This project proposal is inspired by the following question: how can application programmers utilize heterogeneous systems, given that programmer effort should not scale with system complexity? In the past, when instruction-level parallelism (ILP) and single-core systems were dominant, programmers could fully exploit their hardware platform by writing sequential code. More recently, following parallelism and multi-core systems, writing efficient software has required that the programmer be familiar with multi-threading. At present and in the future, the story has become even more complex with heterogeneous systems, each of which potentially employ a different set of a variety of devices. For programmer productivity to keep pace with system complexity, research must establish methodologies for transparently mapping software to a wide range of systems.

To help meet this challenge, I want to research how a system can perform what I call *device inference* on application software. Device inference is a system's ability to determine which available device is most apt for executing each segment of code within a larger application. In this light, the programmer's responsibility is to write code that *looks* like it will run well on a particular type of device(s). The system's responsibility is to *recognize* similarities between code fragments and available devices' execution models, and take those similarities into account at runtime. Therefore, a key consideration throughout this project will be to study (a) the overheads between performing work offline vs. online, and (b) the tradeoffs between concurrent/interruptive approaches within the context of heterogeneous systems. Broadly speaking, device inference raises the abstraction for writing applications by hiding the target system's device composition.

My research hypothesis is that device inference will depend on how well thread-thread *affinity* and thread-device *fit* can be measured. The affinity between thread A and B refers to how important it is that thread B is run on D given that thread A is run on D, for any chosen device D. Fit refers to how well thread A maps to D, for any chosen device D. To start, I want to study thread memory aliasing, which indicates when and how often multiple threads share data, as one way to measure thread affinity. As it stands, managing threads and the data that they share is already a bottleneck in many single-chip systems[1]. In heterogeneous multi-chip systems, the problem is even more pronounced, given various device interfaces through which data must travel.

To evaluate my hypothesis, I will break my proposal into two parts. The first part, which I will call *characterization*, will be to find sets of thread and device properties that provide a meaningful measure of thread-thread affinity and thread-device fit. The second part, which I will refer to as *lowering*, will be to develop acceptable runtime algorithms for clustering threads by affinity and fitting clusters to devices. The high-level idea behind the two parts is to determine what makes threads run well together on different devices, and to exploit that knowledge when scheduling each thread.

I propose to approach characterization through studying the runtime traces of multi-threaded applications and the architectural makeup of different devices. The key research goals for this part are to determine (a) a suitable set of runtime properties that can be used to measure affinity/fit, and (b) each property's device-specific importance. My research hypothesis predicts that part of goal (a) will be thread memory aliasing. Goal (b) will then address how memory aliasing is a more important consideration for devices with less support for inter-thread data sharing (such as graphics processing units/GPUs), relative

[1] M. Suleman et al., Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures; *ASPLOS, 2009.*

to devices with extensive memory hierarchies (such as multi-core CPUs). Memory aliasing is not the whole story, however. Studying other properties (instruction mix and ILP, for example) will be important in characterizing more applications, and so that findings can contribute back to related work on architectural analytic models. (For example, Tejas Karkhanis et al.'s[2] processor model takes runtime profiling information as input).

To study lowering, I propose an offline-online *learning* approach to measure thread properties. With this scheme, the set of relevant properties from characterization are determined offline, and each property is measured per-thread at runtime using profiling techniques. For example, a property representing a thread's memory access pattern (which is a basis for determining memory aliasing) may be measured through counting and binning that thread's memory accesses, based on address. As each thread is run multiple times, the learner's responsibility is to decrease overhead, by disabling the profiler, when the learner thinks a new measurement's outcome will be the same as old measurements.

Once each property is measured, one way to compute thread-thread affinity may be to compare each property between each pair of threads, per-device. If threads are graph vertices and affinity is shown through undirected edges, this step assigns device-specific weights to each edge. This extends Jian Chen et al.'s[3] work by considering properties (such as memory aliasing) that may be incomparable through simple distance metrics. The scheduling problem is then to reconcile how each thread can only be fit to a single device per run, yet be broken into a different affinity-based cluster for each device. State of the art approaches[4,5] guide this process through programming-level frameworks and implementation alternatives for algorithm kernels. Given per-device affinity clusters, one approach for the proposed lowering scheme may be to schedule each thread based on whatever thread-device mapping minimizes the overall loss in affinity between threads.

My interest in efficient mappings to heterogeneous systems was kindled through my work on reconfigurable computing systems (my research experience essay discusses this work). Last Summer, I worked with Professor John Wawrzynek, and others, to map a Bayesian inference algorithm to several data-parallel accelerators, such as GPUs. Using device inference, we would have coded the algorithm *once*, in a form most appropriate for Bayesian inference. Computationally, Bayesian inference forks into compute-intensive threads that perform read-only operations on disjoint sets of data. Thus, in deciding to map to a GPU (for example), the runtime must *recognize* which threads (a) spin in an inner loop for long periods of time, and (b) do not memory alias with one another.

During my NSF tenure, I want to build and integrate a prototype device inference system into other groups' projects. As I discuss in my personal statement essay, getting feedback on a variety of devices, and the user code that is mapped to those devices, will be very important in developing a robust set of properties to describe both threads and devices. I plan for prototyping to evolve from passively profiling (during characterization) project applications to actively scheduling them (during lowering). Through leveraging current projects, I want to widen the set of devices and applications that are studied and reciprocate what is learned to help improve those projects' use of their systems.

---

[2]T. Karkhanis et al., A First-Order Superscalar Processor Model; *Computer Architecture News, 2004.*

[3]J. Chen et al., Efficient Program Scheduling for Heterogeneous Multi-core Processors; *DAC, 2009.*

[4]J. Ansel et al., PetaBricks: A Language and Compiler for Algorithmic Choice; *PLDI, 2009.*

[5]C. Luk et al., Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping; *MICRO, 2009.*