write operation in picoJava-II. On the other hand, these registers are assigned to the variables in the software virtual machine implementation, such as JDK, instead.

Table 1: picoJava-II Registers

Category	Register
Program Counter	**PC**
Stack Registers	**VARS** **FRAME** **OPTOP** **OPLIM** **SC_BOTTOM**
Constant Pool	**CONST_POOL**
Memory Protection	**USERRANGE1** **USERRANGE2**
Program Status	**PSR**
Monitor Caching	**LOCKCOUNT0** **LOCKCOUNT1** **LOCKADDR0** **LOCKADDR1**
Trap	**TRAPBASE**
Garbage Collection	**GC_CONFIG**
Breakpoint	**BRK1A** **BRK2A** **BRK12C**
Global	**GLOBAL0** **GLOBAL1** **GLOBAL2** **GLOBAL3**

1.1.2 Instructions

The picoJava-II supports 226 bytecode instructions which Java Virtual Machine Specification [5] defines and 115 extended instructions added in picoJava-II Programmer's Reference Manual. The extended instructions are capable of either one of the following functions.

- Direct memory access
- C language interface
- Cache manipulation

1.1.3 Traps

Trap handlers can be divided into the following categories.

- Instruction emulation
- Hardware origin
- Monitor cache interface
- Garbage collection interface

Some of the Java bytecode instructions are too complicated to implement in hardware, such as *new*. This is originated in the fact that these instructions are strongly related to the constant pool resolution. Therefore, the system programmers should write the emulation code for such instructions for picoJava-II and the specific Java platform.

Similar to the conventional microprocessors, the picoJava-II core notifies the error condition in hardware as a trap of the software. The errors include wrong memory alignment, illegal instruction, and so on.

As we see above, the picoJava-II core supports not only the monitor cache register but also its bytecode instructions, *monitorenter* and *monitorexit* in hardware. When the exception occurs in the monitor cache interface of hardware, a trap is notified to the software. Then, it is up to the software's task to continue the appropriate action for the monitor.

The picoJava-II core has a register for the garbage collector. When the garbage collection timing can be observed by the hardware, this is also notified to the software.

1.2 The picoJava-II core features

The picoJava-II core directly executes Java bytecode instructions defined in picoJava-II Programmer's Reference Manual and supports extended bytecode instructions which enable a direct memory access. Its instruction set also includes the *_quick* instructions by which the resolved object can be handled in a much faster manner. The instructions that require constant pool resolution are processed in the trap handler software followed by the trap. The picoJava-II core has the following features to improve the Java application performance.

- Stack cache

- Stack dribble
- Instruction folding
- Write barrier support
- C function interface

1.2.1 Stack Cache

As the Java VM Specification defines a stack machine architecture, the picoJava-II core is based on a stack machine architecture. Software implementation JVM, such as JDK, uses a straightforward memory area as the Java stack. On the other hand, the picoJava-II core takes advantage of the hardware cache for the stack. This improves the bytecode execution performance. The Java stack cache of the picoJava-II core consists of 64 word entries. Once the stack pointer grows/shrinks beyond hardware limits, stack dribbling starts and the contents of the cache are moved to and from the memory. Therefore, the stack cache is transparent to the software.

1.2.2 Stack Dribble

The picoJava-II caches the top 64 entries of the stack. When the software accesses the stack within this range, the core takes advantage of benefits from the high performance of the stack access. Here, one of the following two operations will be started by the hardware. The *spill* writes entries in the stack cache out to the data cache to make some space for new value, while the *fill* reads entries into the stack cache from the data cache to provide the value for the coming instruction. These are operated concurrently in the background.

1.2.3 Instruction folding

The picoJava-II core provides a solution to the access inefficiency problem of stack machines. Its stack cache is actually a full random access register file, so the pipeline is capable of the immediate access of all 64 entries in the stack cache.

```
iload_1          add local1,local2,local3
iload_2
iadd
istore_3
```

(a) (b)

Figure 1: Instructions to Add Two Locals

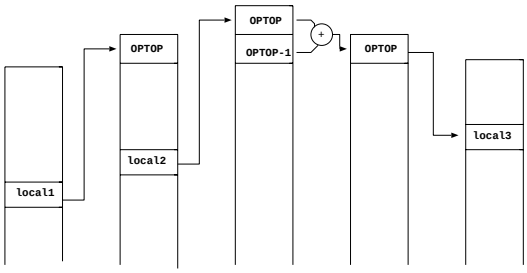

Figure 2: Execution without Instruction Folding

The execution technique called *instruction folding* takes advantage of this. When two values are added in the stack machine, the instruction sequence depicted in Figure 1 (a) is used. This is also visualized in Figure 2.

