A profiler and compiler for the Wonka Virtual Machine

Dries Buytaert
dries.buytaert@acunia.com
Frans Arickx
Johan Vos

DEVELOPED at ACUNIA [1], Wonka™ is an open source[2] Virtual Machine (VM) for executing Java™ bytecode methods.

Wonka is a self-contained, clean-room VM implementation designed for use in embedded systems and is targeted towards Java 2 compliance: it has no dependence on external libraries (except for `libc`) and comes with all the essential standard class libraries (Foundation Profile/CDC), including an Abstract Windowing Toolkit (AWT), named Rudolph™ [1].

Rudolph is suitable for running Wonka directly on a memory-mapped or framebuffer device, supports all required graphics operations, has a built-in font rendering engine, supports GIF and PNG images, provides input/output device handling, and more.

For maximum portability, Wonka is accompanied by its own real-time operation system (RTOS), called OSwald™. Thanks to OSwald, Wonka does not require a host operating system and can run on bare bones hardware.

As such, Wonka provides a complete, portable open source solution for embedded systems.

Currently Wonka runs in interpreter mode only which imposes a performance penalty due to the runtime overhead of fetching, decoding and executing bytecode instructions. For performance's sake, ACUNIA is developing a runtime compiler to translate Java bytecode to native machine code during program execution.

I. Profiling and critical path detection

To do *selective compilation*, methods that represent a highly favorable opportunity for runtime compilation and optimization need to be identified. Such methods are typically part of a *critical path*. Selecting methods for compilation is done in two steps.

First, each method's execution frequency is tracked using an invocation counter. If this counter exceeds a preset threshold, the method is denoted a *critical method*, and may be part of a critical path. The invocation counters are periodically reset to ensure that only frequently executing methods are marked as critical.

The second step must detect whether the execution is in a critical path. The algorithm is inspired by work of M. Merten [2] but is software-driven. An additional per-thread saturating counter is used to measure the dynamic execution percentage of critical methods. Initialized at its maximum value, the counter is decremented by D each time a critical method is invoked and incremented by I each time a non-critical method is invoked.

If the counter reaches zero, a critical path is identified and the compiler is activated to compile the critical methods. As such, the compiler compiles those methods that have been called a minimum number of times (*what to compile*) but only when they account for a large execution percentage or when they have run for a longer period of time (*when to compile*).

The simple counter-based nature of the algorithm imposes minimal runtime overhead and the profiling framework seems both fast and accurate. Preliminary benchmarks using `SPEC JVM98` show performance improvements up to 15% and performance never being degraded compared to unconditional compilation.

Our profiling framework can be easily extended to include other heuristics or metrics. One interesting possibility offered by OSwald is to collect accurate per-thread timing information independent of context-switches, and hence to identify methods with particularly time-consuming loops. We plan to experiment with this and with other metrics such as bytecode size, call stack depth, and so on.

II. Compiling and optimizing methods

Once a critical path has been identified, the compilation and optimization phase can start. To make optimizations independent of both front- and backend, analysis and optimization phases are performed on a method's *intermediate representation* (IR). Methods are translated to their IR through *symbolic execution* using a simulated operand stack.

At compile-time, each method's IR can be exported and converted to a PostScript file which allows for quick debugging as the effect of optimizations can be visually inspected. Figure 1 shows the

D. Buytaert is PhD student at the University of Antwerp and software engineer at ACUNIA.

Prof. Dr. F. Arickx is a faculty member of the Department of Mathematics and Computer Science at the University of Antwerp, Belgium. E-mail: `frans.arickx@ruca.ua.ac.be`.

Dr .Ir. J. Vos is Director New Technology at ACUNIA, Belgium. E-mail: `johan.vos@acunia.com`.

[1]ACUNIA is a provider of software and hardware solutions for the next generation telematics services.

[2]Wonka is available under a BSD-style license and can be downloaded from `http://wonka.acunia.com/`.

IR of `java.lang.Math.max(II)` as emitted by the compiler itself.

