# Programming Language Support for Architecture (and Operating System) Innovation

Tito Autrey and Sally McKee

**Problem:** The languages we use for all current, significant benchmarks inhibit architectural innovation by constraining both the compiler's and the architecture's ability to make intelligent choices about how a computation is realized. C, C++, and Fortran are too concerned with details of *how* a particular computation is accomplished, instead of describing just *what* is to be computed.

Consequence of languages being too focused on the *how* of a program include the potential presence of memory aliases, overly constrained memory-layout of data structures, and fixed order of operations at the time that the program is written. Allowing the programmer to create aliases (multiple names that refer to the same memory location) inhibits two important optimizations: the compiler cannot safely keep an aliased variable in a register, since modification using one name might not update the register copy; and the compiler and hardware are more constrained when scheduling memory loads and stores, since, for instance, a read of a variable cannot be moved before a write to a potential alias for that same memory location. In general, the presence of aliases forces the compiler, the operating system, and the hardware to make conservative decisions about managing the memory hierarchy, the negative performance impacts of this consequence will grow with the processor/memory performance gap. C and C++ allow aliases by providing an address (pointer) type. Fortran allows aliases by using call-by-reference semantics, although it eliminates some of the optimization penalty it creates by allowing the compiler to optimize as if there were no aliases. In C, C++ and Fortran, arrays, structures, and unions (in whatever form they take) all have language-defined, memory-layout constraints. Generally, these constraints require that the layout of an entity correspond to the syntactic description of the type. Except when interacting with fixed interfaces external to the CPU (e.g., device registers), the compiler should be allowed to control the layout of aggregate data structures, so that the program might use the memory hierarchy most efficiently. Short-circuit predicate operators and floating point operators are defined to have strong ordering requirements that can limit opportunities for correctness-preserving optimizations.

**Proposed Solution:** The solution is to develop a language which is better suited to optimization. A higher-level language would indicate *what* computations need to take place, and leave the *how* of the computations (including memory layout, scheduling of computations, and scheduling of memory accesses) to an aggressive optimizing compiler or to intelligent hardware. This higher-level language should be free from misfeatures that allow the *programmer* to constrain the *how* of computations.

This is not a vote for a *One True Language* that could be used to solve *all* problems *very* well. Rather, we desire a language that supports general-purpose programming at least as well as Java, but that prevents the programmer from specifying a computation at the wrong level of detail. We envision a language that supports the creation of domain-specific languages, either as a compilation target or through the addition of libraries of primitives and optimization procedures to extend the compiler.

**Why this is a good approach?:** Computer architecture is the last element in a process for the realization of a computation. The first element in the chain is the program, a specification of what is to be computed, and in many cases how to compute it. The programming language itself affects the amount of *how* in the program. The second element is the compiler or program transformer. The third element is the hardware that performs the actual operations necessary to produce the solution. We believe that architects should intervene earlier in the process to ensure that the "input" to their part of the system is as favorable for them as possible. By examining languages that allow greater flexibility in the realization of a given computation, architects can free themselves to innovate.

Some people will be concerned about the viability of recoding applications. Certainly, there are companies that will balk at the notion, but there are many Fortune 500 companies already undertaking Y2K revisions, converting Autocoder to Cobol, translating RPG to Java and other bet-the-business recoding efforts. Applications need to be maintainable and extendible. Any significant application will be recoded during its lifetime. Why not make the next language favorable for compiler writers and computer architects?

The primary problem facing computer architecture is not a lack of ideas, but the benchmarks with which an idea is evaluated. Change the language with which benchmark solutions are described, and the goodness of innovative hardware for producing solutions can change.

In summary, "Its the benchmarks, stupid!"