However, once the hardware detects the two entries are in the stack register file, it can add the values and store the result to the stack cache in one sequence with the instruction folding technique as seen in Figure 1 (b) and Figure 3.

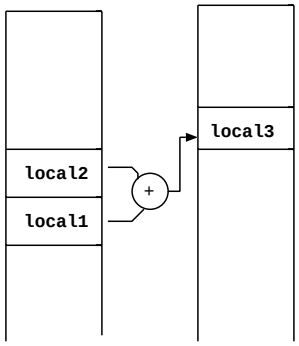

Figure 3: Instruction Folding

1.2.4 Write Barrier Support

The picoJava-II core has the garbage collection accelerating facility called *write-barrier* register. When the accessed object resides in the mem-

ory area beyond the write-barrier, the picoJava-II core detects it and notifies the software of the event by *gc_notify* trap. The garbage collection condition can be controlled by **GC_CONFIG** register. Utilizing this register is strongly related to the garbage collection algorithm ([8] and [9]).

1.2.5 C Function Interface

While the Java method is invoked by the method invoke instructions, such as *invokevirtual* or *invokespecial*, as Java Virtual Machine Specification defines, C language functions invocations relies on the platform dependent implementation. The *call* instruction is added to the picoJava-II core to support the C *native* function. This also accelerates the Java Native Interface (JNI) method invocation as well as the internal JVM functions written in C.

1.3 Fujitsu MB86799

Fujitsu's implementation of picoJava-II technology, MB86799 [3] , consists of picoJava-II core, the external bus interface, and PCI bus interface. This implementation has an 8KB instruction cache, an 8KB data cache, a 64-entry stack cache, and a floating point unit as shown in Figure 4. The MB86799 can run at the external maximum frequency rate 33MHz and internal 66MHz. The frequency ratio between the internal and the external can be varied from 2 to 5. The chip consumes 360mW with the source 2.5V at 66MHz.

2 Overview of the Software

We now present a high-level overview of the software implementation on our picoJava-II chip.

We ported Sun's PersonalJava 3.02 to picoJava-II running Fujitsu's real-time OS, REALOS[4], a variant of μ-ITRON RTOS. Figure 5 is an overview of our system software.

[3]MB86799 is based on β-version specification of the picoJava-II core. There is a minor difference of the instruction operation between the β version and the current FCS release.

[4]REALOS is a trademark of Fujitsu Limited.

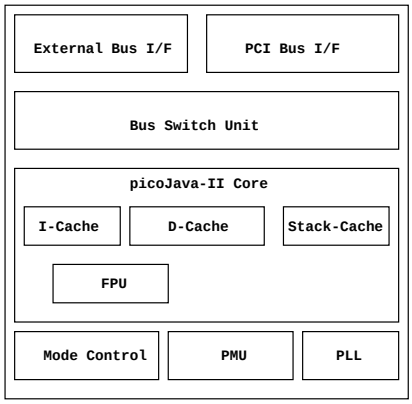

Figure 4: MB86799 Block Diagram

We did not write our own software virtual machine for picoJava-II from scratch, instead we ported PersonalJava because of the following reasons.

- Porting was estimated to be finished in a shorter time.
- Scratch-built virtual machine would be more difficult to accomplish the compatibility with reference software.

The source code of the latest PersonalJava, namely PersonalJava 3.1, can be obtained through SCSL[6].

