Memory-Centric Architectures: Why and Perhaps What

Doug Burger and James R. Goodman

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706 USA
{dburger,goodman}@cs.wisc.edu

Relative to computation, the costs of communication in computer systems are growing with each new generation of technology. This trend applies to both communication within the processor chip and communication between the processor and the rest of the system. Communication within the processor is growing more expensive as the RC (resistive-capacitive) delays of the on-chip wires scale poorly compared to the processor clock and transistor switching speeds. Communication off of the processor chip grows more expensive as long memory latencies and limited bandwidth (across the memory bus and the processor package) inhibit performance gains.

Long communication delays between system resources force the partitioning of resources, which should do as much work as possible within each partition, communicating with remote resources infrequently. As communication latencies climb, the extent to which resources must be partitioned will grow correspondingly. We view partitioning as occurring in four stages: computation, instruction fetch, control flow, and memory. The first partitioning to occur on-chip has been functional unit clusters (as in the DEC Alpha 21264 and other proposed architectures). Although the actual computation is partitioned, instruction fetch and control flow remain centralized. The Multiscalar architecture (and subsequent derivations, such as the trace processor) partition the instruction fetching as well, while retaining a single logical flow of control. Chip multiprocessor architectures (such as the Stanford Hydra) take distribution one step further, partitioning the control flow. These partitioning models all assume that the computational core is centralized, isolated from the system memory. The fourth partitioning in this conceptual framework is memory, in which each of the partitioned unit "owns" a fraction of the system's physical memory. It is this level of partitioning that we argue is the right model for the longer-term future, where performance will be dominated by communication overheads.

Distributing processors to regions of memory necessitates partitioning the problem and decomposing the data to the partitioned regions. Both can be hard to do well statically; some codes lend themselves well to one or both, while others are not amenable to static analysis. If the problem partitioning does not match the data decomposition, extremely poor program performance will result. When both problems cannot satisfactorily be addressed statically, we propose to partition the program dynamically based on the given data decomposition. It is this concept that forms the basis of what we call memory-centric architectures.

We have proposed two such architectures; DataScalar and DDT. DataScalar architectures use massively redundant computation to improve communication performance. In a DataScalar architecture, physical memory is divided into distinct regions, each of which is coupled with a processor. All processors execute the same program (asynchronously), performing all of the program's computation redundantly. Furthermore, all communication consists of sends; whenever a processor reads an operand from its local memory, it broadcasts that operand to all other processors. No processor thus ever sends a remote request for data; all non-local data it needs are broadcast by the owners of those data. Thus all communication is one way; remote writes are never sent (since all processors generate all store values). Furthermore, individual processors may run ahead on data dependences found locally, thus effectively prefetching down that dependence chain for the other processors. Our simulation results show 9% to 100% performance improvements on a four-node DataScalar system running SPEC95.

The second memory-centric architecture that we describe here is called DDT, for *Dynamic Data Threads*. In a DDT machine, the memory is distributed among multiple processors, as with a DataScalar architecture, but computation along a local dependence chain occurs uniquely at one node. When a data dependence spans nodes, the source register value is broadcast to all processors, and the intermediate instructions are squashed at all processors except for the owning processor that executed them. Three techniques can benefit dynamic data threads: *computation updating* (described above)*; control updating*, in which a processor ahead of the others sends a point at which the others should resume control flow; and *speculation throttling*, in which processors speculate down multiple paths *only* on data they find locally (inter-processor dependences thus throttle speculation). DDT architectures can benefit from other support, such as static and run-time software support, tagged instructions, and memory system support (allowing a finer-grain naming and migration of data than the granularity of a page).

Increasing communication costs will force microprocessor-based systems to be more and more partitioned. We argue that this partitioning must eventually include the system memory, and that for codes that are hard to analyze statically, the problem partitioning will be done dynamically, based on the data decomposition. We call such architectures *memory-centric*, and describe two (DataScalar and DDT) that are evolutionary first steps in this unconventional direction.