Data Locality Background

In a perfect world, computers would have an unlimited amount of cheap, fast memory. In this one, small memories are the fastest, while speed and capacity cost money. The field of data locality has been developed in the last half-century to address the problem of how to structure both programs and memory hierarchies in order to minimize time spent retrieving data. In the last decade, the field has grown to include a mature theory that can predict, with high accuracy, the miss ratio experienced by a program when run with any size cache [6]. But there is still more to do. In particular, with the growth of multicore computing, there is a need for some foundation on which to make locality-based decisions. My research focuses on characterizing, predicting and optimizing program locality in the context of this next frontier in computing.

Contributions

I have contributed to several research projects. Program Assisted Cache Management proposed the use of program locality metrics to provide cache hints to architecture, both in single-core [2] and multi-core [1] settings. Example-Based Source-to-Source Translation analyzed intermediate representations of C and Fortran code in the ROSE compiler in order to build a corpus of building blocks for translation between the two languages. Partition-sharing outlines the options available for assigning cache partitions to many programs on multicore machines. And Optimal Footprint Symbiosis outlined a theory to predict shared-cache performance based on per-program metrics [5].

Ongoing Work

Shared-Cache Locality Theory

While we have a sound theory for predicting miss ratios experienced by one program run on a single-core processor, that theory breaks down when two or more programs share a cache (as is quite common in today’s multicore architectures). An individual program has the same data locality each time it is run on the same input, and the locality can in many ways be predicted when it is run on different inputs. However, when multiple programs run on a multi-core machine, a shared cache sees some indeterminate interleaving of the memory accesses from each program.

To first approximation, the overall locality can be assumed to arise from some uniform interleaving of the memory access traces, as calculated from each program’s overall data access rate (this method was used in the CCGrid paper mentioned above). But this approach fails to account for the non-uniformity of data access in real programs. As a toy example, consider two copies of a program that alternates between traversing an array and sleeping. They would interact favorably if they alternated accessing memory, but they could also have heavy interference if their memory-intensive phases happened to coincide.

Unfortunately, it is not in general an option to test and store performance metrics for every possible group of programs. With 4 processors sharing a cache, and 25 programs to schedule, there are 12,650 possible scheduling choices at any one time.
Instead, we would like a theory that can predict shared-cache locality using per-program metrics. I am currently working to extend the single-core locality theory to include access time data that will allow us to characterize and predict cache sharing behavior.

**Partition Sharing**

There are two popular options for sharing an individual resource: partitioning and sharing. In partitioning, agents (e.g. users, CPU’s) will each get a fraction of the total resource dedicated to them. In sharing, agents each have access to the very same resource. Partitioning provides each agent protection against aggressive partners, while sharing prevents resources from going unused when they may be needed. If there are only two agents, they may either share or partition any given resource. However, for three or more agents, partitioning and sharing may both be used. For example, two agents may share a partition, while the third agent has its own partition. We call this type of scheme partition sharing. I have ongoing work on determining optimal and fair partition sharing arrangements that I hope will be useful to operating systems programmers for future hardware that allows dynamic partitioning.

**Affinity Hash**

Data affinity is a measure of temporal locality between different data. The affinity between two objects measures how likely one is to be accessed soon, given that the other was accessed recently. It is very common for data in an associative array to have strong affinity relationships. For example, if I always call Sarah immediately after calling Bob, their entries in my phone book will have high affinity.

Hash maps are pervasive in object-oriented languages such as Python and Ruby, partly because they are used to implement classes. By monitoring the affinity between objects, the hash map implementation can store data with high affinity close-by in memory and improve cache-block-level temporal locality. In addition, programmers can be given the ability to monitor affinity for their own purposes. My colleagues and I are designing, building, and incorporating a hash table with this functionality into the Ruby language.

**Summary**

The field of data locality has grown and matured much in recent years, and this has brought about a surge of new questions and potential applications. How can we adapt the theory to understand and improve the behavior of programs on new microarchitectures? And how can we use locality to build better data structures? My research finds new ways to use data locality and bring it into the multi-core era.
Jacob W. Brock - Research Statement

References