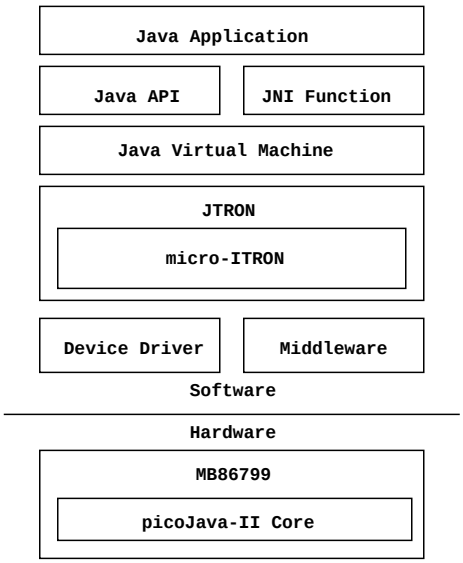

Figure 5: Overview of Software Structure

2.1 Operating System

We were motivated to use the REALOS operating system in order to meet the strong demand for real-time assurance support from the embedded device community. REALOS is μ-ITRON RTOS specification compliant. All the operating facilities are provided by REALOS and our JVM takes advantage of them. The μ-ITRON specification and its Java interface variant, JTRON, specification are available at TRON Project Home Page[7].

2.2 Java Virtual Machine

Now that we are able to execute the bytecode on hardware directly, the term *virtual machine* may be somewhat incorrect. However, since no appropriate naming has been proposed so far, let us continue to call it *virtual machine* at the moment. As mentioned above, our virtual machine is based on PersonalJava 3.02. The five items we had to consider while porting the virtual machine to the picoJava-II chip are as follows.

1. Object data structures
2. Exception handling
3. JNI support
4. Lock register
5. JCC (JavaCodeCompact)

3 Porting Strategy

We show here our strategy to port PersonalJava onto picoJava-II architecture, focusing on how to adapt software virtual machine for direct bytecode execution engine. We use C and assembler language to implement the virtual machine.

3.1 Object Data Structure

The picoJava-II core directly executes the Java method and handles the Java object. This implies objects must be placed on memory in the exact format as the core expects. Since PersonalJava is derived from JDK, its object format is different from the one that picoJava-II specifies. Figure 6 and 7 are the examples of the fixed object format for PersonalJava and picoJava-II respectively. In picoJava-II, the array reference always points to the array header position, which is followed by an array length. The actual array data follows after the length field. The array contents can be accessed by a reference and indexing.

We modified the array structure definitions and their access macro definitions of PersonalJava so that picoJava-II hardware can manipulate the array structure directly.

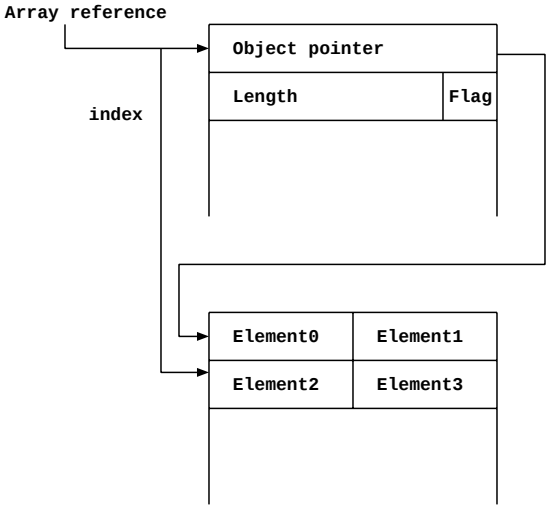

Figure 6: An Example of PersonalJava Array - short primitive

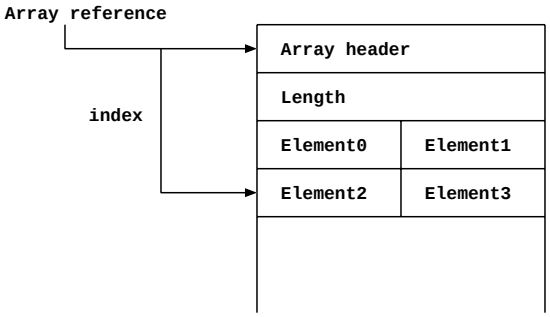

Figure 7: An Example of picoJava-II Array - short primitive

3.2 Exception Handling

We now describe the following four topics on the difficult handling of the exception for picoJava-II.

