CMP SCI 635: Modern Computer Architecture

Simultaneous Multithreading: Maximizing On-Chip Parallelism

Name: Kunjal Panchal | Date: 8th Oct, 2019

Student ID: 32126469 | Paper: 10 – Multithreading [Tullsen95]

Strengths:

1. As shown from the experiments of SMT vs single chip MP, SMT is outperforming MP in every case: number of functional units, bandwidth, hardware contexts. That satisfactorily concludes that for most real-life applications, SMTs will perform better, utilization-wise. Even with fewer threads and granularity changes. It is cost effective to add hardware contexts and wide bandwidth than to add one more processor which would just be ideal if not maximum number of applications are running simultaneously.

2. Hyperthreading seems perfect fit for graphics heavy, rendering and streaming applications. But only when it is one of tasks running, and it, by no means, is the only prioritized one for the user. Otherwise, there are always GPUs having multiple threads that can run on the same resources but contrary to CPU those threads will not be really independent.

3. SMT covers stalls in one thread with instructions from the other. As long as SMT covers more stalls than it causes (mostly cache eviction and memory bandwidth issues), it is a net performance gain, even though each thread is running slower as a result of splitting the core between both threads.

Weaknesses:

1. As the author says, they might have been optimistic in guessing the number of pipeline stages required for instruction issue and also in, guessing the data cache access time for the shared cache.

2. It still puts burden on compilers, OS and programmers to check if the simultaneous multi-threading, for the given scenario, won't create performance bottlenecks while ran with some other tasks, and if the programmer, through software, switches off the SMT mode or adds some logic to be in high priority always, OS cannot intervene in that case.

Questions/Assertions:

1. How does the fine-grained multithreaded processor switch context without any overhead?

2. How will multiflow trace scheduling compiler take care of the case where two different programs, each having an instruction which is dependence-free from all other instructions in the same program, but are using the same resources as the instruction from the other program? All the compiler can do is to declare the instruction can go out of order for a pipeline, but when context switch occurs and now hyperthreading comes into play, what shared resource that other program's instruction use?

3. One solution to decrease the wasted issue slots because of TLB misses is to increase the TLB size, wouldn't this just take away time from misses but increase the lookup time? Is looking up for an existing entry in a large table better than getting that entry from one level low in the hierarchy?

4. How does data caches handle protection violations in multithreaded shared cache environment? [Intel's hyperthreading in NetBurst based processors has a vulnerability through which it is possible for one application to steal a cryptographic key from another application running in the same processor by monitoring its cache use.]

5. From what I gathered, if you are running anything that's able to full saturate a core, the hyper thread will not have any time. Meaning that, when programs expect to be diving out work to 8 cores, but only 4 do work quickly enough. There must be some instances where turning off hyperthreading provides a boost, because threads "bounce around" less. Even if this edge case is very less likely to occur, is there ever an issue like: the two threads both issuing a lot of instructions, but fighting over cache by evicting each other's data. And hence, creating bottleneck?

6. Is the clock speed an important factor in SMT based cores?

7. Am I right in assuming that: If the application aims to be instantly responsive for the user, it won't have many space for SMT to do its magic. If an application can be "turned on" and left alone processing, there's likely gains from SMT.