5. HPC Usability Research Team

5.1. Team members

Toshiyuki Maeda (Team Leader)

Yoshiki Nishikawa (Visiting Researcher)

Akiko Yoshioka (Assistant)

5.2. Research Activities

The mission of the HPC Usability Research Team is to research and develop a framework and its theories/technologies for liberating large-scale HPC (high-performance computing) to end-users and developers. In order to achieve the goal, we conduct research in the following three fields:

1. Computing portal

   In a conventional HPC usage scenario, users live in a closed world. That is, users have to play roles of software developers, service providers, data suppliers, and end users. Therefore, a very limited number of skilled HPC elites can enjoy the power of HPC, while the general public sometimes gives a suspicious look to the benefit of HPC. In order to address the problem, we are designing and implementing a computing portal framework that lowers the threshold for using, providing, and aggregating computing/data services on HPC systems, and liberates the power of HPC to the public.

2. Virtualization

   Virtualization is a technology for realizing virtual computers on real (physical) computers. One big problem of the above mentioned computer portal that can be used by wide range of users simultaneously is how to ensure safety, security, and fairness among multiple users and computing/data service providers. In order to solve the problem, we plan to utilize the virtualization technology because virtual computers are isolated from each other, thus it is easier to ensure safety and security. Moreover, resource allocation can be more flexible than the conventional job scheduling because resource can be allocated in a find-grained and dynamic way. We also study lightweight virtualization techniques for realizing virtual large-scale HPC for test, debug, and verification of computing/data services.

3. Software verification

   Software verification is a technology that tries to prove certain properties of programs by analyzing them. By utilizing software verification techniques, we can prove that a program does not contain a certain kind of bug. For example, the byte-code verification of Java VM ensures memory safety of programs. That is, programs that pass the verification never perform illegal memory operations at runtime. Another big problem of the above mentioned computing portal framework is that one computing service can be consists of multiple computing services that are provided by different providers. Therefore, if a bug or malicious attack code is contained in one of the computing services, it may affect the whole computing service (or the entire portal system). In order to address the problem, we plan to research and develop software verification technologies for large-scale parallel programs.

5.3. Research Results and Achievements

5.3.1. Protocol/API Design for Computing Portal Framework

As a first step of designing and implementing a computing portal framework that can be used by wide range of users, in FY 2012, we designed an experimental API/protocol for computing services. More specifically, we designed APIs/protocols that handle registration of services and their providers, registration and authentication of users for each registered service, invocation of computing services, data sharing among multiple computing services, and so on.

The APIs/protocols are designed in such a way to work with the current popular web-based application frameworks (e.g., HTTP, JSON, etc.). Therefore, in theory, we can write programs that utilize multiple computing services in various programming languages (e.g., Ruby, Python, JavaScript, etc.). In addition, a computing service can be registered and published by writing a simple XML file, provided that the application programs of the computing service are installed on the backend system. Based on the experimental API/protocol, we are also implementing a prototype system that can be used to publish existing applications as computing services.

One limitation of the designed experimental APIs/protocols and their prototype implementation is that security mechanisms are not fully integrated or realized yet. That is, malicious computing services can access data of other computer services, and/or leak data of one user to another user. To address the problem, we are planning to improve the APIs/protocols from the viewpoint of security and enhance the prototype implementation by integrating virtualization techniques.

5.3.2. Virtualization Techniques

1. Lightweight virtualization for testing/debugging parallel programs

   In order to utilize the full power of today’s HPC systems as the K computer, users have to write massively parallel programs. However, writing parallel programs is difficult compared to conventional sequential programming. This is because parallel programs have inherent non-determinacy (e.g., process/thread execution order), that is, even if a parallel program contains a bug, it is not always easy to reproduce the bug. In addition, performance bottlenecks of parallel programs are not apparent from their source code because network latency, synchronization costs, scalability, etc. cannot be inferred directly from the source code. One possible solution to the problems is to utilize static source code analysis and dynamic performance profiling, however, some kind of bugs arise only when the number of processes/threads used by parallel programs is huge (e.g., several tens of thousands or more).

