Open Sourcing Your Software is Not a Sustainability Strategy – Until it Is!

David E. Bernholdt, Oak Ridge National Laboratory

When asked how they plan to sustain their software, many (naïve) software developers will say that they plan to make it open source. And that's often their whole plan. There is an assumption that the mere act of exposing the software to the public will create a community who are able and willing to contribute to the support, maintenance, and perhaps the enhancement of the software product.

Those who have more experience observing how open source software works will realize that it is very rare for a project to reap significant benefits from the broader community. The first problem is the signal-to-noise ratio. There are a *lot* of open source codes out there; more every day. It is much rarer for codes to be retired, withdrawn, or for the maintainers to openly state that they should no longer be used. So how is an particular code to stand out, to be noticed, and attract contributors? Basically, it takes work. It is not sufficient for a piece of computational science and engineering (CSE) software to be used in high-quality papers by the developers – though that certainly helps gain recognition. The code needs to be of high enough quality, capability and generality to both have value to others, and be trustworthy. This if often work above and beyond what is needed by an individual developer or development team. Do the developers recognize this need? Do the do the extra work? Sometimes they do, but often they do not.

Code quality and the level of effort required to understand someone else's code is one of the oft-stated reasons for preferring to reinvent rather than reuse existing software. But we must acknowledge that many in the CSE and technical computing community have a strong "not invented here" bias which influences their interest in getting involved in software projects that originated elsewhere. Depending on the circumstances, many rationalizations can be invoked to support this bias.

If a code does manage to gain recognition and interest from the broader community, then next question is who are these people? The majority are like to be users of the software. As such, they're like to report issues with the code and "suggest" enhancements (often much less politely than that), which can be viewed as a "cost" of opening up a code. How many of these users are likely to have the skills, experience, and willingness to actually contribute to addressing those costs? Typically, much less likely. And when such people step forward, the development team needs to be ready to provide a welcoming and supportive environment to encourage their on-going contribution. Once again, the core developers need to be ready to incorporate contributions that may not be completely aligned with the goals of the core development team, from contributors who may not be that familiar with team's preferred development practices, etc. Unfortunately, it is not uncommon for code teams who reach this point to find themselves unable or unwilling to work with outside contributors, and thus, unable to keep them in the long run.

So, while it is not impossible for open source projects to develop a community of contributors, it requires a fair amount of work beyond simply slapping an open source license on the code and making it available on a public hosting site, and it is relatively rare for projects to achieve. When it does happen, however, it can be extremely powerful. Often a noteworthy aspect of projects like these that have managed to cross the chasm to sustainability as open source projects is the size and breadth of their constituencies. By their nature, this can be challenging for CSE applications to achieve, though if a research community manages to come together around a small number of widely used codes, it can happen. There are also some key examples of success from the infrastructure for CSE and high-performance computing (HPC), such as compiler suites like GCC and LLVM. These are somewhat akin to research software, in that they are under continual development, in response to the continuous stream of new hardware architectures that must be supported, advances in the languages and runtimes they support: not only the ISO language standards, like C, C++, and Fortran, but for CSE/HPC, also things like OpenACC and OpenMP. There is often a co-design aspect between the applications, the language/runtime standards, and the compilers that implement them, so there can be a significant amount of R&D associated with these tools that we usually think of as production.

As far as how they engage with their communities, GCC and LLVM tend to operate rather differently. While LLVM is very open and has a large number of contributors, GCC is much more controlled and there are actually relatively few organizations or individuals who are "trusted" to contribute to the code base. I believe that GCC takes this approach as a quality control measure, but LLVM is also recognized as having a high level of quality, so it is not the only way to achieve it.

Another interesting aspect of these projects is the level of involvement of for-profit companies in contributing to these open source projects. Companies clearly find value in contributing staff time to the projects. The use of a copyleft license for GCC means that the enhancements tend to go back into the public code base. LLVM, on the other hand, uses a permissive license, so that companies can choose to contribute back to the public code base, or keep some enhancements private and build separate products based on the LLVM tool chain. Most often, HPC vendors are *expected* to provide compilers in conjunction with the systems they sell. I don't have insight into the financials within such companies, but compilers are large, complex pieces of software that are hard to maintain and support, and I don't think any company is making money off of their compilers. Certainly there are few standalone companies that produce compilers, and more defunct companies than currently active ones. So the CSE/HPC community now finds itself in the interesting situation that an increasing fraction of the compilers that are available for any given hardware system are derivatives of LLVM.