1. Stack Usage
2. Stack Cache Coherency
3. Stack Growing Direction
4. Exception Origin

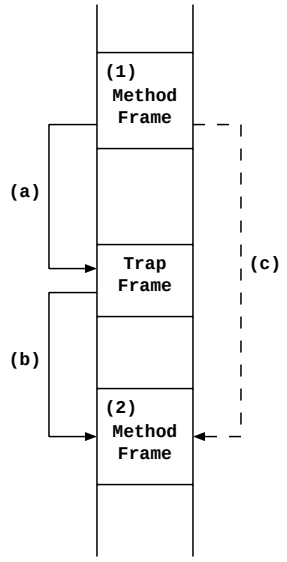

Figure 8: Exception Table Trace

3.2.1 Stack Usage

We noticed the difference of stack usage between the JDK/PersonalJava JVM implementation and the picoJava-II handling. While the former uses the two distinct stack, i.e., C stack and Java stack, in its memory area, the latter uses only one stack which is shared among JVM internal C functions, the JNI method, and the Java method. In the former case, the JVM simply traces the frame structure in the Java stack to find an appropriate exception handling method when exception occurs. However, in picoJava-II, it is not that straightforward. The picoJava-II core defines three kinds of frame format, Java method frame, trap handler frame and C function frame. When an exception occurs, the exception handler search routine tries to locate the Java method frame that catches it, skipping trap handler frames and C frames. Both frames are not a concern for Java exception.

In addition, in our approach, the JNI method is implemented by the Java stub method as described below. Therefore, we have to ignore the Java stub method frame for the JNI call as well.

Figure 8 is an example of the exception tracing. Here, the method with the frame (2) invokes another method (1) via a trap frame. Each frame is linked as (a) and (b). As explained above, the trace method routine needs to behave as if the method frames were linked as (c) in order to trace them. JNI case can be explained by substituting the trap frame by a JNI stub frame.

3.2.2 Stack Cache Coherency

There is another difficulty for traversing method frames on the picoJava-II stack. When we scan the stack, we usually use a pointer addressing the stack. To do that on picoJava-II core, we have to flush the stack cache before accessing the stack frame, since there is no coherency between its stack cache and the data cache.

But flushing the stack cache is a time consuming operation. By using *load*/*store* instructions [5] , one can access the stack data correctly without a stack cache flush. To traverse method frames, however, we must set the **VARS** register to an appropriate address successively. This is also cumbersome, as we can't use local variables while changing the **VARS** register.

We used the stack cache flush scheme for PersonalJava 3.02 port, as it was simpler.

3.2.3 Stack Growing Direction

The stack growing direction also prevents us from manipulating frame data directly. The picoJava-II

[5] Local variables are accessed by the offset index from **VARS** with *load*/*store* instructions, such as *iload_0* and *astore_1*. **VARS** is a register that points to the first argument address in its Java stack or C stack.

Java stack grows downward, i.e., from the higher address to the lower. Figure 9 shows Java frame format (a) and trap frame format (b) in picoJava-II stack. Note that **FRAME** register always points to the previous or return **PC** address and the position is located at the highest offset or the middle of the data.

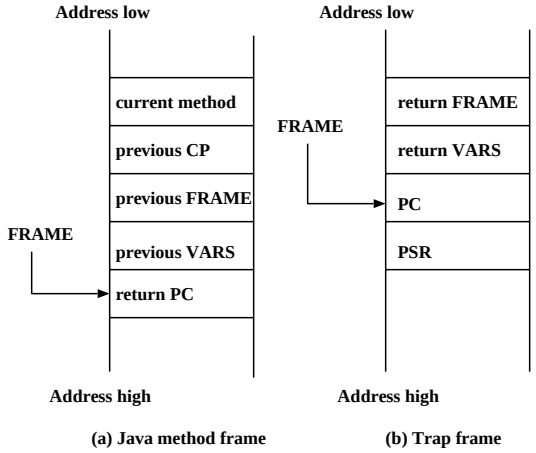

Figure 9: Stack Growing Direction

```
struct javaFrame {
   METHOD        *currentMethod;
   CONSTPOOL     *previousCONSTPOOL;
   FRAME         *previousFRAME;
   VARS          *previousVARS;
   unsigned char *returnPC;
};
```

Figure 10: Java Frame in C Structure

On the other hand, when we refer the stack frame by a data structure in C language, a pointer for the structure always points at the lowest address. Thus, when we trace the Java frame data in the high level language such as C, we have to declare the frame structure in the reverse order from the picoJava-II core definition, as seen in Figure 10. Then we have to recalculate the next frame's start address using `previousFRAME` field value every time we move the pointer.

3.2.4 Exception Origin

Some of the exceptions, e.g., NullPointerException, can be raised by both software and hardware in picoJava-II. When the instructions detect the null reference, the hardware automatically raises the exception trap. This has initially been the responsibility of the virtual machine software, and it still exists on picoJava-II. One has to take into account both of the exception trap case and the software originated exception to save the code space.

