Research Statement: Sean Treichler

Today's computer architectures are too complicated for human programmers to reason about precisely. Today's applications are too complicated for compilers and their associated runtimes to reason about precisely. The immediate result of this gap between human and compiler understanding is that most applications fall far short of the performance theoretically available on existing systems. My research interests are in breaking down the traditional boundaries of programming languages, compilers, runtimes, and system microarchitecture to build both new programming models and new computer architectures that allow human programmers, software tools, and the underlying hardware to work together to map a desired computation onto a target system and get the best performance possible.

Existing computer systems are often heterogeneous, mixing processors of different types (e.g. CPUs and GPUs). This specialization allows for both absolute performance and power efficiency that can be an order of magnitude better than would be possible from a homogeneous system design, and the trend is towards further specialization in the future. In these systems, each CPU socket or GPU has its own local memory. Access to another processor's memory is typically done at reduced performance and may require software assistance. An application must be *mapped* onto such a machine by deciding on which processor each part of the computation will be performed and in which memory (or memories) the application's data will be placed. This mapping may be *static*, remaining unchanged for the entire run of the application, or it may be *dynamic*, changing based on variations in the application workload and/or the system behavior. The complexity of both applications and computer architectures is such that optimal mapping for an application is often not obvious, and can vary significantly between different computer architectures.

Current programming models fall into two camps with respect to the mapping of an application onto a given architecture. The first camp consists of "low-level" languages in which the programmer has complete control over the mapping, but must manually implement the desired mapping, adding code to the application to control when and where code runs and data movement is performed. This effort can be significant (recent efforts to port applications to new supercomputer architectures often required more than an engineer-year's worth of effort per application) and must be redone for each new mapping. Recognizing that a programmer's time is often then most critical resource, the second camp of programming models take ownership of the mapping (both its selection and its implementation), offering the promise of portable application code that can run on a variety of computer architectures. The systems that obtain the best performance with this approach are those that specialize for a narrow application domain (e.g. linear algebra) — extracting the necessary information from arbitrary application code written in general-purpose languages like C is impossible to do precisely in all cases.

A primary goal of my research on the Legion programming model at Stanford was a middle ground between these approaches, providing a way for the programmer to control the mapping from portable code while allowing automated tools (i.e. a compiler and/or runtime system) to handle the laborious job of implementing the mapping. The key concept that made this possible was the introduction of a richer data model to the base language. Legion's *logical regions* allows application code to portably describe what data is used by different parts of the computation. The programmer's desired mapping policy is explained using these same constructs, bridging the understanding gap between the programmer's intent and the Legion runtime's implementation of the task launching and data movement for each type of processor and memory in the system.

The existence of an expressive data model such as Legion's opens doors to many other areas of research. Possibilities include improved program analysis, compiler optimizations of data layouts, and better tools for functional and performance debugging of complicated distributed applications. My recent research has been on extending the data model to describe *dependent partitioning*, a framework of concise (and usually statically verifiable) operations that compute the partitions of application data that are needed to enable parallel execution in a system with a distributed memory hierarchy. The initial results are excellent for computing completely new partitions, and point the way to future work on the very important related problem of efficient computation of incremental updates to partitions (e.g. for dynamic load balancing of an application).

Looking forward, I would like to incorporate the computer architecture itself into the conversation. I believe this will result in systems that are significantly more powerful than can ever be achieved by current methodologies. Part of this improvement will come from blurring the lines between software and hardware. For example, expensive hardware features such as caches and instruction reordering logic may be reduced or replaced by simpler mechanisms that can be controlled by a compiler or runtime that has better knowledge of an application's behavior. In the other direction, transformations typically done by a compiler might be profitably moved to the hardware, which can take advantage of dynamic information not available during compilation. The other part of the potential improvement comes from the ability to explore hardware designs that have traditionally been considered too hard for either humans or tools to reason about individually. Examples include architectures in which processors communicate directly (e.g. via active messages or as a systolic array) rather than through random-access memory, or those with significantly greater heterogeneity, either in processor types (e.g. further specialization for certain workloads) or in performance (e.g. dynamic clock speeds per core based on process, voltage, and temperature variations). By recasting the problem as a team effort between the programmer, the software tools, and the hardware, these boundaries are removed and the space of potentially interesting computer architectures is virtually limitless.