Scheduled Dataflow Architecture: A Synchronous Execution Paradigm For Dataflow*

Hyong-Shik Kim   Krishna Kavi
University of Alabama in Huntsville

Ali Hurson
Pennsylvania State University

Even though dataflow model and architectures have been studied for more than two decades and held the promise of an elegant execution paradigm with the ability to exploit inherent parallelism in applications, the actual implementations have failed to deliver the promised performance. We feel that the primary limitations of the pure dataflow model that prevented commercially viable implementations are:

- too fine-grained (instruction-level) parallel processing,
- difficulty in exploiting memory hierarchies and registers in instruction execution, and
- asynchronous triggering of instructions.

Many researchers have addressed the first two limitations of dataflow architectures – including macro-dataflow approaches and cache support for dataflow model. Often dataflow properties are applied at thread-level parallelism, while individual threads still consists of conventional control-flow instructions.

Our idea is to retain functional properties of the dataflow model even within a thread. Individual instructions take operands as dataflow instructions do, and produce the result which will be consumed by consequent instructions. Instead of passing the result directly to another instructions, it will be put in a designated frame slot. For that purpose, each instruction is associated with a frame slot, where all its operands are stored. In other words, we deviate from the asynchronous triggering of dataflow instructions within a thread, and schedule pre-ordered instructions of a thread in synchronous fashion. Every operand of each instruction is guaranteed to be made available before execution, when a thread is generated.

By introducing the synchronous property into dataflow paradigm, we expect many advantages which could not be found otherwise:

- no side effects,
- no WAW and WAR dependencies between pipeline stages,
- easy register renaming and reuse, and
- no need for dynamic instruction reordering at execution time due to non-blocking nature of our threads.

In addition, we also make use of a separate functional unit which permits decoupling of memory accesses from *execution pipeline*(EP). The other unit which will be called *synchronization processor*(SP) not only schedules the next available thread, but also prepares operands in registers on behalf of EP (*pre-load*) and writes back some values into a frame after the execution (*post-store*). This decoupling memory fully hides memory latencies, when sufficient fine-grained parallelism is available, without complex hardware techniques.

*This research is funded in part by NSF grants, MIP-9796310 and MIP-9622836.