An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Processors

Matthew Curtis-Maury, Xiaoning Ding, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos

The College of William & Mary

Content

Motivation

- CMPs and SMTs are gaining popularity
  - SMTs in high-end and mainstream computers
    - Intel Xeon HT
  - CMPs beginning to see same trend
    - Intel Pentium-D
  - Combined approach showing promise
    - IBM Power5 and Intel Pentium-D Extreme Edition
- Given this popularity, evaluation of codes parallelized with OpenMP timely and necessary

William & Mary

Three Goals

- Compare Multiprocessors of CMPs and SMTs
  - Low-level comparison (hardware counters)
  - High-level comparison (execution time)
- Locate architectural bottlenecks on each
- Find ways to improve OpenMP for these architectures without modifying interface
  - Awareness of underlying architecture

William & Mary

Content

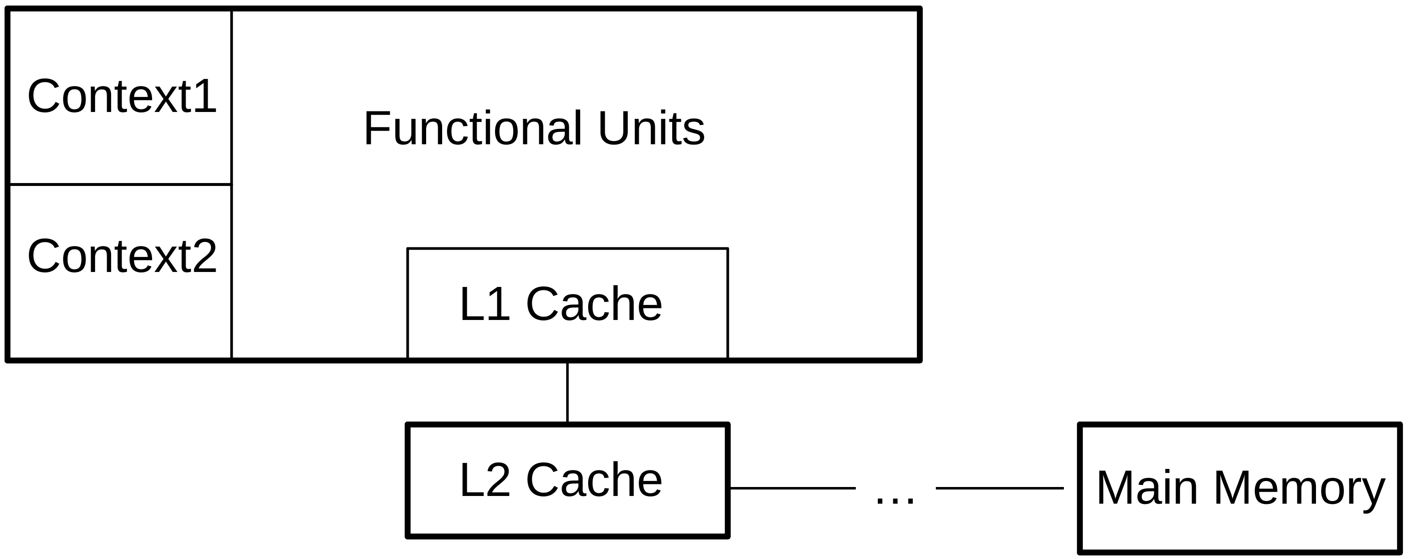
William & Mary

Multithreaded and Multicore Processors

- Execute multiple threads on single chip
- Resource replication within processor
- Improved cost/performance ratio
    - Minimal increases in architectural complexity provide significant increases in performance

