ZettaBricks: A Language, Compiler, and Runtime Environment for Anyscale Computing

Saman Amarasinghe, Alan Edelman, Una-May O'Reilly, Martin Rinard, Chris Hill
Massachusetts Institute of Technology

Abstract: With the dawn of the multicore era, programmers are being challenged to write code that performs well on an increasingly diverse array of architectures. A single program or library may be used on systems ranging in power from large clusters of servers with hundreds or thousands of cores to small single-core netbooks or cell phones. A program may need to run efficiently both on architectures with many simple cores and on those with fewer monolithic cores. Some of the systems a program encounters might have GPU coprocessors, while others might not. Looking forward, processor designs such asymmetric multicore, with different types of cores on a single chip, will present an even greater challenge for programmers to utilize effectively.

Programmers often find they must make algorithmic changes to their program in order to get performance when moving between these different types of architectures. For example, when moving from a sequential to a parallel architecture a programmer may have to change his algorithm to expose more parallelism. A similar need for algorithmic change arises when switching between different types of parallel architectures. In a recent paper, we showed that even when changing between similar architectures with the same number of cores, dramatic algorithmic changes can be required to optimize performance.

To obtain portable performance in this new world of more diverse architectures, we must build programs that can adapt to whatever hardware platform they are currently running on. Our team of researchers at MIT has developed PetaBricks, a new, implicitly parallel language and compiler that allows the user to specify algorithmic choices at the language level. Using this mechanism, PetaBricks programs define not a single algorithmic path, but a search space of possible paths. This flexibility allows our compiler to build programs that can automatically adapt, with empirical autotuning, to every architecture they encounter.

Project Objective: We are developing ZettaBricks: a programming language, a compiler and a runtime environment that can drastically reduce the burden of programming exascale systems by eliminating the need for architecture-specific performance tuning and simplifying error handling. ZettaBricks will provide a *Write Once Run Anywhere* programming model where a single program can work efficiently on a megascale handheld device and also scale to an exascale supercomputer.

Key Ideas: ZettaBricks introduces algorithmic choice at the language level, letting the programmer provide high level descriptions of multiple algorithms for a given problem, each of which may work best under different conditions. These choices are further expanded in the compiler by automatically generating different iteration orders, computation partitions, data layouts, and replication strategies. In order to handle deep memory hierarchies as well as hardware failures, ZettaBricks introduces Bricks, a hierarchical data representation and data management system that seamlessly spans from in-core caches to main memory to local disks to SANs and other distributed storage systems. All these choices are exposed to a machine learning runtime system that will use application-specific rapid online adaptation to find the best choice given the current execution environment. ZettaBricks will also incorporate architecture-specific long-term learning to evolve the compiler and runtime system to prune away perpetually underperforming choices. In long computations on large failure-prone clusters, ZettaBricks will provide reliability and resiliency by duplicating and re-computing data as needed. ZettaBricks also introduces data structure repair to identify and fix rare input, algorithmic and hardware errors that can ruin a long running computation. ZettaBricks will also support variable precision computations by selecting the algorithms and levels of precision for intermediate values that efficiently lead to output with the required

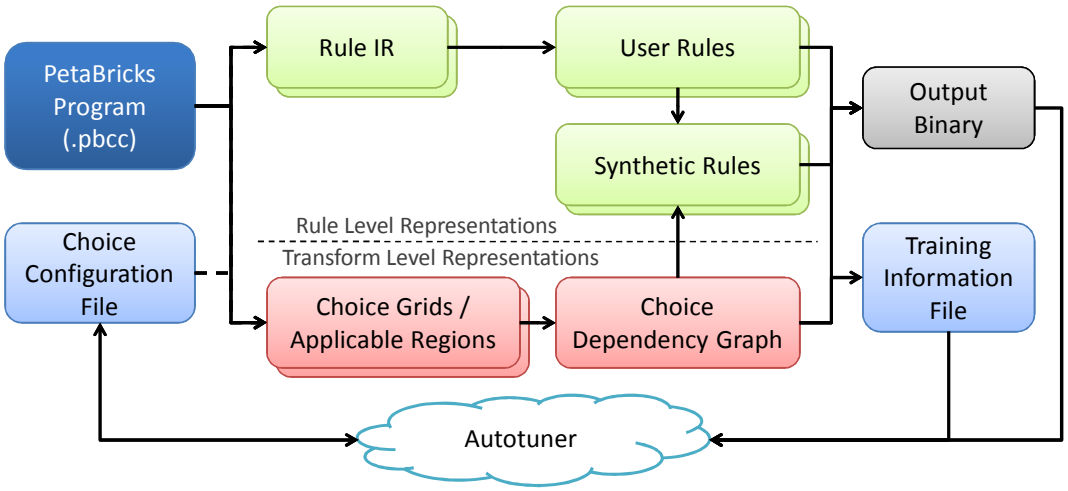

Figure 1: Flow for the compilation of a PetaBricks program with a *single* transform. (Additional transforms would cause the center part of the diagram to be duplicated.)

precision. The ZettaBricks system will be built on our current successful PetaBricks effort that demonstrated some of these concepts on shared memory architectures.

Expected Impact: A programming environment that lets scientists and researchers worry about the science, not the intricacies of the execution environment, will have a huge impact on the science community. Imagine a world where a scientist writes a program by describing a handful of possible solvers for a problem in a simple language focusing on the science with no regard to where and how it is run. Then she uses a small data set in her own megascale desktop to test and debug the algorithms to make sure that the right problem is solved. When satisfied, she can start doing science on a terascale cluster at her institution with a real dataset. Finally when the science is proven to be good, she can get some time on an exascale machine at a national lab and run her program on a global dataset. All this without ever changing the program or any performance tuning! Today, programmers at Google have a similar workflow for a limited class of applications with map-reduce. We would like to make this the future for the scientific community with ZettaBricks.

ZettaBricks Compiler Infrastructure: Figure 1 displays the general flow for the compilation of a PetaBricks transform. Compilation is split into two representations. The first representation operates at the rule level, a construct in the language, and is similar to a traditional high level sequential intermediate representation. The second representation operates at the transform level, and is responsible for managing choices and for parallel code synthesis.

The compilation generates an *output binary* and a *training information file* containing static analysis information. These two outputs are used by the autotuner to search the space of possible algorithmic paths. Autotuning creates a *choice configuration file*, which can either be used by the output binary to run directly or can be fed back into the compiler to allow additional optimizations. The autotuner follows a genetic algorithm approach to search through the available choice space. It maintains a population of candidate algorithms which it continually expands using a fixed set of high level mutators, which are generated through static code analysis. The autotuner then prunes the population in order to allow it to evolve more optimal algorithms. The input sizes used for testing during this process grow exponentially, which naturally exploits any optimal substructure inherent to most programs.