Resilience is a Software Engineering Issue

a position paper for the Software Productivity for Extreme-Scale Science (SWP4XS) Workshop, by

David E. Bernholdt, Al Geist, and Barney Maccabe
Oak Ridge National Laboratory
bernholdtde@ornl.gov

Historically, high-performance computing (HPC) applications have relied heavily on the underlying hardware to detect and mask faults, leaving the application itself to deal only with catastrophic errors, typically through checkpoint/restart mechanisms. Trends suggest that we will not be able to rely on future HPC systems in the same way, increasingly pushing resilience issues into the software stack. Though there is an extensive body of research in resilience mechanisms at many levels of the software stack, none of these projects offer a silver bullet. The future of resilience in HPC applications ultimately boils down to *good software engineering.*

Today's HPC systems experience faults on a frequent basis. However, we have designed the hardware for these systems to protect against many of these faults with error detection and correction capabilities on many memory elements (main memory, caches, registers) and data paths (networks, busses, and within chips). While uncorrectable and even undetected errors do occur, their impact on the results are infrequent enough on today's systems that times to system interrupt are measured in days for systems with tens of thousands of nodes, hundreds of thousands of cores, and millions of potentially-faulty components.

Future systems are expected to offer much more challenging environments in terms of resilience. First, the simple scale increases of coming generations of machines will worsen the resilience situation on a purely statistical basis – more components lead to higher likelihoods of failures in the aggregate system, assuming the failure rates of individual components hold steady. Unfortunately, that assumption seems unlikely to hold up either. First, trends in lithographic techniques result in progressively smaller feature sizes (i.e. transistors are comprised of fewer atoms), which are increasingly vulnerable to external upsets from thermal and electrical fluctuations, particle strikes, and other sources. Chip makers are unlikely to give up easily on their long progression of lithographic improvements because they provide an important means of reducing device power consumption. Second, in an attempt to reduce power consumption even faster, chip makers are increasingly looking to drive the chips at "near-threshold" voltage, which exacerbates their vulnerabilities. There are predictions in particular for dramatic increases in undetected transient or soft errors in memory (also known as silent data corruption) which are particularly insidious because they can result in an application producing incorrect output, without necessarily giving any overt indication that the error has occurred.

Whether these challenges can continue to be addressed exclusively at the hardware level, as they have been in the past, is an open question at the moment. By their nature, hardware-based resilience solutions are relatively blunt instruments – in today's systems they are "on" whether they are needed or not (i.e. media processing vs. financial calculations)[1]. Further, there may be some situations where a fault can be more easily and efficiently handled somewhere in the software stack rather than always at the lowest levels (hardware). All these facts point to the necessity going forward of a new, cooperative model of resilience, involving (perhaps switchable) detection and correction capabilities in the hardware and at multiple levels of the software stack, up to and including the application itself.

[1] One counter-example is the fact that in some recent models of NVIDIA GPU accelerators, error correction (ECC) for the on-card memory can be switched off and on, with impacts on the capacity and performance of the memory system, of course.

While we can't say yet exactly what these future systems will look like to the programmer, we can anticipate some general characteristics. First, adaptable software systems will become much more important to ensuring it has the necessary reliability. While the primary resilience-related questions a programmer has to answer today involve where to checkpoint the code, and what state needs to be saved, tomorrow's applications can expect multiple options at different levels of the stack, with different trade-offs. This is, in some respects, similar to the situation with parallelism – exposing parallelism at multiple levels, and determining the trade-offs in which to use (which may be machine- and problem-specific) will require new ways of thinking about the structure, or architecture of the entire software system. This may entail the development of new tools for resilience, just as we have lately embarked on the serious exploration of new tools for parallelism for extreme-scale computing (runtime environments). But, just as surely, and especially in the early days, there will also be a strong reliance on identification and dissemination of abstractions, patterns, best practices, and other approaches that *guide the design and development* of the software without necessarily dictating specific tools. Second, detection of problems within the application will be key to error containment and efficient recovery. If an error propagates too widely in time or space, before detection then recovery becomes expensive. Developers must have a strong understanding of the *requirements and specifications* for the software system, and must utilize strong *testing and verification* regimes to ensure that the software conforms to those requirements. Fundamentally, in an environment in which errors are frequent and undetected in the hardware, programmers must have the understanding of, and confidence in their software implementations to be able to distinguish erroneous results from correct ones if they are to have any hope of being able to deliver reliable results. This includes, for example, understanding and documenting the bounds of applicability of algorithms, so that both inputs and outputs can be *verified* as being "reasonable", even if it is not possible (without significant additional expense) to confidently say that they are correct in the strictest sense. This also suggests that testing and detection schemes will have to be expanded. Whereas today most *testing*, where it is done at all, is done as a set of jobs that are quite separate from "production" science runs, in the future, a great deal more testing and verification will have to be done during production runs in order to have confidence in the specific results of the run.

The concept of error containment will have to be engineered into future software systems both to make on-the-fly error recovery more efficient and to avoid having an error propagate into a checkpoint file, corrupting the last resort recovery method.

All of these characteristics fall well within the bounds of the discipline and practice of software engineering. As we begin to figure out the resilience story for extreme-scale computing, we would do well to recognize and leverage this fact. Better software engineering practices can have many benefits, even apart from the anticipated resilience challenges. We now have an opportunity to use a strong HPC-based driver (resilience) as a lever to bring better software engineering practices in general into HPC. Let's embrace it!