Simultaneous Multithreading

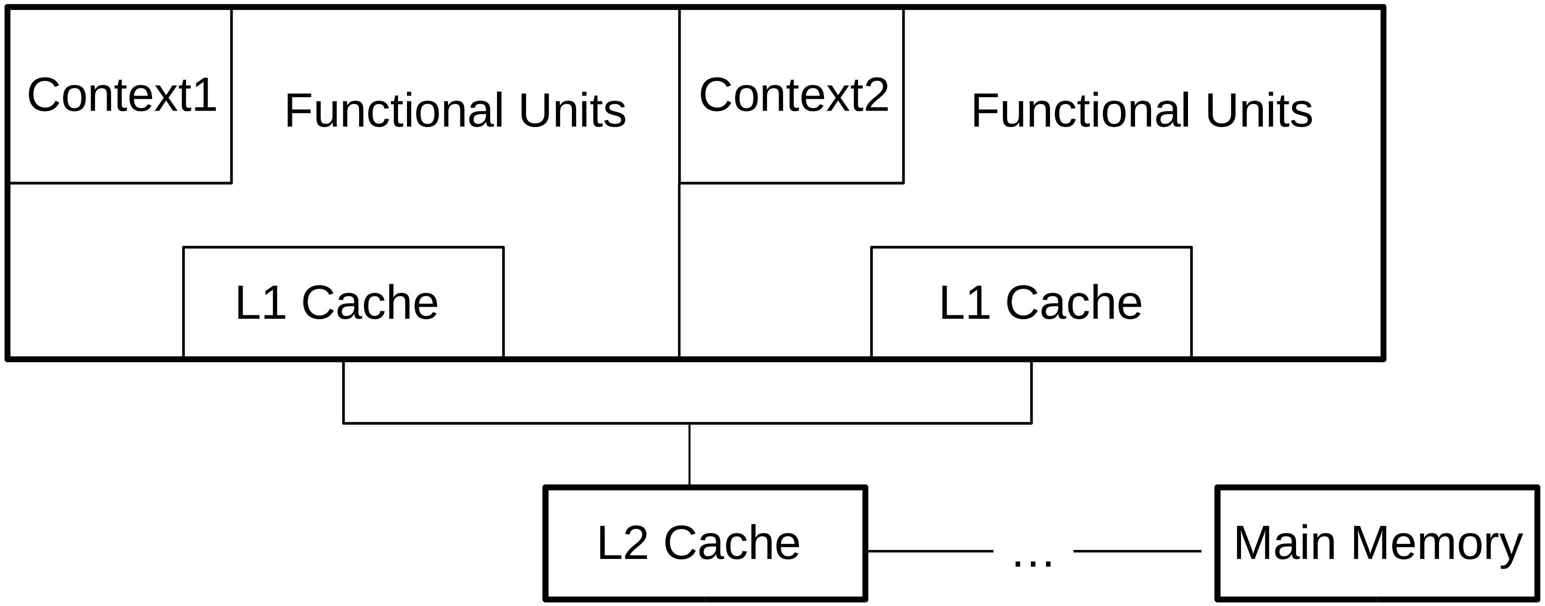
- Minimal resource replication
- Provides instructions to overlap memory latency
- Separate threads exploit idle resources



William & Mary

Chip Multiprocessing

- Much larger degree of resource replication
  - Two complete processing cores on each chip
  - Outer levels of cache and external interface are shared
- Greatly reduced resource contention compared to SMT