3.3 JNI

Again the term JNI is not an exact term since the native instruction set of picoJava-II is the bytecode. But let us call the interface between a Java method and a function written in C, JNI here.

As shown in Figure 11, the calling convention of Java method is different from that of C function on picoJava-II. Re-pushing the argument variable to the stack is necessary when the C function is invoked from a Java method. The method is invoked by invoke instructions, for example, *invokevirtual_quick*, on picoJava-II, whether it is written in C or Java. In other words, the method is invoked in a different manner depending on its access flag as defined in Java Virtual Machine Specification when implemented by software. However, the hardware picoJava-II does not care about such a flag but always invokes the method as if it were written in Java.

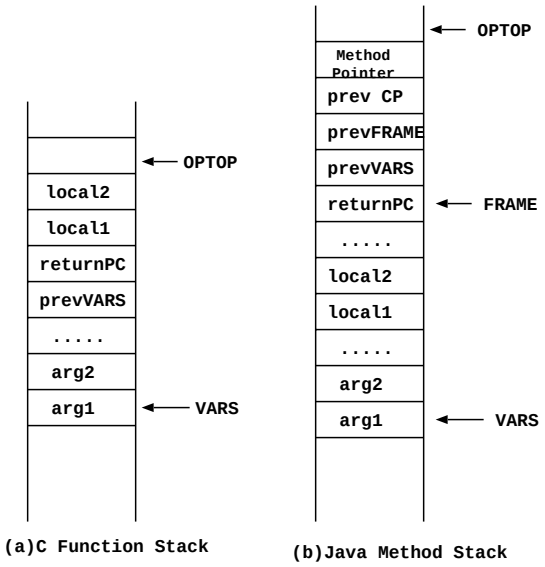

Figure 11: C and Java Stack Usage

There are two approaches to resolve this problem as explained in picoJava-II Reference Manual .

- Create a stub method between the calling Java method and the callee C function
- Invoke the C function from the trap software code of *invoke* instructions

We chose the former strategy since apparently the latter sacrifices the performance without using the *_quick* instruction of hardware.

3.4 Lock Register

Java virtual machine controls the shared object data with a monitor by locking with mutex. The picoJava-II core has two pairs of **LOCKCOUNT** and **LOCKADDR** registers for this purpose. The *monitorenter* instruction has the following logic:

- Compare the object address with the value of either one of **LOCKADDR** registers.
- If the address matches, increment the corresponding **LOCKCOUNT** register.
- Otherwise, the trap handler is invoked.

This indicates that when the object is not actually shared, the mutex is accomplished merely by setting the object address to the **LOCKADDR** register and incrementing or decrementing the contents of **LOCKCOUNT** register. As a result, the overhead for the monitor creation is significantly reduced.

The virtual machine also needs to manipulate the *synchronized* method code in order to utilize the hardware *monitorenter* and *monitorexit* instructions as follows.

1. Replace all *return*, *areturn*, *ireturn*, *lreturn*, and *dreturn* instructions with the extended *exit_sync_method* instruction. [6]
2. Insert code set shown in Table 2 and 3, depending on the attribute of the method.
3. Change the exception table for the method so that `start_pc`, `end_pc`, and `handler_pc` of the entries are each incremented by 8 or 12 to map onto the relocated code.

[6] All the bytecodes in the method must be scanned in order to locate and replace the *X*return instruction due to the variable length of the bytecode instruction set. Analysis involving *lookupswitch*, *tableswitch*, and *wide* is often complicated.

Table 2: Code Prepended to Synchronized Non-static Methods

Address	Instruction	Action
0	*aload_0*	Get the object reference
1	*monitorenter*	Synchronize on the object reference
2	*jsr*+6	Push PC on top of the stack, which is also **FRAME** - 20, and jump to the next instruction.
5	*aload_0*	Get the object reference
6	*monitorexit*	Exit the monitor.
7	*Xreturn*	Return to the caller of the correct type.

