A Compiler-Directed Approach to Software Cache Coherence Management via X10 Programs Transformations

Al. Levchenko

SPbPU Supercomputer Center

In upcoming many-core architectures, the cache coherence problem remains an important challenge, as traditional hardware schemes are not always sufficient to maintain coherence at current high degrees of parallelism. Recent research efforts [1–3] have shown that software-managed coherence is an all-sufficient approach per se both for the emerging hybrid software-hardware coherence schemes and current non-cache-coherent architectures with purely software support for partitioned shared memory. Furthermore, the compilers and runtime systems based on partitioned global address space model can automatically control data transferring, thus eliminating overall memory complexity from the user.

The specific contributions of this paper are to provide a number of compiler related considerations towards maintaining cache coherence through the sequential source-to-source transformations of X10 programs. Proposed multi-tiered approach includes (1) high-level optimizations of X10 AST (Abstract Syntax Tree), (2) optimizations at the level of the resulting Native X10 code, namely source-to-source C++ translation, and, last but not least, (3) runtime optimization, including the level of the X10 coherence protocol (most promising studies for the third level now left for future work). At the first level, the previously developed chain based on X10 and ROSE compiler infrastructure [4] was used as a mid-end to provide a set of APIs, handle input sources and implement optimizations. The principle of this connection is that X10 compiler parser constructs an AST from X10 sources, then resulting tree is transformed into ROSE AST, that also supports APGAS-specific nodes like *async/at/finish*. At this point, the first-level optimizations developed within the framework of this work contribute to the efforts being made in paper [4] by reusing loop unrolling, live variable and use-def analysis that ROSE provides. At the final stage of the first level, the unparser produces the faster code, namely C++ back-end, using the appropriate X10RT implementation, e.g. TCP/IP sockets, standalone host or MPI. Managed X10, i.e. alternative Java back-end, is not considered here due to some degree of uncertainty regarding the Java support of shared memory exceeding 3Tb per node. Unfortunately, the machine-generated code derived from the first level can be suboptimal, because the native X10 compiler on several occasions can omit a number of optimization opportunities, as confirmed by authors of the X10/ROSE connection [4]. Due to lack of machine-generated C++ optimizations, the second level optimizations of proposed approach are implemented as an entire compiler which is back-end for x10c++ compiler (i.e. native X10 in this case) and is a front-end for system C++ compiler. Allocating C++ optimizations to a separate (second) level allows to implement an increased set of optimizations for any (A)PGAS code on multiple architectures. This paper investigates the limited application of the algorithm of coherence instructions generation using parallel region analysis, presented in the work [3].

Early-stage experimental results are reported to show the correctness of the previously mentioned optimizations, at least maintaining unchanged or improving the performance of the optimized code. In this paper, the widely-studied hydrodynamics proxy application LULESH 2.0 was used to compare overall performance after the first and second level optimizations with the results of the reference non-PGAS implementations. As LULESH is representative for the wide range of current scientific HPC applications, it has been redesigned in co-design efforts to support multiple programming models (e.g. hybrid MPI/OpenMP, Charm++, Chapel, MPI/UPC++, X10, etc.) [5] on target systems like BG/Q and Cray XE6. In this paper, the original C++, X10

port and optimized X10 code of LULESH are compared to estimate the *Grind Time*, which is average per-element time (in microseconds) required to update the solution variables through one time increment. For this purpose, the ccNUMA machine with up to 12Tb logically indivisible memory was used with scheduling one place per NUMA node. Despite the fact that in the target system the basic cache coherence is provided through the hardware directory based protocol, satisfactory performance of such systems with up to 24Tb RAM will inevitably require a hybrid scheme implying the co-existence of both hardware and, as its superstructure, software cache coherence support. Table 1 compares the preliminary results of the optimized code with the reference version and demonstrates that transformed LULESH exhibits superior *Grind Time* to a reference code. Despite the minor reported changes in this primary metric, the results presented show that the toolchain under study is fully functional, and thus X10 programs transformations can be considerably extended, giving priority to the second and third levels of the approach.

Table 1. LULESH Grind Time on target ccNUMA system (lower is better)

LULESH models	Grind time (μs/z/c)
Original C++ (hybrid MPI/OpenMP) vs. X10 port (reference code)	3.74/3.85
(+) X10 port optimizations in ROSE (first level)	3.61
(+) Native X10 optimizations (second level)	**3.54**

Future work will investigate the opportunities to introduce APGAS-specific optimizations at the second and third levels of the presented approach, e.g. an implementation of novel techniques for transferring pointered data structures via shared memory shown in a recent paper [1]. In the context of software-managed coherence, a topic of long-run future work will be to explore the opportunities for improving not only performance, but also fault tolerance of APGAS/X10 programs.

References

1. M. Mohr, C. Tradowsky. Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores // Design, Automation and Test in Europe Conference and Exhibition. 2017. P. 1781–1786. URL: http://dx.doi.org/10.23919/DATE.2017.7927281

2. A. Jimborean, J. Waern, P. Ekemark, S. Kaxiras, and A. Ros. Automatic Detection of Extended Data-Race-Free Regions // 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2017. P. 14–26. URL: http://dx.doi.org/10.1109/CGO.2017.7863725

3. S. Tavarageri, W. Kim, J. Torrellas, and P. Sadayappan. Compiler Support for Software Cache Coherence // 2016 IEEE 23rd International Conference on High Performance Computing (HiPC). 2016. P. 341–350. URL: http://dx.doi.org/10.1109/HiPC.2016.047

4. M. Horie, M. Takeuchi, K. Kawachiya, and D. Grove. Optimization of X10 Programs with ROSE Compiler Infrastructure // Proceedings of the ACM SIGPLAN Workshop on X10, ser. X10 2015. New York, NY, USA: ACM. 2015. P. 19–24. URL: http://doi.acm.org/10.1145/2771774.2771777

5. J. M. Hashmi, K. Hamidouche, and D. K. Panda. Enabling Performance Efficient Runtime Support for Hybrid MPI+UPC++ Programming Models // 2016 IEEE 18th International Conference on High Performance Computing and Communications. 2016. P. 1180–1187. URL: http://dx.doi.org/10.1109/HPCC-SmartCity-DSS.2016.0165