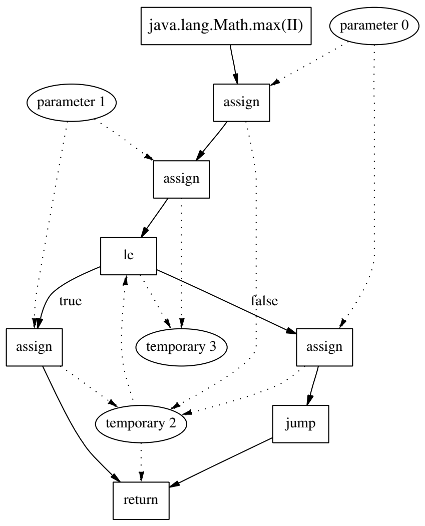

Fig. 1. The IR of `java.lang.Math.max(II)`. The solid edges represent the control flow and the dotted edges represent the data dependencies. The boxes denote IR instructions and the ellipses denote literals.

After the translation and optimization phase, a platform-specific backend is responsible for instruction selection and register allocation. Instruction selection is done using a BURS tree pattern matcher emitted by a *code generator generator* that selects the lowest-cost implementation of a given IR-construct [3]. The design of the code generator generator is based on `iburg` but uses a slightly different machine description language that makes it possible to attach *dynamic costs* and *actions* to rules in order to facilitate code emission. Register allocation is performed locally, at the basic block level, using a *linear scan register allocator* [4].

The final phase is the translation of the IR to binary machine code. A built-in assembler makes a single pass through the IR, emits binary machine code, performs basic peephole optimizations, patches unresolved jumps and links the compiled method into the running VM.

III. Interacting with a concurrent garbage collector

Even though Wonka supports mixed-mode execution of native and interpreted code, native code generated by the runtime compiler does not use the Java operand stack like interpreted methods do. As a direct result, object references used in native code have to be made reachable for the garbage collector (GC) to prevent them from being garbage collected.

We decided against generating native code that explicitly registers and deregisters object references through function calls. It would introduce a large amount of method calls which would complicate a method's IR, increase compilation time, and drastically sacrifice runtime performance.

Because of its concurrent GC, Wonka doesn't use so-called *garbage collection points*. As such, our compiler can't export *stack maps*, a solution that is being adopted by some of today's runtime compilers [5]. Because we can't determine what the layout of the stack is at garbage collection time and because the stack may change during garbage collection, we have adjusted the GC to make it conservatively scan registers and stack frames of compiled methods for pointers to the Java heap. As native stack frames get pushed and popped during program execution, object references are implicitly registered and deregistered with minimal runtime overhead. Using this simple mechanism, garbage collection can be done at any time, using a concurrent GC.

IV. Conclusions and discussion

This paper presents our profiling framework and the manner in which methods are selected for compilation. We provide insight into the overall design of our runtime compiler that we implemented as part of Wonka, and describe the simple *native call stack scanning* algorithm used to make our compiler and concurrent GC coexist.

While we already generate working machine code that significantly improves Wonka's performance, we have yet to explore most of the existing optimization algorithms as well as prototype new technologies. We are confident that the presented compiler architecture will lend itself to this as we have already begun to take advantage of it.

Our current research work involves extending the ideas presented in this paper. The outcome of this work will be a compiler meeting the restrictions and the real-time characteristics of embedded systems, and a tool to investigate and prototype selective compilation heuristics.

References

[1] D. Buytaert, *Java AWT voor embedded systemen*, Licenciaatsthesis, Department of Computer Science, University of Antwerp, 2000.

[2] M. C. Merten and A. R. Trick and C. N. George and J. C. Gyllenhaal and W. W. Hwu, *A hardware-driven profiling scheme for identifying program hot-spots to support runtime optimization*, ISCA, 136-147, 1999.

[3] C. W. Fraser and D. R. Hanson and T. A. Proebsting, *Engineering a simple, efficient code-generator generator*, ACM letters on programming languages and systems, 1(3):213-226, 1992.

[4] M. Poletto and V. Sarkar, *Linear scan register allocation*, ACM transactions on programming languages and systems, 21(5):895-913, 1999.

[5] M G. Burke and J. Choi and S. J. Fink and D. Grove and M. Hind and V. Sarkar and M. J. Serrano and V. C. Sreedhar and H. Srinivasan and J. Whaley, *The Jalapeno dynamic optimizing compiler for Java*, Java Grande, 129-141, 1999.