William & Mary

Content

William & Mary

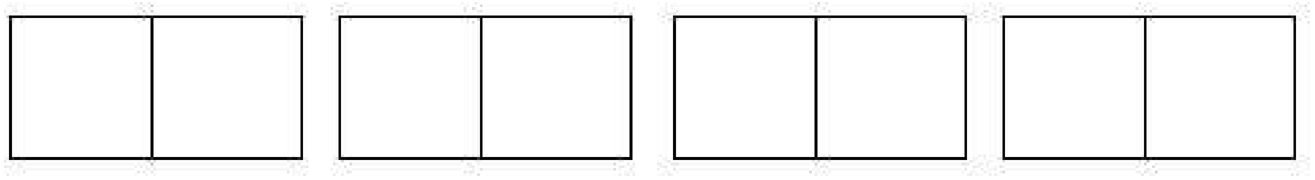
Experimental Methodology

- Real 4-way server based on Intel's HT processors
  - Representative of SMT class of architectures
  - 2 execution contexts per chip
  - Shared execution units, cache hierarchy, and DTLB
- Simulated 4-way CMP-based multiprocessor
  - Used the Simics simulation environment (full system)
  - 2 execution cores per chip
  - Configured to be similar to SMT machine (cache configuration)
    - 8K data L1, 256K L2, 512K L2, 64 entry TLB, 1GB main memory
  - Private L1 and DTLB per core doubles effective space
  - Shared L2 and L3 caches

William & Mary

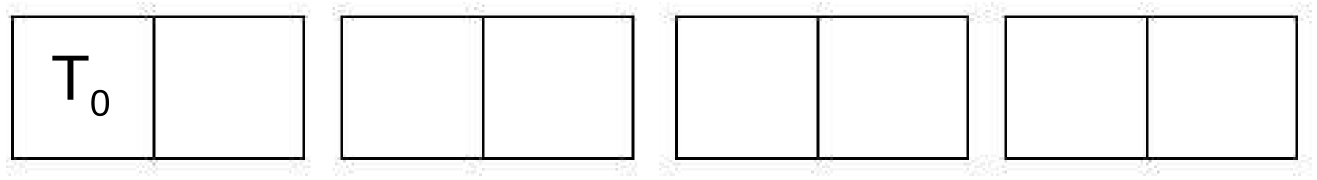
Benchmarks

- We used the NAS Parallel Benchmark Suite
  - OpenMP version
  - Class A
- Ran 1, 2, 4, and 8 threads
  - Bound to 1, 2, and 4 processors
    - 1 and 2 contexts per processor



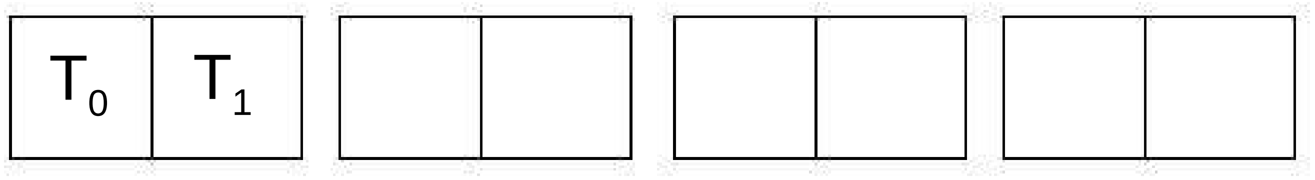
Benchmarks

- We used the NAS Parallel Benchmark Suite
  - OpenMP version
  - Class A
- Ran 1, 2, 4, and 8 threads
  - Bound to 1, 2, and 4 processors
    - 1 and 2 contexts per processor



Benchmarks

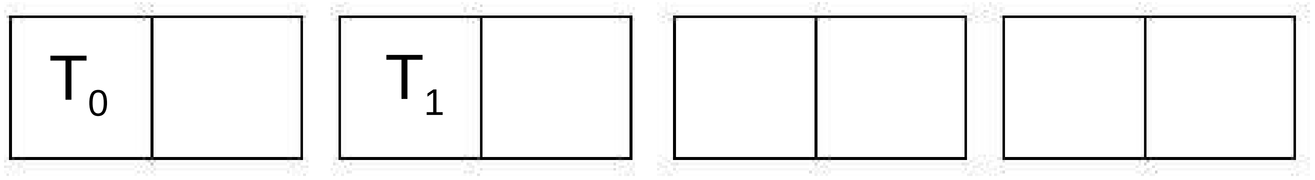
- We used the NAS Parallel Benchmark Suite
  - OpenMP version
  - Class A
- Ran 1, 2, 4, and 8 threads
  - Bound to 1, 2, and 4 processors
    - 1 and 2 contexts per processor



