How compilers and tools differ for embedded systems

Michael Wolfe
STMicroelectronics, Inc.

The study of compilers includes almost all of classical computer science: programming languages, formal languages, algorithms, data structures, instruction set design, computer architecture, implementation, everything almost down to logic design. Parts of the compiler include parsing, flow analysis, code improvements, parallelism detection at the instruction level as well as for multiple processors, resource allocation (registers, functional units), scheduling, and so on. Associated tools include assemblers, linkers, disassemblers, profilers, optimized libraries, and more. All of these have a long and distinguished history, actively developed over the past half century.

So, why is it so hard to create a satisfactory programming environment for today's embedded systems? Why and how must such an environment be different from those in use on general purpose computer systems?

I'm going to discuss this problem from the top down, starting at the marketplace for programming environments (read: compilers) for embedded systems versus the market in the general purpose workstation or cluster world. I work for STMicroelectronics' SST Portland Lab, which until the year 2000 was The Portland Group (PGI), an independent developer of very high performance C and Fortran compilers. Since 2000, we have developed highly optimizing compilers for embedded processors while continuing to grow The Portland Group business as part of our Lab operations. That business includes a large and growing customer base in the Intel and AMD x86 Linux market. That market includes multiple competing compiler vendors, which is an unusual situation compared to the traditional RISC/UNIX workstation and server market This situation developed largely because the primary CPU vendor, Intel, did not develop and market a compiler solution until the turn of the century. Now, of course, Intel has a highly visible compiler group, targeting the Pentium family, XScale embedded systems, and the Itanium.

At the SST Portland Lab, our continuing work in the general-purpose and high-performance computing markets in addition to work on compilers for ST embedded systems, in particular for the ST100 family of DSPs, gives us an interesting perspective. We're not alone in this regard. Some independent compiler companies can say the same, as well as, of course, Intel.

So, we have this healthy business selling compilers for the high performance Linux applications business. Who are our customers? Let me generalize and break our customers into two categories. In one category is a set of users who develop programs for internal use. These include laboratories and corporations developing critical modeling applications, such as an oil company developing seismic signal processing codes, or an aircraft engine manufacturer developing heat analysis, stress analysis, and fluid flow codes. These developers are in a rapid compile–test–run cycle. With these customers, the number of end-user applications per compiler sale is relatively low, or, conversely, the number of compiler sales per end-user application is relatively high.

In another category are independent software vendors, or ISVs. These are developers of large commercial applications, like fluid flow or crash test applications that are purchased and used by large corporations or laboratories in binary form. Here, to the compiler vendor, the direct value of the sale is relatively low; there may be many end users of the application for each compiler sale. The importance of these customers comes from the marketing value of a large ISV using our software, and also because sales of these applications to a user will often indirectly result in additional compiler sales, which would then fall into the first group.

The point here is that we have two types of external customers. The first type results in large numbers of compiler users, who are constantly beating on the compiler with a wide variety of programs, resulting in lots of feedback to the compiler developers (us) about problems with correctness, performance, and compatibility. The second type results in very widespread use of executables generated by our compilers, which results in extreme stress on those executables from a quality standpoint and which in addition often generates even more widespread use of our compilers.

This is in great contrast to the embedded systems marketplace. In the embedded systems market, there are very few (if any) compiler users who are developing applications just for themselves. Take as a hypothetical situation, suppose ST were to market a processor core for inclusion into a cell phone. You might think ST would look forward to the compiler sales that come with a design win, but you'd be wrong. The compiler and software development tools are often thrown in with the design win. This changes the financial impact of compiler development; instead of being a profit center, it's a cost center. Money spent on compilers is money not spent on architectural improvements, manufacturing process, marketing, and more.

Look at how the design win was made. Very similar to the way high performance system sales were made in the 1980s, the potential customer comes up with a benchmark program or programs that exemplify the types of computation for which the embedded system will be used. Here, the customer can be very precise, since they already have the application. A team of specialists inside the semiconductor company then use whatever tools are available to tune the program for the platform, while their competitors will have another team doing the same for their platform.

Once a design win is secured, how many applications (read: compiler sales and users) will this entail? Answer: few. In fact, since the design win probably depended on successful porting of the application (singular) to the platform, there's very little post-win application development, though there is some amount of tuning, as the final product design becomes more concrete.

In all, this results in little desire on the part of product managers to invest in compiler and tools development. They can invest in an applications engineer who will help port and tune the application and get this design win, or in a compiler engineer who will tune the compiler with improvements that will probably come in too late for this design win and may or may not help the next one. Even relatively simple and standard product features that appear in any successful workstation compiler may be missing in these products. As one example, we acquired the software development kit from one of our competitors. Our experience was that the kit was hard to buy, took a long time to get delivered, was hard to install, and once it was finally installed, it was hard to use. This kind of experience would never happen in a competitive environment where the vendor was forced to compete for the business and loyalty of their users.

