Java will be faster than C++

Kirk Reinholtz
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, California 91109-8099
Mailstop 303-310
kirk.reinholtz@acm.org

Abstract

This paper shows that it is possible for Java to have better general performance than C++, and that it is likely that this will actually occur. Java performance can exceed that of C++ because dynamic compilation gives the Java compiler access to runtime information not available to a C++ compiler. This will occur because the rapidly growing market for embedded systems will be driven to extend battery life. Since each CPU clock cycle consumes some power, battery life is extended by improving performance, thus achieving more computation per clock cycle.

Keywords

Java, C; C++; compiler performance; compilers; dynamic compilation; comparative performance of Java and C++.

Summary

There are numerous compelling reasons to use Java. It is relatively easy to learn and use, so training and debugging costs are reduced. It brings numerous best computing practices to the general programming community, in the form of its extensive and standardized class libraries. Finally, it enables new architectures by its ability to easily send code and data over a network.

Unfortunately, contemporary Java environments provide deficient performance relative to those of C++. The low performance of the tools is often taken to imply that Java itself is inherently less efficient than C++, which generally hinders its adoption. Trade studies of Java vs. C++ are often of the form "is the performance hit worth it?".

Java, by virtue of its ability to compile the program as it executes, can achieve performance greater than that of C++ because the compiler has access to information that just isn't available to a traditional C++ compiler. There is a lot of work between here and there, but it can be done.

Java performance will constantly improve because there will be serious money to be had by improving it past that possible with C++. The reason is that battery life is a serious concern and powerful market differentiator in the growing embedded market, and improved performance improves battery life because it basically means more bang per CPU cycle. Each CPU cycle drains the battery a little bit, so the less CPU cycles used for a given task, the longer the battery lasts.

There is a lot of work being done today to make Java suitable for embedded/real-time development. Once that work is codified, it will narrow the semantic gap between Java and the embedded/real-time world. Java will join the set of languages routinely used to develop embedded and real-time systems, and then performance will be the only thing that stands

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

between Java and its becoming the dominant language for such work.

There is a lot of work to be done to make this possibility a reality. It's worth it, because it will move us to the next level of compiler technologies, where it will be routine to use runtime information to evolve the optimization of a program as it executes. We'll be able to use the advantages of Java without penalty, and language research can begin to harmonize language features with this new compilation paradigm.

1 Introduction

The Java language can be either interpreted or compiled. Java fully exploits interpretation, which is one of the key distinctions between the Java language and other contemporary languages. It has various advantages insofar as code footprint and debugging is concerned, but the big win is it enables the "write once, run anywhere" vision: The Java language and class libraries are specified sufficiently tightly that you've got a pretty good chance of your program working on a target platform you didn't consider when writing your program, and Java bytecode technology lets virtually any platform grab and execute your program.

Unfortunately, interpreters, even really fancy ones, are relatively slow. Compilation is used to improve the performance of Java by translating it into machine code that is executed directly by the underlying hardware, rather than by an interpreter. Current Java compilers tend towards one of two flavors: One, known as "static compilation", compiles the source code to machine code on the developers workstation, much as typically done when using C++. The other, known as "dynamic compilation", compiles the Java bytecodes rather than Java source code, and takes place upon the target and as the program is executed. There are two key distinctions between these compilation models: (1) Static compilation has access to the source code and occurs before the program is executed; and (2) Dynamic compilation has access only to the bytecodes and takes place during the execution of the program.

The remainder of this paper will explore the consequences of allowing compilation during the execution of the program, and show that this could lead to Java having better performance than C++.

2 Brief history of compilation

There is a fundamental property of compilation: The more information the compiler has, the better the resulting code can be. One can view the history of compilation in this context as seeking to gain access to and utilize ever more such information in order to generate the fastest code possible.

The earliest compilers knew only the source code, target platform, and perhaps programmer intent as to optimizing for size or performance. A lot of research was performed to figure out how to generate good code given this information, and pretty good compilers were the result.

Researchers kept pushing to improve performance, but they hit a wall: There were a lot of possible translations of a program. Some of those translations are strictly better (i.e. faster or smaller) than others, but many others depend upon the particulars of a given execution of the program. There was no way to pick the proper translation without access to runtime information.

That led to the next phase of compiler development. Compilers were benchmarked against large bodies of existing code, and adjusted to maximize performance for that code. The working assumption is that "code is code". That is to say, if the compiler works well on a few dozen programs, it'll work great on everything. This is pretty much the state of the practice today for contemporary compilers for e.g. C++.

The next step in compiler evolution was to figure out a way to feed the execution specifics of the program being compiled back to the compiler, so it could generate even better code. Numerous schemes were implemented, but they generally created an instrumented version of the program, which wrote various statistics into a file during or after the program was executed. After the program was run through its typical execution profile a few times, the file was then given to the compiler. The compiler would use this information to tune its optimizers in many ways: it could focus on frequently-used routines, unroll heavily-used loops, put frequently-accessed variables into global registers, inline small and often-used functions,

eliminate unused code, improve branch prediction, improve cache and paging performance, optimize vtable dispatch, and so-on. There are contemporary compilers for most languages that can do this.