William & Mary

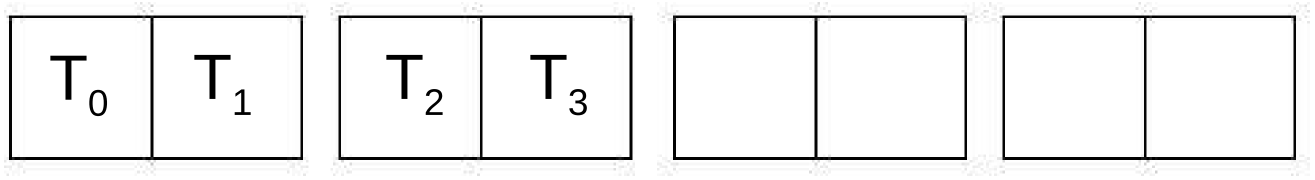
Benchmarks

- We used the NAS Parallel Benchmark Suite
  - OpenMP version
  - Class A
- Ran 1, 2, 4, and 8 threads
  - Bound to 1, 2, and 4 processors
    - 1 and 2 contexts per processor



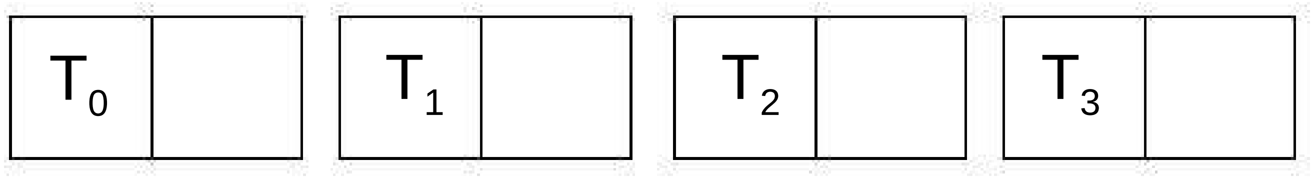
Benchmarks

- We used the NAS Parallel Benchmark Suite
  - OpenMP version
  - Class A
- Ran 1, 2, 4, and 8 threads
  - Bound to 1, 2, and 4 processors
    - 1 and 2 contexts per processor



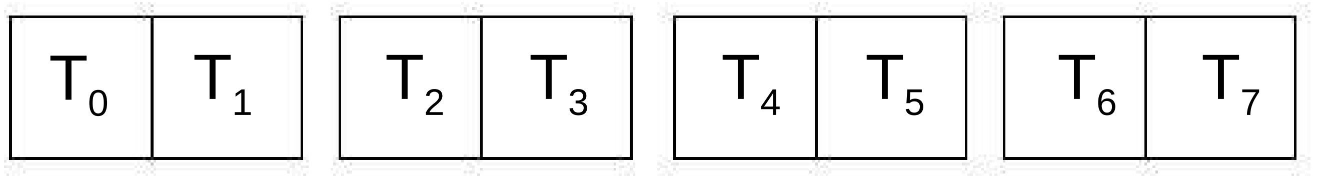
Benchmarks

- We used the NAS Parallel Benchmark Suite
  - OpenMP version
  - Class A
- Ran 1, 2, 4, and 8 threads
  - Bound to 1, 2, and 4 processors
    - 1 and 2 contexts per processor



Benchmarks

- We used the NAS Parallel Benchmark Suite
  - OpenMP version
  - Class A
- Ran 1, 2, 4, and 8 threads
  - Bound to 1, 2, and 4 processors
    - 1 and 2 contexts per processor



Benchmarks, cont.