At the same time, LLVM (much more than GCC), has become the vehicle of choice for a growing number of R&D activities as well, many of them being undertaken by Department of Energy (DOE)-funded researchers in support of CSE/HPC needs. Funding to the IBM compiler research group (rather than the product group) via the CORAL-1 procurement (the LLNL Sierra and ORNL Summit systems) led to initial support in LLVM for GPU offload for OpenMP. The Exascale Computing Project (ECP) supports further work on OpenACC and OpenMP in LLVM, as well as a Fortran front-end. (ECP and other code teams also rely on LLVM as one of the most responsive compilers to the evolution of the ISO C++ standard, allowing them to adopt new language features sooner, to the benefit of their software development.)

Which brings us back to the question of sustainability. It is not enough to make a contribution to a open source project. If it is to be meaningful, it needs to be maintained and supported. Especially in a product like a compiler. Open source projects need to have strategies to ensure that they can continue to support contributions that they accept. Ultimately, this can lead to friction in the contribution process: if the code team is not confident that the contributor will be able to maintain their contribution in the long term, they may be reluctant to accept it, especially more complex or larger contributions, and those that may be further from the knowledge and experience of the core maintainers. In LLVM, there are a reasonably broad community of interest and knowledge in OpenMP, less so for OpenACC, and Fortran is very much a CSE-specific niche.

Nonetheless, these capabilities are very important to the CSE/HPC community, and as noted, we are increasingly relying on LLVM as part of, or the whole of, the compiler tool chains on our HPC systems. So how do we expect those important contributions to be sustained? The ECP project will end in 2023. What follows is not clear, but historically, DOE has not directly supported sustainment of software products – they support R&D. In fact, it has long been the case that R&D project leaders will direct some of their resources to the maintenance of key software that is either critical to, or a product of, their research, with the tacit agreement of program managers. Is this enough?

I would argue that the situation needs to change. While companies often commit significant resources to open source projects for extended periods, with an appropriate business case to justify it, research funding is ephemeral, coming and going in cycles of 3-5 years, and sometimes more quickly. It is hard for research-funded contributors to make strong commitments to the maintenance and support of their contributions, regardless of the strength of the "business case" for it – the level of important to others who might be funded by the same program or agency, much less its broader impact. The DOE is not there yet, though there is an opportunity for things to change. A subcommittee of the Advanced Scientific Computing Advisory Committee (ASCAC) has been charged to report on "Transitioning From the Exascale Project", providing advice on how to sustain investments made in the ECP project.

Although the charge is specific to the ECP, that is far from the only DOE program supporting tools, libraries, and even applications that are important to broader constituencies. And the DOE is far from the only agency supporting the development of important software products. The National Science Foundation (NSF) has begun to acknowledge and support this need with its Software Infrastructure for Sustained Innovation (SI2) family of programs. I'm not sure where other agencies are on this.

Personally, my hope is that the DOE Office of Advanced Scientific Computing Research (ASCR) will take the ECP funding roll-off and the subcommittee report as an opportunity to develop a more comprehensive strategy to provide long-term support for important software products that is not so directly tied to research funding.

Perhaps, this could even be used as a level to engage in a higher level of discussion, across funding agencies (for example through the Networking and Information Technology Research and Development (NITRD) program) to recognize and address the connection between innovation *in* software and innovation that relies on software and the need to identify and sustain software contributions that become important to on-going innovation.

So, open sourcing your software is not a sustainability plan unless and until you manage to cross the chasm and develop the community necessary to make it sustainable. And from the standpoint of that community, would-be contributors need the backing of their institutions and sponsors to ensure that they can continue to participate in the community by supporting and maintaining their contributions. This is often a challenge for those operating in a research environment today. Further discussions are needed with funding agencies about the business case for this kind of support.