So now let's look at the programming environment. In particular, let's look at the goals when developing an application. In our Portland Group business, we have many customers developing large applications, either for their own internal use or as an ISV. We have worked hard to convince ISVs to use our compilers, and we continue to work hard to keep their business. So, what is their criterion for choosing a compiler suite? Correctness. Speed. Portability. And the greatest of these is speed. An ISV with a complex code can work around correctness, turn off the optimizer in one or two files, and usually they have to do that for any of the compilers they use. They can work around portability as well; by portability I mean operating systems and hardware platforms supported. But they can't work around speed. So how do we keep this important customer? Good customer service, new features, and most importantly, increasingly better performance on successive generations of processors and successive releases of our compilers. There are three or four really good, high quality compilers for x86 target systems, so we really have to stay on the ball all the time to keep this business.

Compare this to the embedded world. Suppose I have an application, say a CODEC for a cell phone, a typical performance-oriented application, and I need to build this for the next generation phone product. Suppose I come up with a nifty compiler optimization that will give me a ten percent performance boost for this application. All is good, right? No, not so fast. Memory footprint is another piece of the cost. If the performance boost comes from repeated function inlining, or loop unrolling, these make the program bigger. This may require a larger program memory, which will increase the cost of our piece of the final product, and may cause us to lose the design win. OK, suppose that doesn't happen, now it's running 10% faster. So what? It's not like the customer of the product can talk 10% faster on that cell phone; once we've hit a pre-specified performance target, better performance isn't that important.

On the other hand, if we speed up one part of the application, we can perhaps slow down another part. Slowing down that part may allow it to become substantially smaller, again the speed versus size tradeoff, and reduce the overall memory footprint. Alternatively, we can perhaps slow the clock, reduce the voltage, save power and increase battery lifetime.

In fact, the speed versus size tradeoff is a critical part of the application build process. Much of the work by an embedded systems programmer is taken up tuning compiler optimization switches for the best mix of performance and compact code size. Using a performance profiler and knowledge of the application, the programmer chooses the time-critical parts to optimize for speed, and chooses the rest to optimize for size. If we believe the 90-10 rule, that 90% of the time is spent in 10% of the code, then optimizing that 10% of the code for speed and the rest for size should give the best of both worlds. This doesn't reflect reality, however. There are other reasons to perhaps favor a slightly slower, smaller program, or to favor a slightly larger, faster program, as we've said. The desired goals are too subjective. The balance between speed and size takes more intelligence than we can automate.

Even more interesting are the cases where the instruction set is itself a variable. This may range from a custom coprocessor, which looks more or less like floating point functional units used to, to custom datapath extensions more closely integrated in the core itself, to reprogrammable or reconfigurable pipelines, which I think is a really cool technology. In these situations, we have even more variables to consider in addition to the size and speed of the program: The silicon footprint of the core + associated logic; extra power required for the extra silicon; generation of the system software (compiler, OS) that support the part. Some vendors and much current research is addressing this area. Remember to add the cost of maintenance of these features, plus customization or optimization of the application to use the new instructions; hopefully this is mostly automatic.

Finally, I want to talk about some compiler techniques themselves. We find many of the optimizations developed for various high performance and supercomputers 20-30 years ago are now being rediscovered, honed and redeployed in the general purpose market. Take vectorization, which was aimed at the bleeding edge; it's now used on your laptops. The same technique may be used for packed or stream operations on a cell phone or MP3 player. Parallel processing has now reached the mainstream, with multicore general purpose processors. In the embedded market, it's not performance but power that may drive the multicore product; if two slow cores solve the problem in the same time with a slower clock, they can use less power due to the quadratic voltage to power curve.

But let me focus on two particular features on compilers, both in relatively common use today, but likely to become more important. The first is whole program optimization, also called interprocedural optimization; I'll call this IPA, interprocedural analysis. The intent of having procedures is many-fold: modularity, simplification, code reuse, separate compilation, etc. However, to create a whole program, some tool has to put all this together. For standard imperative programs, this tool is the linker. The linker's job is relatively simple: collect all the objects, look for defined and undefined symbols in each object, resolve undefined symbols from other objects or from libraries, relocate objects and fill in undefined symbols with final values. But the linker typically doesn't inspect the program, in fact it typically doesn't even know whether the `.text` sections are program or in-line data. But here's the point at which real global information becomes available.

I claim that significant improvements can be made by more aggressive use of IPA. I say this for two reasons: one, just as the hardware guys are approaching diminishing returns on how much faster they can push the clock rate, compiler analysis and optimization is approaching diminishing returns on how much better we can make programs with only local analysis. The second is the scope of the optimizations that become available with IPA.

