

CPU-INDEPENDENT ASSEMBLER IN AN FPGA

Georg Acher, Carsten Trinitis

Rainer Buchty

Lehrstuhl für Rechnerarchitektur und Rechnerorganisation
Fakultät für Informatik
Technische Universität München
85747 Garching bei München, Germany
email: [acher|trinitis]@cs.tum.edu

Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung
Institut für Technische Informatik
Universität Karlsruhe (TH)
76128 Karlsruhe, Germany
email: buchty@ira.uka.de

ABSTRACT

We describe a system which enables FPGAs to generate machine code for various CPUs, similar to a conventional assembler. Such conversion from intermediate code to a CPU's native code can be used as the last step in just-in-time compilation for virtual machines like the Java Virtual Machine. The translation system itself and the FPGA logic are independent of the actual target CPU and can be used with both CISC and RISC CPUs. Due to an extended table lookup, the resulting code is very efficient and gains from pre-calculation of selected constants in the FPGA assembler.

1. MOTIVATION

With increasing success of the Java platform and its underlying Java Virtual Machine [1], just-in-time compilation (JIT) techniques gained added interest. JIT performs a transformation of virtual machine code into the native code of the executing processor. This yields much higher execution speed compared to the much simpler, but slower interpretation of the VM instruction stream.

Although execution speedup of JIT systems can be impressive (factors from 5 to 50 are possible), they require a considerable amount of resources for compiling native code, possibly resulting in noticeable pauses during compilation as well as increased memory use and CPU consumption. For desktop systems this can usually be neglected. Embedded systems, however, show limited resources so that use of JIT systems becomes prohibitively expensive and does not account for gained speed-up.

Several hardware-assisted systems for faster JVM execution were proposed. Implementing a JVM-CPU in hardware is one possibility as demonstrated by picojava [2] and JOP [3]. Also, a regular CPU can be modified to directly interpret JVM opcodes as done by ARM with their Jazelle Technology [4]: with only modest addition of logic to the instruction decoder, the JVM can be emulated efficiently and without sacrificing execution of "legacy" code.

A related approach for JIT compilations in restricted environments is the JIFFY concept [5], performing JIT compilation for the stack-based JVM almost entirely FPGA-based.

This paper focuses on an important phase of JIFFY: The transformation of intermediate code into native code of the target CPU. This last step in the compilation process is performed very quickly and in parallel to the target CPU. The transformation logic is CPU independent and based on an extended table lookup concept, hence heterogeneous CPU systems are supported as well as other intermediate codes.

This paper is organized as follows: In Section 2 the JIFFY system is introduced, Section 3 describes the used intermediate code. Section 4 is giving a detailed description of the FPGA assembler architecture, Section 5 presents a preliminary implementation scenario for typical CISC and RISC.

2. THE JIFFY SYSTEM

In JIFFY, a full JIT compilation from JVM code to native code is performed in an FPGA. After analyzing the original JVM code for branches and certain objects, the opcodes are converted into an intermediate language called IJVM. IJVM opcodes are almost lookalikes of JVM instructions, but work on registers. The original JVM stack is emulated in the IJVM with push/pop operations. Since this trivial conversion contains a lot of push/pop opcodes, some optimization phases follow: based on about 18 rules in total, almost all occurrences of push/pop can be eliminated (replaced by register transfers) by an FPGA-based peephole optimizer. Then the intermediate code is converted to native assembly code which subsequently gets linked.

With respect to current compiler techniques such as Sun's Hot-Spot [6], this system is very simple and can be implemented efficiently in an FPGA. The most important advantageous aspects of this concept can be found in the parallel work of FPGA and CPU, and in a very fast translation (10^6 JVM opcodes/second at 50MHz) with acceptable code quality (5-10 times faster than interpretation). Additionally,

the integration in hard- and soft-core CPUs is simple, as only memory access is required.

Potential drawbacks of this concept exist. Most important is requiring an FPGA large enough to contain at least one translation phase. Besides costs, power consumption may be an issue. But if an FPGA with spare resources already exists in the system, JIFFY becomes a viable alternative to CPU upgrade (which is not always possible).

Obviously, the implementation of a JIT in an FPGA needs detailed examination of the translation steps and their impact on FPGA logic resources, translation speed, and code quality. Almost all decisions can be varied with respect to trade-off considerations. The intermediate code described in the following section is one example of this consideration: it is tailored to JVM translation, but can easily be modified for other applications.