In order to address the above mentioned problem, in FY 2012, we designed a lightweight network virtualization technique that is useful for testing/debugging parallel programs. More specifically, we designed a virtualization framework that is able to provide several tens to hundreds of virtual execution environments per real (physical) execution environment. With the virtualization framework, users can test, debug, and/or profile their parallel programs on a limited number of physical computing nodes as if the programs run on a huge number of nodes.

The key of our virtualization framework is to virtualize network related operations at the level of shared libraries. Current popular virtualization technologies adopt virtualization at the level of CPU (with hardware assists) or OS (system calls), which is heavier than the level of shared libraries. This is because main purpose of the popular virtualization technologies is to provide virtual execution environments that are hard to be distinguished from real (physical) ones. Our virtualization framework, on the other hand, gives up to provide such a realistic virtual execution environment, but instead aims to provide as much as possible of virtual execution environments per real (physical) one. By adopting virtualization at the level of shared libraries, the performance overheads of hooking system calls and/or CPU events (e.g., interrupts and exceptions) can be eliminated (while statically linked programs, that is, programs do not rely on shared libraries, will not be virtualized correctly).

In addition, our virtualization framework tries to reduce (or eliminate) exchange of virtual network routing information among real (physical) nodes. In order to correctly route packets from one virtual execution environment to another, all the physical nodes have to share the routing information of the virtual networks because the virtual execution environments may reside in different physical nodes. Therefore, if a single physical node manages the routing information, the node will become a performance bottleneck because all the other physical nodes have to synchronize with the node each time they need to route packets.

To address the problem, our virtualization framework tries to distribute the routing information statically (that is, before executing programs in virtual execution environments) as much as possible. In addition, even if dynamic updating of the routing information is inevitable, our virtualization framework tries to minimize synchronization between multiple physical nodes by separating allocation pool of physical (real) network ports statically.

Based on the above mentioned design, we are also implementing its prototype system, and will continue the development in FY 2013.

2. CPU emulator for SPARC 64 VIIIfx

   One problem of the current K computer is that its CPU is based on the SPARC 64 VIIIfx architecture, which is not so popular in the current PC market dominated by the x86-64 architecture. Therefore, users have to use a cross compiler which generates SPARC binaries on the front-end node of the K computer, or their x86-64 PCs. Thus, it is somewhat difficult to develop software for the K computer because programmers cannot test their programs on their ordinary development environment (e.g., PCs), and have to use the K computer even if they want to conduct very small tests.

   To address the problem, we are implementing a CPU emulator for SPARC 64 VIIIfx. A CPU emulator is a program that emulates behavior of a CPU architecture that is different from that of the host machine on which the emulator itself runs. Our CPU emulator is able to execute SPARC 64 VIIIfx executables on ordinary x86-64 PCs. More specifically, we modified the existing CPU emulator QEMU to support the features specific to the SPARC 64 VIIIfx architecture (e.g., extended general purpose/floating-pointer registers, SIMD extension, etc.).

   In FY 2012, we developed a rapid prototype of our CPU emulator. The prototype only supports a subset of the SPARC 64 VIIIfx architecture and very small number of system calls. Moreover, its emulation is incomplete in the sense that it does not adhere to the formal specification of SPARC 64 VIIIfx and sometimes crashes abnormally. We are planning to develop a more stable and faithful prototype which is usable for practical program development and testing in FY 2013.

5.3.3. Software Model Checking for Partitioned Global Address Space Language (Joint Work with Programming Environment Research Team)

Partitioned Global Address Space Languages (or, PGAS languages) are programming languages for distributed computing systems where the systems consist of large number of computing nodes and their memories are distributed among the nodes. In the PGAS languages, all the processes and/or threads in a program can share a single address space even though the memories are distributed, as in traditional distributed shared memory (DSM) systems. One of the distinguishing features of the PGAS languages is that the shared address space can be partitioned into sub-spaces and they can be bound to a specific process and/or thread explicitly. Thus, programmers can write a locality-aware program which is essential to achieve high performance on massively-parallel distributed memory systems of today (and future).