Table 3: Code Prepended to Synchronized Static Methods

Address	Instruction	Action
0	*get_current_class*	Get the current class pointer.
2	*monitorenter*	Synchronize on the class pointer.
3	*jsr*+9	Push PC on top of the stack, which is also **FRAME** - 20, and jump to the original code.
6	*get_current_class*	Get the current class pointer.
8	*monitorexit*	Exit the monitor.
9	*Xreturn*	Return to the caller of the current type.
10	*nop*	
11	*nop*	Ensure that the code is a multiple of 4 bytes to prevent changes in padding for lookupswitch and tableswitch.

In our implementation, the code is manipulated in the class loader. However, if the system developer is aware that the synchronized methods are rarely called, the manipulation can be done at the time of the method invocation, such as in the trap handler.

3.5 JavaCodeCompact (JCC)

PersonalJava has a tool called JavaCodeCompact (JCC) to improve the class loading performance and reduce the code size. The tool converts the class file into the runtime memory image and links it with the virtual machine. Since the picoJava-II has the unalterable object format which the hardware accepts, and the format differs from that of the software implementation, we had to modify the internal data structure and code generation part of the tool accordingly as well.

4 Benchmark Results

Table 4 shows the Embedded CaffeineMark benchmark result of MB86799. The table contains the actual results. PJEE is the emulation environment of PersonalJava, which has an interpreter loop written in C. We used JDK1.1.8 to compare with a JIT compiler result. But please note that the implementation, especially the data structure, is different from PJEE. The number indicates that the larger it is, the faster the platform is.

Table 4: Embedded CaffeineMark Results

	MB86799 66MHz /33MHz PJ 3.02	Pentium-III 700MHz /100MHz PJEE 3.02	Pentium-III 700MHz /100MHz JDK 1.1.8
Sieve	395	367	11644
Loop	984	339	40248
Logic	667	336	168150
String	594	957	18867
Float	407	333	18981
Method	593	362	18116

the larger the faster

Table 5 is the results of the Java version of Tak function[11]. The Tak is a heavy recursion test program to evaluate function call performance. Our experiment was undertaken with the function call of the argument Tak(18,12,6) for 1000 times.

Table 6 shows the results of Java vesion of Linpack [12] benchmark program.

Table 7 is the result comparison of SPECjvm98 [13] benchmark programs. The number here indicates that the smaller, the faster the platform is. Note that some of the SPECjvm98 programs are not listed here since they are too large to run on our embedded system. Please note that Pentium-III runs more than ten times faster than picoJava-II for internal clock and three times faster for external bus clock.

Table 5: Tak Benchmark Results

MB86799 66MHz /33MHz PJ 3.02	Pentium-III 700MHz /100MHz PJEE 3.02	Pentium-III 700MHz /100MHz JDK 1.1.8
24392msec	38670msec	1593msec

the smaller the faster

Table 6: Linpack Results

MB86799 66MHz /33MHz PJ 3.02	Pentium-III 700MHz /100MHz PJEE 3.02	Pentium-III 700MHz /100MHz JDK 1.1.8
1.798Mflops/s	1.761Mflops/s	22.889Mflops/s
0.38sec	0.39sec	0.03sec

Table 7: SPECjvm98 Results

	MB86799 66MHz /33MHz PJ 3.02	Pentium-III 700MHz /100MHz PJEE 3.02	Pentium-III 700MHz /100MHz JDK 1.1.8
jess	1109.189	164.597	51.674
mpegaudio	400.398	610.888	14.661
mtrt	786.019	174.260	30.564
jack	1534.797	204.043	31.295

the smaller the faster

Among the benchmark results here, the Java version of Linpack runs almost as fast on picoJava-II as on JDK JIT, if the number is normalized by internal clock. Figure 12 is the bytecode statistics during its execution. The bytecode instructions here are categorized as follows:

- Stack manipulation
  iload,*aload*,*istore*,*astore*,*push, pop*,*dup*,*inc*, *iconst_0*, etc
- Arithmetic implemented in hardware
  iadd, *fmul*, *dmul*, etc

- Array load/store
  caload, *bastore*, etc
- Conditional branch
  ifeq, *if_acmpeq*, etc
- Unconditional branch
  goto, *ireturn*, etc
- Field access
  getstatic, *putfield*, etc

All of these categories are implemented in hardware on picoJava-II, and consists of 96.7% of the instructions executed. Obviously, this is very advantageous in achieving good Java application performance on picoJava-II. Figure 13 is the bytecode composition of Linpack class file. 93.2% instructions of the class file are implemented directly in hardware on picoJava-II. The native methods, which often become the bottle neck for picoJava-II execution performance, invoked by Linpack are java.lang.System.currentTimeMillis() and some string manipulation method to produce the result message. The fact implies Java program performance for picoJava-II can be evaluated statically in advance by analizing the instruction category.