3. THE INTERMEDIATE VIRTUAL MACHINE

The IJVM is responsible for an efficient representation of the originating JVM program code. Thus, IJVM inherits most of the JVM properties, like basic instructions and JVM data types. Actual implementations of more abstract JVM-instructions such as local variable/array access, or object-oriented processing are still hidden.

In addition, IJVM is the link to the target CPU's native code and must contain CPU-independent representation of runtime functions or other low level constructs. IJVM can also be used to assemble synthetic functions like data-type dependent stubs for library calls. With respect to these prerequisites, the instruction set architecture for the IJVM comprises three registers `r1-r3`, separated for integer and float, the usual data types and typed data transfer operations. The ALU-operations have a 2-operand format. There are specialized commands for memory access and local variables. Also, traps, prologue/epilogue-, and stub-builder-macros for the runtime system are included.

The IJVM opcode format is very simple. Instruction code, data type, registers, and a 32-bit constant are encoded in 64 bits. As the translated IJVM-opcodes are no longer needed after method translation, memory consumption is not a limiting issue here.

During native code generation, each IJVM instruction translates into one or more native opcodes. As the IJVM codes carry individual types, registers and constants, this translation cannot be a simple table lookup. Furthermore, inclusion of specific compile-time arguments (handler addresses, etc.) is needed. Some pre-calculations are necessary for good code performance: the simpler variant is scaling of constants (inherited from the JVM opcode) to access various tables. For this purpose, multiplication with 4 or 8 is usually sufficient. Access to local variables at the stack requires a more complex calculation, taking the number of

method arguments into account.

In summary, registers, constants, and calculated offsets need to be inserted into the right fields of the native opcode. For registers, this insertion performs a translation into different register parts depending on the data type. For example, register `r1` with 8/16/32/64 bits is converted to the AL/AX/EAX/EDX:EAX-variants on the IA32 architecture.

RISC processors usually have a very regular opcode format, but CISC needs a greater flexibility in register insertion. As a prominent worst case, in IA32 at least 16 different positions and register translations and 15 constant calculations were identified for the various addressing modes and data types. For a 64-bit RISC CPU like the Alpha processor, only 3 different register modes and 4 calculation modes exist.

This overview shows the complexity of developing a translation process even for only one architecture. To avoid multiple implementations, the FPGA logic should be independent of the target CPU. Hence, a universal description for the CPU and its opcode format has to be developed, moving changes in the target architecture from hardware into some form of software.

4. TABLE STRUCTURE, PATCH PROCESS AND IMPLEMENTATION

The basic data structures describing a CPU target and the translation of IJVM opcodes are shown in Fig. 1. The IJVM opcode and its data type form an address into a reference table. The addressed element points into a larger table, containing the formatted opcode templates of the target CPU. As the native opcodes are only templates and include hints for parameterization to form a "real" opcode, these templates were called **Paracode**.

Each IJVM opcode may contain one or more Paracodes, where one Paracode corresponds to one or more native opcodes. Each Paracode contains the opcode template (currently max. 12 byte), length of the template, three patch descriptions, and an offset for applying the patches. Number of target opcodes in the template is limited by both, maximum template length and the up to 3 patches that can be applied. Those patches can work on the whole template space.

Each patch description consists of the patch type, an offset, and a register operand description, specifying which register operand in the IJVM opcode should be translated. The additional offset enables patches to work on different opcodes in one template, or the insertion of constants at different positions within one opcode (needed for CISC).

The patch type references a table containing specific patch information. This includes the byte position where a patch starts, bit position within this byte, number of bits to insert, the type (register or constant), and some flags for endianness and linker operations. If a register needs to be patched, data type and the number of the IJVM register ref-

