The Computer Architect Gets More Choices if the Programmer Gets a More Powerful Computer

Kay P. Litchfield
The Boeing Company

For much of computer programming, the programmer ponders the problem to solve, identifies a strategy for solving it using a mental model of a simple computer, records the steps in a language that is usually not quite as intricate, uses a compiler to convert the steps into the language of a real computer, and runs the resulting program to acquire the desired answer or action. A computer architect hoping to entice the programmer to buy his product seeks to design a computer that will run the program more quickly. Unless his efforts are supplemented by those of the compiler writer, the only improvement strategy is to simply speed up the execution of the original compiled program. A compiler writer may be able to find ways to discover sequences of program steps that can be converted into fewer steps using different instructions, and thereby give the computer architect the opportunity to speed up the program by other than brute force.

The programmer's mental model of the computer drastically limits the ease with which the compiler writer or computer architect can improve the efficiency of the resulting program. If the programmer's model cannot produce both a quotient and remainder in the same division instruction and both are needed, then two division instructions will be requested (and perhaps not consecutively, or not with the same copies of the divisor and dividend). If the programmer needs a simple permutation of bits, then several bit-wise masking and multiplication or division (by powers of 2) instructions may be used. If the programmer just needed to disallow the overflows that result from doing integer arithmetic modulo 2^32, there will be many extra steps to identify those situations and adapt. If the programmer missed one or two of these extra needed steps, the compiler will not be able to recognize that the desired result was simpler. These mental model limitations may originate from experience with traditional computers, but more importantly, they arise from the language being used by the programmer.

Expanding the programmer's model of the computer being used to include everything that any computer architect may ever want to include in new computers gives the architect valuable options. The compiler writer is assigned to bridge the gap, freeing the computer architect to design parts in the most practical order. The new model should be so powerful that it never again becomes the governing constraint—i.e. it is never less powerful than real computers. To be practical, there would be a strategy by which the programmer provides (separate from the main program) the portions of the compiler that the compiler writer has not had justification to write. (Obviously, the portions provided by the programmer may be incorporated in a different fashion than the portions provided by the compiler writer.)

The language used by the programmer needs to be expanded from current (third-generation) languages so that it is easy to ask for what is really wanted rather than translating it into how an already existing computer might do it. The compiler can automate the process of suggesting to the computer architect what will help the most in future designs.

A completely different benefit is what particularly interests me. If the programmer is able to write the program in the way that makes the program most clear and coherent (to people), it will be much easier for him to write it correctly and check it, and it will be much easier for others to check his work.