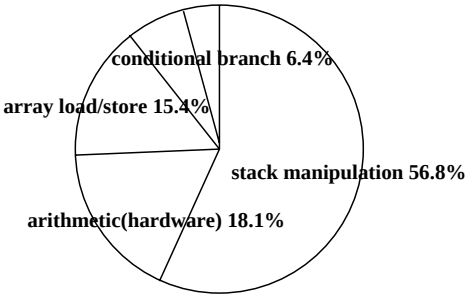

Figure 12: Linpack bytecode execution statistics

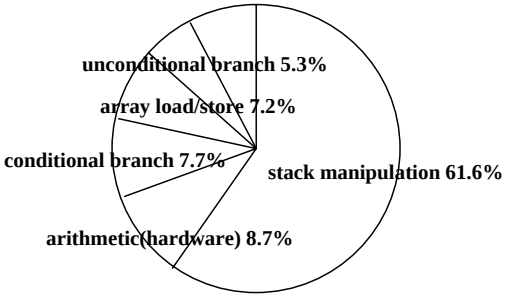

Figure 13: Linpack bytecode composition

The ratio of the stack manipulation instructions of Embedded CaffeineMark is as much as that of Linpack, as seen in Figure 14. However, when the hardware implemented field access instruction, such as *getfield_quick*, is operated, its corresponding emulated instruction, *getfield*, for example, is executed beforehand. In addition to this, a large amount of time is spent on the string manipulation native methods written in C for this benchmark test. This is fairly disadvantageous for picoJava-II technology.

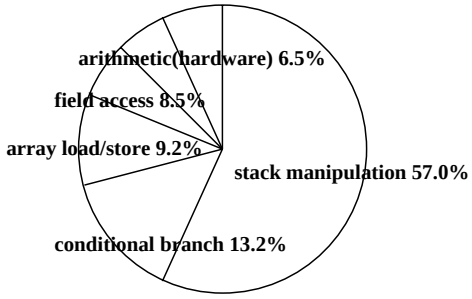

Figure 14: Embedded CaffeineMark bytecode execution statistics

Our benchmark results indicate that our approach on the real hardware stack machine overwhelms the performance of the corresponding C interpreter. In addition, the direct execution of bytecode compares competitively with JIT enabled virtual machine for the bytecode oriented Java applications.

The Java microprocessor technology can be an adequate solution to the strong demand from the embedded Java community.

5 Future Work

As reported by Gu et al. [10], there are a number of JVM code tuning techniques. These can be also applied to our virtual machine since ours is derived from Sun's JDK implementation. For instance, the String result of Embedded CaffeineMark benchmark was improved by about 10% with a few function inlining within EE() function in our experiment. This is still experimental in our work and we continue the code tuning focused on the specific implementation.

We have not used the power of the picoJava-II technology to its full extent yet, especially in its garbage collection support area. We will continue to improve the performance of PersonalJava platform. Besides this, the current PersonalJava is based on JDK 1.1.x technology. However, the present embedded Java community request varies in wide range. Porting the J2ME platform will be one of our main concerns in the near future.

6 Open Problems

There have still been open problems found in our experience. In this section, we report them for the future hardware design and its tool improvement.

6.1 Stack Cache

The current picoJava-II technology heavily depends on the cache access for its performance improvement. However, because of the difficulty of predicting the cache residency, the software performance sometimes varies with a minor modification. This often makes system performance tuning difficult. Besides, the lack of coherency between the stack cache and the data cache often complicates the software designing, especially for those that access the local stack directly, such as trap handlers for the bytecode instruction emulation, for example *invokeinterface* or *getfield*.

6.2 C Language Interface

At the time of writing of this paper, the only C compiler which supports the picoJava-II extended instruction set is MetaWare's High C/C++[7]. The compiler generates the picoJava-II C function interface object code in a straightforward manner. In other words, while the compiler generates the function code with *call*/*return* by register interface, the Java class method expects the code with *invoke*/*return* by stack interface.

Figure 15 depicts an example of this problem. Initially, Java method (3) invokes the C function (1) as a native method. Although the object code of the function (1) returns the value by setting it to register **GLOBAL1**, the caller method (3) expects the value in the Java stack. In our approach, we insert the stub method (2) between them.