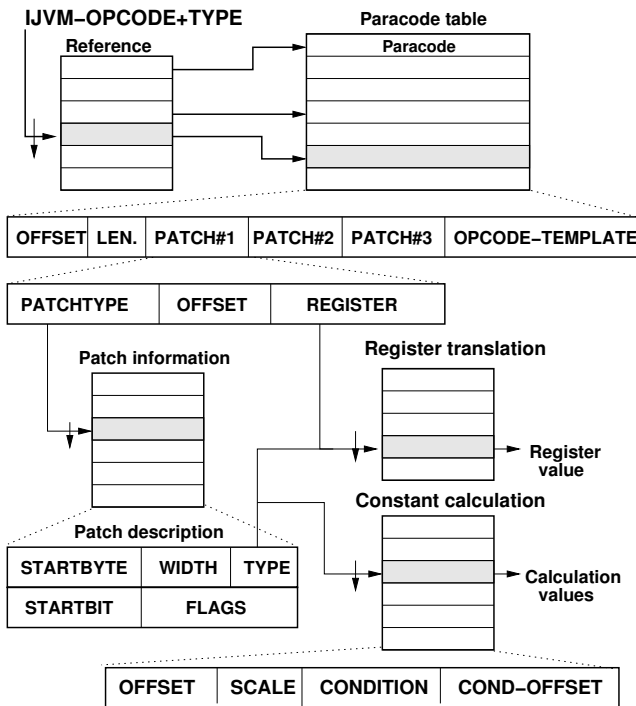


Fig. 1. Structure of Patch Descriptions

erence another lookup table, resulting in the effective bit pattern of this register for the target CPU.

If a constant is inserted, the type determines how to calculate this constant with offset (displacement), scaling, and conditional offset for local variable access. The constant source can be the constant from the IJVM instruction or one of the argument constants determined prior to compilation.

During development of the Paracode for IA32 and Alpha, the maximum number of three patches for one native opcode was not experienced as a restriction. Although the IA32 opcode format would allow more insertions, the actual IJVM translations did not need more than three insertions in one opcode for good code quality.

Fig. 2 demonstrates in detail how the translation works: the IJVM instruction to be translated (`movlvr_l 6, r2`) is a 64-bit read access of a local variable. The IJVM opcode specifies the actual number of the variable (6) and the (virtual) 64-bit register `r2` as the first register operand.

The IJVM opcode references a sequence of two Paracode entries, each describing an instruction with a base pointer relative move into a 32-bit register. The first template “`movl %v1(%ebp),%r1.l`” encodes in `%v1` the relative offset of the lower 32 bits on the stack. In the corresponding Paracode, the opcode template (8b 85 and a DWORD constant) is enriched with the two required patches for the `%v1` and `%r1.l` placeholders. The first patch describes a register translation from the lower 32 bits of a virtual register into the ModRM-byte, the second one a constant calculation.

When inserting the values from the IJVM opcode field (local variable 6, virtual register 2), both translations result in bit patterns which are inserted into the specified locations of the opcode template. After patch insertion, the next paracode entry can be processed.

If a patch description indicates that the constant is actually a relative address (e.g. for relative jumps), the position is stored in memory. Due to the way the IJVM opcodes are generated from the JVM instructions, a special IJVM opcode specifying jump targets exists. This “label”-opcode stores the actual program counter. Thus, after one FPGA assembler pass all relevant information is available for the concluding linker phase (see [5] for details).

In the implemented patch unit the opcode template is stored in a buffer of 12 bytes. As CISC patches may be inserted at any byte offset, a 4 byte window can be extracted with a multiplexer and fed into the patch block. The patch block then performs one patch insertion with the given data. Its output is then written back into the buffer. After the last patch is processed, the opcode is stored in a FIFO. Since CISC opcodes may be single bytes, the FIFO also combines data into consecutive 32-bit words.

5. RESULTS

The described translation concept was first simulated in software and IA32 was chosen as a typical CISC architecture with a complex opcode format and numerous addressing modes. The Alpha-21164 architecture was chosen as a typical RISC architecture. For both a pre-assembler was written to generate the Paracode tables. It reads sequences of native instructions for one IJVM opcode implementation, looks for indicated parameters, and generates the opcode template along with the patch description. A total of 180 IJVM opcodes was implemented, most of them resulting in 1 to 5 native instructions, some have up to 20 instructions.

The size of the Paracode table is about 24kByte for both architectures, including patch type description and reference table. Since Paracode entries are aligned on 32 byte boundaries for easy access, the table is only sparsely packed and contains mostly zeros. It can be easily packed to about 5-6 kByte, with gzip compression to about 2.5 kByte.

Method	sieve	fibonacci	sine	string-length	matrix multipl.
IJVM opcodes	93	20	34	36	140
IA32 opcodes (bytes)	374	118	222	203	542
Clock cycles	1174	303	539	516	1768
Cycles/bytes	3.1	2.6	2.4	2.5	3.2

Table 1. FPGA Clocks for IA32 Code Generation

The software implementation was instrumented to estimate the necessary clock cycles. With reasonable assump-

