NDP: Network-driven Processing for Chip Multiprocessors

Philo Juang, Kevin Ko, Eric Chi, Julia Chen, Qiang Wu, Li-Shiuan Peh, Margaret Martonosi
Department of Electrical Engineering, Department of Computer Science
Princeton University, Princeton, NJ 08544

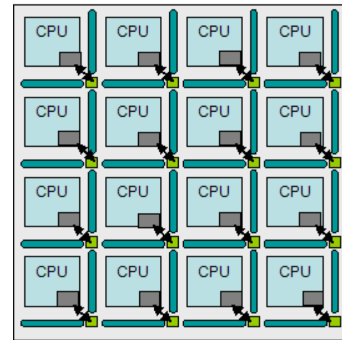
"Unfortunately, no one can be told what the Matrix is. You have to see it for yourself." – Morpheus, The Matrix (1997)

With industry already building chip multiprocessors (CMPs), how can a new CMP possibly be a wild and crazy idea?

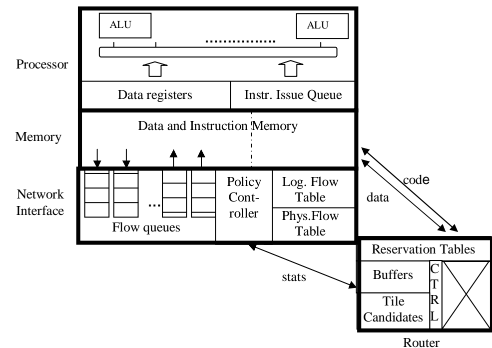
In today's chip multiprocessors where processor cores communicate through an interconnection network, the network has typically been subservient to the individual processors, serving as little more than a data carrier that processors have full control over. *What if, instead of processors controlling the network, the network controls the processor?*

What we propose is a network-driven processor (NDP) where execution is dynamically controlled through the network, or the "Matrix," so to speak. NDP is composed of a battery of tiles, each consisting of a fully independent processor, or a "host," connected to the "Matrix" through a policy controller in the network interface as well as the network router. Collectively, the policy controllers and network routers serve as "Agents" into the network, determining how, when, and where code is parallelized. While the host processor itself believes it is a fully autonomous machine, it is in reality directed by and controlled by the network, which, in turn, is itself regulated by a set of rules encoded in the hardware routers and policy controller state machines.

Why Dynamic Adaptation? Unlike most prior CMP designs that are compiler-driven, NDP enables dynamic, adaptive execution – through run-time parallelization, placement and frequency/voltage scaling. First, as it continues to be difficult to analyze programs thoroughly at compile-time, dynamic adaptation continues to be very important and effective. Second, program behavior often changes with dataset sizes or workload multiprogramming. As such, the desired level of parallelism or the desired degree of locality can change at runtime as well. It makes sense, therefore, to offer low-level hardware and network primitives to support this dynamic control. The third reason to choose such dynamic control concerns the speed versus energy tradeoffs increasingly common in processors today. Adaptive DVFS can help with speed/energy/load balancing. Rather than compile for each eventuality, we must be able to smoothly adjust on-the-fly.



(a) Overall chip architecture



(b) Tile architecture

Why Network Control? Adaptive control requires status information to be passed from tile to tile. Through both intentional sends/receives as well as "eavesdropping", the network is most efficiently poised for an up-to-date view of chip status. Since communication latency is often the primary hurdle for parallel program performance, placing control in the network minimizes the overhead – the means for solving a problem lies close to where the problem originates.

How NDP works. In NDP, the compiler identifies blocks of code that, together with their input and output data, represent a *flow.* Flows are bounded by *parallelism points*, which occur at function boundaries for function-level parallelism (Flows can also represent other types of parallelism such as data-level, instruction-level, etc.) For function-level parallelism, a flow represents each invocation of a function.

While executing a program, when a tile comes across a flow, it is not executed immediately, but instead inserted into a *flow table.* To the program, the flow appears to have executed as normal (and instantaneously). Hardware policy controllers examine entries in the flow table and make a decision – whether to execute the flow locally on the tile, remotely, or wait. The decision is based on inspection of network flow queues as well as chip and tile status information collected by the network.

NDP dynamically manages parallelism in a number of ways. It can *clone* a point, which copies the instruction blocks and divides the input data and output destinations amongst the original and the clone. NDP can also *cleave* at a parallelism point, dividing an instruction block into parts, thus chaining processors together in a pipelined fashion. Conversely, NDP can *merge* processors at a parallelism point to load one processor in order to shut down the others. In addition, the network and policy controllers also set the frequencies of each individual tile.

Research Issues. Many questions remain – How can multiple levels of parallelism be captured as flows? How can we design a stable and effective policy controller that balances power and performance appropriately? How do we architect a network from ground up that is optimized for dynamic execution? Significant flow state needs to be maintained – how can it be distributed across the network?

Ultimately, the success of NDP depends on the "rules" of the "Matrix" – the protocols governing the network routers and policy controllers, as well as the "Matrix" infrastructure – NDP's chip architecture. As Architects, we seek to create an ideal NDP "Matrix" that drives towards optimal power-performance for parallel programs.