Let's look at what IPA can do, even in just C programs. The simplest optimization is constant argument propagation; there can be several benefits. The argument may be used in a conditional, allowing the conditional to be evaluated at compile time. If the argument is used as a loop limit, the constant value can be used to determine how the loop should be optimized. This also promotes modularity and code reuse, by allowing one version of the function to serve both as the general purpose code, and as the specialized version in particular performance critical applications. Related analysis can propagate the value of initialized, unmodified globals to their uses.

```
int debug = 0;
    ...
    foo( a, 10 );
    ...
    foo( b, 10 );
    ...
    void foo( float* x, int n ){ ...
        for( i = 0; i < n; ++i ) x[i] ...
        if( debug ) ....
    }
```

Similar analysis propagates the values of pointers; do the values of `x` and `y` interfere in the routine `saxpy`? This is the critical question for optimization of that routine. ANSI C now allows the `restrict` keyword, declaring that they must not conflict. IPA can frequently automatically determine when this information applies. In the general purpose research community, pointer target analysis or pointer alias analysis is largely taken up with determining the shape and characteristics of the dynamic data structure that is implemented with pointers, e.g., a tree, linked list, ring, etc. In the embedded world, simpler mechanisms can be very successful, since dynamic memory allocation is less prevalent.

```
    float z, v[100], w[200];
    saxpy( 100, z, v, w );
    ...
    void saxpy( int n, float a,
                float* x, float *y ){
        int i;
        for( i = 0; i < n; ++i ) y[i] += a*x[i];
    }
```

Other potentially important interprocedural analyses are controlling function inlining, propagating argument memory alignments, determining whether global variables are modified by a call, finding pure functions, and so on. Higher level languages, like C++, show a raft of additional possibilities exposed by IPA.

The second characteristic I want to discuss is performance feedback. Here, I don't mean profiling tools, though they are important and should be part of a complete package. I mean feedback from the compiler about what parts of the program are likely to run well, and what parts are likely to run poorly. More than that, the feedback should include information about what features or characteristics of the architecture or compiler prevent this part of the program from running well.

Let me illustrate this by an example; I apologize to those who may have heard me give this lesson before. Back in 1978, I attended a workshop at the Los Alamos Scientific Laboratory, as it was called then, on scientific parallel and vector computing. The Cray 1 had been installed there for two years, and there were many Cray users, along with users and vendors of other supercomputers. Most of the Cray users were migrating their codes from Control Data 6600 and 7600 machines. The big initial selling point for the Cray 1 was that it was twice as fast as the previous fastest machine at the time; it had an 80 MHz clock (12.5 ns), whereas the 7600 was running at 40 MHz. That the Cray had a vector mode was a bonus on top of that, but a pretty big bonus. The common lament, repeated often, was that the users had to spend hours rewriting all their programs because the unmentionable Cray Fortran compiler wouldn't vectorize their inner loops. The crying and whining was pretty intense, and these were bleeding edge programmers, who were really interested in getting the last drop of performance out of the machine. I took this as a charter to continue research on automatic vectorization and parallelization; in fact, some of us started a little software company to sell tools based on our research.

Jump ahead just eight years: I was at another workshop, sponsored by SIAM, the Society for Industrial and Applied Mathematics. There, we were talking about how great our vectorization software was, but the response we got, from a number of Cray users, was the Cray Fortran compiler was pretty good, it vectorized all their codes, they got all the performance they wanted, and never had to resort to CAL, the Cray Assembly Language. So what happened? Did the Cray compiler really get an order of magnitude better in just eight years?

In fact, the compiler had improved quite a bit, but that wasn't really what satisfied all their customers. There was another effect, caused indirectly by the compiler. This was in the days of batch computing, so the compiler would produce a program listing, which included some very important information; for each loop, it would tell the user whether the compiler was able to generate vector code or not. More than that, if not, the compiler would give information about what part of the loop prevented vectorization. This loop, this line, this array reference, this variable in this array reference at this line in this loop prevents me (the compiler) from vectorizing the loop; change this line, and maybe I can do a better job.

The difference between vector and scalar performance on Cray systems was so dramatic that users paid very close attention to the compiler feedback. This is an interesting interaction; it's almost Pavlovian: bad programmer, you wrote a nonvector loop. Good programmer, this loop vectorizes. And the result was two-fold. By 1986, all those old programs had been rewritten, and the inner loops all vectorized. The second, and more important, effect was all those old programmers had been trained by the compiler to respect the limits of the compiler. All inner loops must run down the stride-1 dimension, use simple subscripts, subroutine and function calls must be removed, and so on.