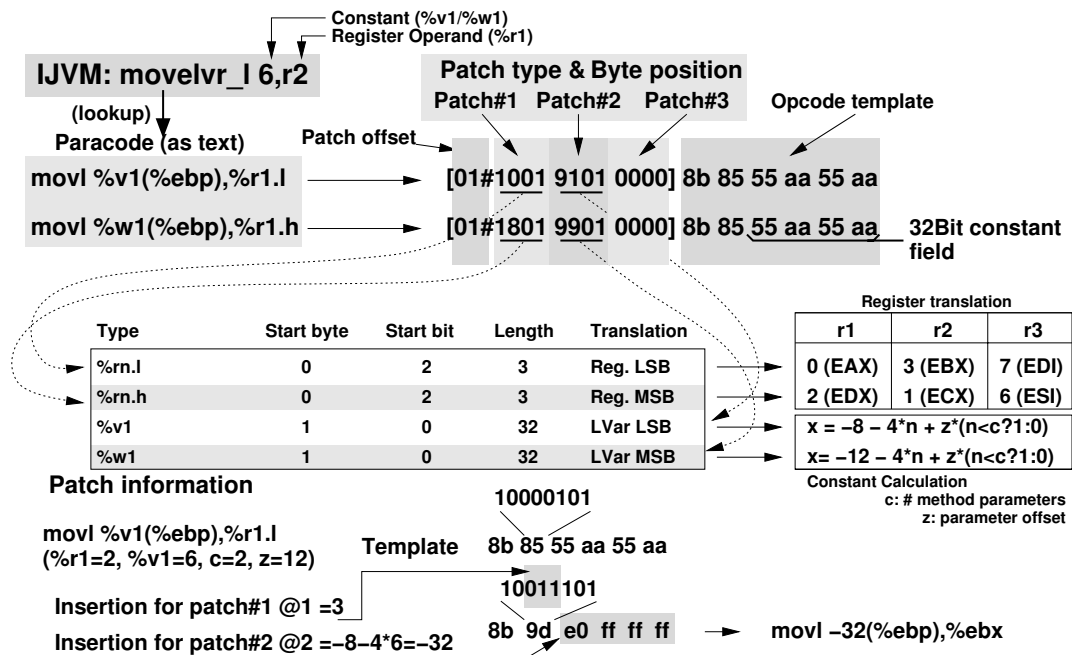


Fig. 2. Example of Patch Process

tions for processing work and memory accesses, translations from JVM to native code were performed for small methods. Number of IJVM opcodes, byte count for the resulting native IA32 code and needed clock cycles are shown in Table 1. Values for Alpha code are about 20% to 50% higher as more native instructions are needed. The cycles/bytes-relations reveal quite fast code generation: at just 50MHz about 10 to 20 MBytes/s of native code can be produced.

Finally, a VHDL implementation of the FPGA assembler was developed and evaluated. For Xilinx Spartan 2E and 3, synthesis with XST 6.3 and routing resulted in approx. 800 slices containing about 450 flip-flops. This includes internal distributed RAM containing register translation values, calculation instructions, and translation constants. Hence, the “full-featured” FPGA assembler needs only about 22% of the logic resources of a 3S400. A stripped-down version for RISC-only reduces this to about 500 slices (14%). The clock frequency after routing was between 55MHz (XC2S200E-5) and about 85MHz (XC3S400-4). Pipelining and a more detailed examination of the synthesized logic should allow more reduction in the used FPGA resources and higher frequency.

6. CONCLUSIONS AND OUTLOOK

The FPGA assembler has shown to be a very efficient and elegant way of generating code for various CPUs. The extended table lookup concept allows easy development and updates while preserving sufficient flexibility to target CISC

and RISC architectures. The possibility of inserting calculated or predefined constants and the transfer of registers from the intermediate language results in the generated code looking almost hand-coded.

The actual implementation of the JIFFY concept with all phases included is suitable for FPGAs like the Xilinx Spartan 2S200. An adaptation for the PowerPC integrated in Virtex2Pro or Virtex4 seems to be a very interesting target.

7. REFERENCES

- [1] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
- [2] Sun Inc. picojava microprocessor cores. [Online]. Available: <http://www.sun.com/microelectronics/picoJava>
- [3] M. Schöberl. JOP - Java Optimized Processor. [Online]. Available: <http://www.jopdesign.com>
- [4] ARM Inc. (2001) Arm Jazelle Technology. [Online]. Available: <http://www.arm.com/products/solutions/Jazelle.html>
- [5] G. Acher, “JIFFY - Ein FPGA-basierter Java Just-in-Time Compiler für eingebettete Anwendungen,” Ph.D. dissertation, Technische Universität München, 2003. [Online]. Available: <http://tumblr.biblio.tu-muenchen.de/publ/diss/in/2003/acher.html>
- [6] Sun Inc., “The Java HotSpot Virtual Machine – Technical White Paper,” 2001.