Despite the above mentioned advantage, one big problem with PGAS languages is that programmers can easily introduce concurrency bugs. For example, if multiple threads access a portion of a single address space simultaneously without proper synchronizations, race condition bugs can be easily introduced even if the accessed portion is bound to a specific process and/or thread. To make things worse, introducing synchronizations is not as easy as it sounds because excessive use of synchronizations severely degrades performance, while lack of them introduces hard-to-debug and non-reproducible concurrency bugs.

To address the problem, in FY 2012, we proposed and implemented a software model checking framework for PGAS languages. Software model checking is a program verification approach which tries to prove that a given program satisfies a certain property by exploring all the program states that can be reached during program execution. One problem of model checking PGAS programs is that it tends to suffer from the state explosion problem because these programs allow concurrent and/or parallel execution and memory sharing. To avoid this problem, it is essential to perform proper abstractions based on the properties to be verified because they can dramatically reduce the number of states to be explored. However, it is not always easy to automatically infer proper abstractions because programs and properties to be verified vary.

To address the state explosion problem, we proposed a model checking framework that includes user-definable abstractions. The key idea of the framework is that it exposes the intermediate representation of the program's abstract syntax tree, enabling users to define their own abstractions flexibly and concisely by creating a translator to translate the trees. We also implemented CAF-SPIN, our proof-of-concept prototype of a model checking tool for Coarray Fortran. The experimental results with CAF-SPIN showed that abstractions can be defined easily and concisely by users, and the number of states to be explored for model checking is dramatically reduced with the abstractions.

Moreover, we also implemented XMP-SPIN, our software model checking tool for XcalableMP (http://www.xcalablemp.org/), and conducted several experiments with XMP-SPIN. More specifically, we conducted model checking of a number of parallel stencil computations written in XcalableMP (from small test programs to large application programs). Although stencil computations offer a simple and powerful programming style in parallel programming, they are sometimes error prone when considering optimization and parallelization because optimization of stencil computation may involve complex loop transformations and/or array reindexing, and parallelization requires explicit communication (data synchronizations) among multiple processes. In the experiments with XMP-SPIN, we checked whether there are no missing or redundant data synchronizations in the target programs, and successfully found four bugs in a reasonable time with a reasonable amount of memory.

5.4. Schedule and Future Plan

In FY 2013, we will improve the APIs/protocols for our computing portal framework designed in FY 2012 from the viewpoint of security. As mentioned above, the current experimental

APIs/protocols and their prototype implementation do not consider security mechanisms seriously. Therefore, malicious computing services can access data of other computer services, and/or leak data of one user to another user. In addition to the improvement of the API/protocols, we will also improve and enhance the prototype implementation by integrating virtualization techniques.

Regarding the virtualization technologies, we will continue to implement the lightweight network virtualization framework for testing/debugging parallel programs and the CPU emulator for SPARC 64 VIIIfx for running binary executables for the K computer on the ordinary x86-64 computers.

Regarding the software verification, we will conduct more experiments with CAF-SPIN and XMP-SPIN to evaluate their effectiveness and practicality. In addition, we will also consider applying our software model checking approach to other PGAS languages. Moreover, we will also plan to extend our model checking approach to take into consideration relaxed memory consistency models.

As a slightly longer-term goal, we plan to start integrating the research results of the virtualization technologies and the software verification into the computing portal somewhere from the second half of FY 2014 to the first half of FY 2015.

5.5. Publication, Presentation and Deliverables

(1) Conference proceedings (Refereed)

1. Tatsuya Abe, Toshiyuki Maeda, and Mitsuhisa Sato. Model checking with user-definable abstraction for partitioned global address space languages. In Proceedings of the 6th Conference on Partitioned Global Address Space Programming Models (PGAS2012), Online. Santa Barbara, October 2012.
2. Tatsuya Abe, Toshiyuki Maeda, and Mitsuhisa Sato. Model checking stencil computations written in a partitioned global address space language. In Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS2013), to appear.

(2) Presentations

1. Toshiyuki Maeda. Brief introduction of HPC usability research team. The 3rd AICS International Symposium, March 2013.

(3) Deliverables

1. CAF-SPIN: A software model checker for Coarray Fortran (in preparation for release)

(4) Posters and presentations

-None

(5) Patents and Deliverables

-None