**Summary (Part 2)**

**Multiscalar Processors**

Mrinal Nath (ID: 3307043)

February 2, 2005

Did this paper address an important issue? Explain.

The basic problem being tackled in this paper is how to extract more ILP from programs that are inherently sequential. This is an important issue. Conventional superscalar processors have a limited instruction window size, due to the limited branch prediction accuracy (if the window spans over too many - 5 or 6 - predicted branches, then the instructions in that window are not very likely to be along the correct execution path). Conventional processors have to examine all the instructions in the window to determine which ones can be issued in parallel. The logic required to issue multiple instructions in one clock expands quadratically with the number of instructions issued simultaneously. Issuing loads and stores in a conventional superscalar (with OOO execution) requires a lot of hardware to check for any violations.

Also, as transistor density on a single chip is increasing, wire delays and logic complexity are becoming the limiting factors to increasing clock speeds of the design. Such high density processors will be easier to design if they can be composed of small, fast elements (CMP).

This paper proposes the Multiscalar paradigm, which provides quite acceptable solutions for all these problems.

Are the proposed approaches valid? Describe its strength and weakness.

The paper uses both hardware (microarchitecture) and software (compiler) techniques to alleviate/reduce the above mentioned problems.

**Strengths:**

* The proposed architecture is somewhat like a CMP architecture. This is good since it helps in designing processors with higher transistor densities.
* The proposed techniques depend heavily on the compiler to split a program into tasks and to give other hints to the microarchitecture while it is executing the program (forward and stop bits, create mask, task descriptor, etc.). This shifts a lot of complexity from the hardware to the software, and allows the hardware (Processing Units or PUs) to be smaller, simpler and faster.
* An important strength of the proposed architecture is that the instruction window can be effectively very large, while avoiding the need to examine every instruction in the window. Only a subset of instructions in each PU are examined at a time by that individual PU.
* Each PU issues instructions independently. So the issue logic complexity in each PU is limited. Since multiple PUs are issuing instructions in parallel, the IPC can be really high.
* The loads and stores in each PU can be issued independent of the other PUs. (A mechanism is required to verify that inter-task dependencies are not violated). This allows each PU to execute instructions at a more rapid rate, since it does not have to wait for addresses in other PUs to be resolved.

**Weaknesses**

* The sequencer and the ARB are centralized structures. Their bandwidth limitation can be a perfor-
mance bottleneck. Also, writing back of all the buffered speculative memory updates from the ARB into the data cache when the task commits can cause bursty traffic, and delay the allocation of a new task to the PUs.

* The new architecture will require a recompilation of the existing code base. Fortunately, the authors have proposed the generation of Multiscalar binaries from existing binaries. Also, some new instructions (like \textit{release}) have to be added to the base ISA for the Multiscalar programs to work properly.

* The Multiscalar architecture depends heavily on the compiler to do a good job of splitting the program into tasks, which are of roughly equal size and are highly likely to be independent. If the compiler can not do a good job, then the architecture will not be able to perform well. This may happen in case of some 'badly written' programs, and may require a re-writing of the source code.

**Do the results support the conclusions? Explain.**

Yes, the results show a significant speed-ups for most of the benchmark programs, as a result of the increased number of instructions being executed in parallel. It seems that the Multiscalar architecture performs really well if the compiler can find enough loop-level parallelism. However, for the programs in which the execution is evenly distributed throughout the program and there are a lot of data dependencies among successive basic blocks, the speed up may be very small (or there may even be a slowdown) due to a large number of squashes. However, in some cases, the compiler needed manual intervention (\textit{espresso}) or the program was restructured (\textit{sc}). But overall the speed-up results are quite impressive.

**Describe the potential future works.**

One of the problems with the proposed Multiscalar architecture is that the ARB is centralized. Decentralizing this important block can be useful.

If a task violates a data dependency, it and all its following tasks are squashed. This can reduce performance. Only those tasks that were dependent on the wrong data value used should be squashed, so that useful work done in other independent tasks is not lost. However, such a scheme may require complex control logic.

This architecture can be extended to exploit thread- and process-level parallelism, so that if a particular thread or process is not able to utilize all the PUs, then multiple threads or processes can be used to maximize the utilization.