Exploring Parallel Out-of-order Re-execution

D. Morano, D.R. Kaeli
Northeastern University
dmorano, kaeli@ece.neu.edu

8th January 2004

Current high performance superscalar processors have widely employed instruction level parallelism (ILP) to achieve their high single-threaded performance levels. The primary mechanisms used in these microarchitectures for implementing their ILP extraction have been control flow prediction, out-of- order instruction issue and speculative execution, multiple parallel function units, and the use of a reorder buffer for dependency management and for instruction squash. Large instruction windows have been important for ILP extraction and instruction windows of from about 100 to 200 instructions are in use with larger ones still likely.

Although control flow prediction and speculative execution have been paramount towards improving ILP extraction from serial programs, various proposals for the use of value prediction have also appeared to offer promise of even more ILP extraction. Many value prediction proposals have focused on memory load addresses or memory load values. A more aggressive value prediction proposal would include not just the prediction of branch target addresses (direct or indirect) or memory operand addresses (and in some cases memory operand values – like the Pentium-4 microarchitecture), but also register operand data values – indeed all operand values. Although value prediction seems to have been promising, using it on a large scale adds a substantial additional logic burden to the control portion of any microarchitecture that employs the technique. The existing complexity of implementing value prediction has prevented its investigation into larger and more parallel microarchitectural designs using it.

We present a microarchitecture that features the ability to re-execute instructions as a primary and expected event. In our microarchitecture, instructions are dispatched in decoded form to microarchitectural structures that closely resemble reservation stations or instruction issue window slots. However, rather than having the instruction vacate the issue station when it is transferred to an execution pipeline, in our microarchitecture the instruction remains in the issue station structure until it is retired (either being committed or abandoned). The instruction will request an execution resource when needed but will wait for result operands from execution to return to the station structure rather than proceeding to update something like a reorder buffer or the register file (either architected or speculative). The instruction station structure also holds all input operands and output operands for the instruction along with other state information that guides the instruction through repeated executions until it is retired. Generally instructions will request execution when all of their input operands become available whether they may be speculative or not (similar to the classic reservation station) but they can also request additional re-executions when some condition changes that warrants or advocates a new execution. Conditions that indicate the need for a re-execution include a new input operand arriving from a previous instruction (which may have also re-executed providing a new result), or a new input operand prediction being made.

Using this microarchitectural approach, instructions can be executed and re-executed with a low overhead. This low overhead facilitates large-scale parallel use of value prediction for input operands that has not yet been well explored in other proposed microarchitectures. Instructions that re-execute do not have to reflow through any of the: fetch, decode, rename, or dispatch stages of the machine. Although this approach is not entirely new (witness Pentium-4 with load instructions) we have elevated the process of speculative re-execution to a level not seen in any conventional microarchitecture or proposed microarchitectures.

Our approach to program dependency ordering is realized by tagging all instructions that get dispatched to issue stations with tags that reflect the dynamic program order. These tags essentially represent a time stamp of a sort to place the instruction into its proper dynamic program order as compared

with all other instructions being executed. We further tag all program operands, whether they be register operands or memory operands. The program ordering tags are just small integers that have a number of bits that is approximately log-base-two the number of instructions allowed in flight in any given machine implementation. The value of a tag is that is assigned to a dispatched instruction (effectively assigned to the issue slot itself) is successively higher for each new instruction. Generally, the instruction that is oldest in program ordered time and nearest to program commitment is assigned a tag value at or near zero, while those instructions (both non-speculative and speculative) that are younger and more speculative are assigned successively higher values. These tags are periodically uniformly decremented, at the occasion of instruction commitment, throughout the whole of the execution core in order to prevent overflow related issues. Although a register file is used for storage of the architected registers, no renaming registers are needed in this microarchitecture. All instruction operands are effectively automatically and uniquely named by the association of the operand with its program ordering tag. In this way, all false operand dependencies are eliminated without having to provide explicit register renaming resources, the number of which could have been otherwise limiting for instruction issue if there was not a sufficient number available as compared with the number of operands in flight in the execution core.

The execution resources of this microarchitecture are similar to other machines. We employ several pipelined function units of varying pipeline depth depending on the execution latency for the different units. Unlike conventional machines, execution results are returned to the issue station from which the originating decoded instruction operation came from. Use of a particular function unit is available to any set or group (possibly including all) of the instruction issue stations. Of course, since each issue station may be occupied by any type of instruction, at least one of each type of function unit is available to each issue station for use, but different forms of function-unit multiplexing can be explored.

Although all instructions are dispatched to issue stations in-order, once all instructions and operands are tagged using our technique, all instructions are allowed to issue and execute in any order and with any suitable input operands regardless of whether the input operands are correct or not. In those circumstances when an input operand is not the final committed program value, the instruction will dynamically determine through operand snooping that it will need to re-execute at least one additional time (with the correct committed input operand value) before it can be committed (if it is even going to be committed). All instructions are allowed to execute simultaneously as available execution function unit resources permit. Input operands to instructions later in program-ordered time snoop for output operands from previous instructions. In this way, all dependencies are determined dynamically and in a distributed manner. This distributed operand dependency mechanism facilitates easier expansion of the machine size (numbers of various units) while still allowing for the proper control to achieve the maximum amount value prediction, all in parallel. Executions or re-executions only need occur when input operand values change from their previous values. Eventually, all correct operands are snarfed and if the oldest dispatched instructions have executed at least once, commitment of those instructions can occur. Any number of instructions (at the granularity of a single instruction – unlike some other microarchitectures) is allowed to commit in each clock period. This microarchitecture is also suitable for use in implementing any of the existing instruction sets or any foreseeable new ones also (including those that have architected predication). Using this new microarchitecture with current sized (around 200 instructions) or larger instruction windows, it is expected that a higher amount of ILP can be extracted from serial integer programs.

Our research methodology is to create a new cycle accurate simulator that will allow for various characterizations of the several configurable machine features. Configurable elements include such things as the numbers of various units (issue stations, function units and types, et cetera) but also the amount and configuration of various operand exchange bussing. The simulator will have the ability to execute standard benchmark programs using an existing instruction set architecture and compiled using standard (including vendor supplied) compilers and linkers.