One would notice that there are a number of C native methods in the Java system class. For example, java.lang.Math.pow() function only returns the native pow() function result. As we explained earlier, we have two options to support JNI on picoJava-II, creating a stub method or call the C function in the trap handler without using the *_quick* hardware instructions. If the C compiler could compile a C function following the Java method calling interface, the JNI performance would be improved significantly.

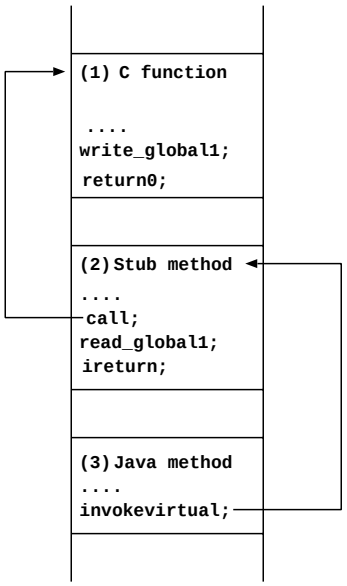

Figure 15: C-Java Language Interface Problem

6.3 Aggregate Stack Problem

The picoJava-II core provides a stack named *aggregate stack*[8] for the C language local variables whose addresses are used like arguments passed by reference. Aggregate stack is designed to solve the stack cache incoherency problem. However, this is sometimes inclined to increase the overhead in the JNI functions.

The stack often complicates the system program-

[7] High C/C++ is a trademark of MetaWare, Inc.

[8] The aggregate stack is a portion of memory area pointed by **GLOBAL0** register.

mer's work because it is difficult to handle the aggregate stack both by the C compiler and the hand-written assembler code. Also, system programmers are required to be careful that the aggregate stack must be released appropriately when an exception occurs in a Java method and the exception is caught beyond a JNI function that uses the aggregate stack.

7 Conclusion

In this paper, we have shown our porting strategy of the software Java virtual machine onto the real hardware stack machine using picoJava-II microprocessor as an example.

The direct bytecode execution engine demonstrates that it can be even competitive with JIT enabled software virtual machine, especially for the bytecode oriented applications. Our benchmark results also indicate that Java microprocessor can be one of the effective solutions for the embedded Java technology where low power consumption, small memory footprint and quick start are preferred.

The picoJava-II core is a well defined CPU to run not only Java but also C, although there still exists some room for improvement. Some will be solved by hardware modification and others by improving the C compiler.

References

[1] Gosling, J., Joy, B., Steele, G., The Java Language Specification, Addison Wesley, 1996

[2] McGhan, H., O'Connor, M., PicoJava: A Direct Execution Engine For Java Bytecode. In IEEE Computer Vol 31, No 10, October 1998

[3] Sun Microsystems, Inc., picoJava-II Programmer's Reference Manual, March 1999

[4] Sun Microelectronics, picoJava Microprocessor Cores, *http://www.sun.com/microelectronics/picoJava/*

[5] Lindholm, T., Yellin, F., The Java Virtual Machine Specification, Addison Wesley, 1997

[6] Sun Community Source Licensing, Source Code Catalog, *http://www.sun.com/software/communitysource/*

[7] The ITRON Project, JTRON2.0 Specification, *http://tron.is.s.u-tokyo.ac.jp/TRON/ITRON/home-e.html*

[8] Grarup, S., Seligmann, J., Incremental Mature Garbage Collection, M.Sc. Thesis, Aarhus University, Computer Science Department, August 1993.

[9] Hudson, R., Moss, J.E.B., Incremental Garbage Collection For Mature Objects, Proceedings of International Workshop on Memory Management, St. Malo, France, September 16-18, 1992

[10] Gu, W., Burns, N.A., Collins, M.T., Wong, W.Y.P., The Evolution of a high-performing Java virtual machine, IBM Systems Journal, Vol. 39, No. 1, 2000, IBM Corporation

[11] Gabriel, R.D., Performance and Evaluation of Lisp Systems, The MIT Press, 1985

[12] Linpack Benchmark - Java Version, Netlib Repository, *http://www.netlib.org/benchmark/linpackjava/*

[13] SPEC JVM98 Benchmarks, *http://www.spec.org/osg/jvm98/*