How is this important today? Again, let me illustrate by anecdote. In our Portland Group business, one of the goals is good benchmark performance, and one of the important benchmarks is SPEC. Below I show one of the critical loops in the ART benchmark. The performance of the benchmark depends on the performance of this and a couple of similar loops, and the performance of this loop depends entirely on the performance of the cache and memory. But there are several problems here. The struct `f1_layer` has 8-members, each 8 bytes long. This means each element of `f1_layer` takes up a whole 64-byte cache line. So while we're traversing the array `f1_layer`, we only use one value out of each cache line. Even worse, the two-dimensional `bus` array is dynamically allocated, and we're traversing down the columns; because `bus` is dynamically allocated, the compiler can't assume anything about its addressing. A few more comments: it turns out that the inner loop has a length of about 10,000, while the outer loop has only 6 iterations, and the conditional around the inner loop never trips, at least not for the benchmark datasets.

```
struct {
    double *I; double W, X, V, U, P, Q, R; }
    *f1_layer;
double **bus;
    for( tj = 0; tj < numf2s; tj++ ){
      Y[tj].y = 0;
      if( !Y[tj].reset )
        for( ti = 0; ti < numf1s; ti++ )
          Y[tj].y += f1_layer[ti].P * bus[ti][tj];
    }
```

I've seen at least three ways to optimize the program to make this loop run faster. Some compilers, automatically mind you, do what is sometimes called the ART hack. They reorganize the *data* for the struct `f1_layer`. The array-of-struct is restructured to become a struct of arrays. It takes up the same amount of space, but now each element `f1_layer[ti].P` becomes `f1_layer.P[ti]`, so becomes adjacent in the cache line; this in itself gives a significant performance boost in this loop and in other places in the program.

The second trick employed by some compilers is to interchange the two loops; the key here is the `bus` accesses are non-adjacent, have no cache locality, and require a row pointer load followed by a data load; so let's switch the two loops. There are two problems here; one is that the inner loop is very long, but after interchanging, the new inner loop is quite short. Also, interchanging takes the conditional that was outside that loop and brings it inside both loops. This is quite dangerous, yet at least one compiler implements this, if only because of the (somewhat artificial) importance of this benchmark. This gives over a factor of two improvement in the whole program, on one particular cache-dependent general purpose architecture, just from implementing this optimization.

The third trick, not implemented by any compiler that I know of (yet), is to leave the loops the way they are, and to transpose the array `bus`. It turns out `bus` is all dynamically allocated, in the normal C manner, but it is generally accessed by the wrong dimension. If we transpose the dimensions, the performance of this loop really screams. In fact, the performance drops by about a factor of 8 (not a misstatement) for this cache-dependent architecture, by restructuring the data.

The point I'm trying to make is not that compilers need to start looking at questionable optimizations, or data reorganization, though artificial benchmarks may drive that development anyway. The key is that in a real application with real performance sensitive users, if the user gets reliable feedback that there is a performance problem in this loop, or this routine, here are the parameters, here are the performance bottlenecks, here is a list of possible things to check out for correcting it, that user is going to be (a) motivated to focus on these bottlenecks, (b) learn how to work around such problems, and (c) trained how to avoid such bottlenecks in the future. The end result is better tuned applications for all, and more effective use of programmers talent.

Summary

The economics and marketplace realities of the embedded world make it difficult to convince program managers to invest sufficiently in compilers and tools for their systems. The software that exists is often a step or several steps below the quality and utility that we expect and receive in the general purpose workstation business. It takes a visionary to correct this situation. Economics will not make the argument.

Most compiler optimizations key on reducing the running time of the program. Two other important metrics in the embedded world are the program size and the power used. There is a careful balance among these three, and they have not nearly been fully explored. Some compilers optimize for size by simply disabling certain size-detrimental optimizations, like loop unrolling or function inlining. I claim that this is insufficient, and more needs to be done here. It's interesting that texts written in the 1970s, when I took compiler classes, at least gave a nod to the idea of optimizing for size, though most modern texts don't even mention it.

As for compilers for low power, a couple of years ago I surveyed the field of compiler optimizations for low power systems. I found a number of papers claiming to address this problem, but one of the first sentences would be something to the effect of "Programs that run faster use less power, so we focus on improving program performance." While the work may be significant, it's being recast as something it is not; the community, that's you, should require that papers with disclaimers like this be rewritten and resubmitted, probably to a different conference or journal.

And finally, whole program optimization and performance feedback are becoming an important part of the whole toolkit for optimizations. Whole program optimization or IPA is becoming standard practice in the general purpose community, at least for benchmarks, and is increasingly important for the user community; it opens up a whole new array of optimization opportunities. Performance feedback is an effective way to bring the programmer into the optimization loop; the programmer can do things the compiler can't even dream of, and if we can focus that creativity in the right place, good things happen.