The first phase of compiler evolution was oblivious to the execution profile of the program it was compiling. The next phase was aware of an archetypical execution profile, which let it generate better code, and the phase following that could use an execution profile of the code being compiled. This was pretty good, but there are a couple of weaknesses. First, if the profile of a given run varies from the example given to the compiler, performance won't be as good as possible. Performance may even be degraded, depending upon the aggressiveness of the optimizations. Second, if the execution profile varies substantially during an execution, the compiler just can't pick a single translation that's best for the whole execution. For example, many programs have distinct phases, and often the phases will have different optimization needs as to global register allocation, inlining, branch prediction, and so-on.

3 Runtime compilation

Clearly, the next step in compiler evolution is to compile the program as it executes. This form of on-the-fly compilation is normally viewed as a way to mitigate the performance cost of bytecodes, and bytecodes are important because they enable the Java vision of "write once, run anywhere". The general hope within the Java community is that only a minimal performance reduction need be suffered in order to gain access to the advantages of Java. That's the mindset of today. However, we're really witnessing a major step in compiler evolution, if not a full-blown paradigm shift that will change the face of programming forever. The reason is this: Java performance is bound to exceed that of C++.

How can the performance of Java is bound to exceed that of C++? After all, compiler technology has something of a "speed of light" analog: The optimizations can only do so well, even given a zillion CPU cycles to improve them. And worse, most of the pay-off is in the first few billion cycles of computation. The compiler quickly reaches a point where a whole lot more "pushing" results in very little speedup. The law of diminishing returns rules with an iron hand in this world. Traditional C++ compilers are about as good as they're going to get. There's a little room for improvement, but not a lot[1].

One way past this performance barrier is to compile the program at run-time, which will give the compiler access to information not available to previous generations of compilers. The added information gives the compiler more opportunities for optimization, as well as the chance to evolve the optimizations during the execution of the program.

Java will have unprecedented performance because it's the vehicle by which this phase of compiler evolution will be mainstreamed. There is nothing specifically magic about the technical aspects of the Java language. There are numerous other languages with the prerequisites. The magic of Java is that it has brought fantastic research focus on pushing compilation technology to this next step. The technology in one form or another will probably be retrofitted to other languages, including C++, but Java can be the first.

4 Tradespace

One is generally working in a trade-space of one form of another: Once the cherry-picking is over, one doesn't get something for nothing and one should understand what trades against what. In this case, there are a couple of new performance issues to consider: compilation takes time, and it takes CPU cycles. The argument that runtime compilation will eventually yield better performance than earlier compiler technologies was based upon the heretofore tacit assumptions that (1) the information used to do better optimization is still valid when the compilation is finished and so provides some advantage; and (2) any target CPU cycles required to do the computation are amortized

---

[1] One can use the concepts in this paragraph to judge the level of compiler maturity within a domain. For example, as long there are claims of one implementation of Java running twice or ten times faster than another, you can know the compilers are still immature and the developers are "cherry picking". When comparisons start showing differences of perhaps ten percent or less, you can know they've started working on the hard stuff and are nearing the wall.

by the resulting performance improvements. The upshot of this is that some programs will run faster, some will run slower, and the degree of impact, good or bad, will depend upon the program and perhaps its inputs.

So, the performance gain or loss of an arbitrary program depends, in part, upon the actual source code being executed and its inputs as well. We know we'll never eliminate the potential for slow-down without reducing the potential for speed-up, because it takes CPU cycles to detect a slowdown. Basically, there is no guarantee that dynamic compilation will improve the performance of an arbitrary program. It will do so on the average, but we can't know it will in every case.

Fortunately, there is a subtle aspect to the above that we can exploit to improve the picture: It is true that theory says that not all programs will exhibit the same, or even any, performance improvements, but the developer doesn't care about all programs. The developer cares only about the program he's going to write, and there is nothing that says he can't write it such that it works in harmony with the compilers. It has always been true that the programmer needs to work with his tools if he wishes to maximize performance. Dynamic compilation is just one more such technology.

5 Battery Life

A battery is an energy storage device: it can emit a certain number of watts for a certain amount of time. It can emit half as many watts for about twice as long, or twice the watts for half as long. A CPU consumes a certain number of watts per MIP. Similarly to the battery, it takes about half the watts to run at half the MIPS, and twice as many watts to run at twice the MIPS. RAM behaves in a like manner, consuming a certain number of watts per accesses per second.

Taken together, this means that as the clock rate of the compute system is reduced, its battery life is proportionately improved. This isn't a big deal for devices that last months on a single battery charge, but for high-consumption devices where the battery life is measured in hours, even a slight improvement in battery life can make a big difference in the convenience and usability of the device.

Improved energy efficiency can also reduce the cost of a device, because it can provide tolerable battery life using inexpensive mainstream battery technologies, rather than requiring the use of expensive exotics.

So, improving the performance of a system can translate directly into improved battery life. The more efficiently the code uses each CPU cycle, the longer the battery will last.

6 Conclusion

On-the-fly compilation is a paradigm shift. Paradigm shifts generally lead to exciting and unanticipated new possibilities. In particular, Java may become faster than C++ because the compiler has access to information not available to a C++ compiler. Market forces will tend to make this possibility become an actuality because improved performance means improved energy efficiency and battery life. Battery life will be an important product discriminator in the rapidly growing market for embedded systems.