- On SMT machine, ran benchmarks to completion
  - Collected HW statistics with VTune
- Simulator introduces average of 7000-fold slowdown on execution for CMP
  - Ran same data set as on SMT
  - Ran only 3 iterations of outermost loop, discarding first for cache warm-up
  - Simics simulator directly provides HW statistics

William & Mary

Content

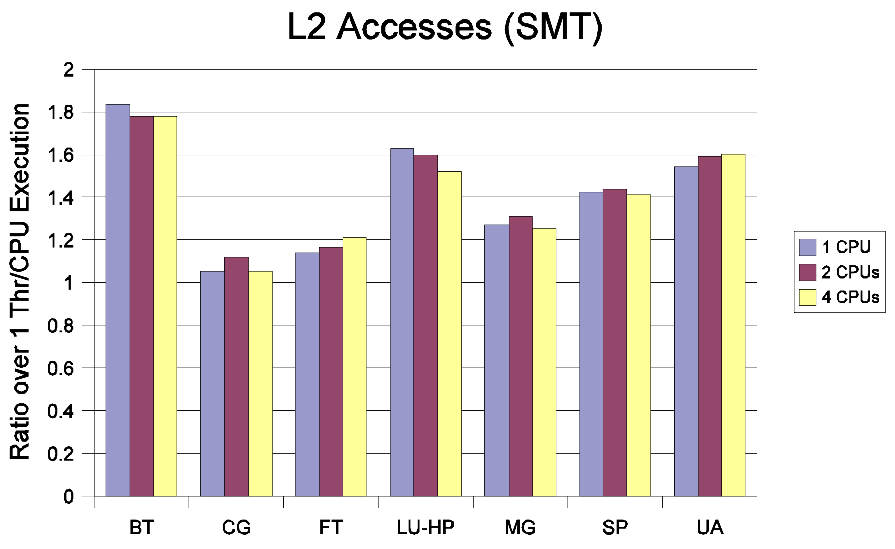
William & Mary

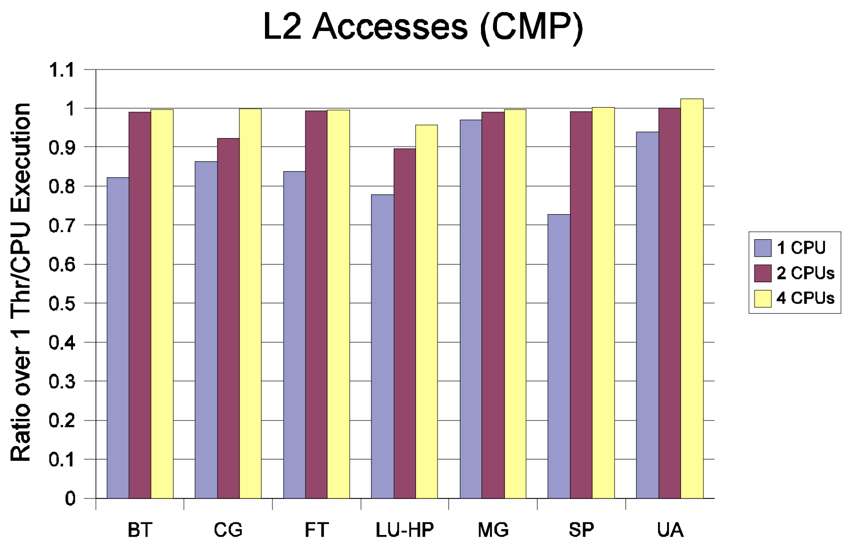
Hardware Statistics Collected

- Monitored direct metrics…
  - Wall clock time, number of instructions, number of L2 and L3 references and misses, number of stall cycles, number of data TLB misses, and number of bus transactions
- …and derived metrics
  - Cycles per instruction and L2 and L3 miss rates
- Due to time and space limitations, we present:
  - L2 references, L2 miss rates, DTLB misses, stall cycles, and execution time
    - Most impact on performance
    - Provide insight into performance

William & Mary

L2 References

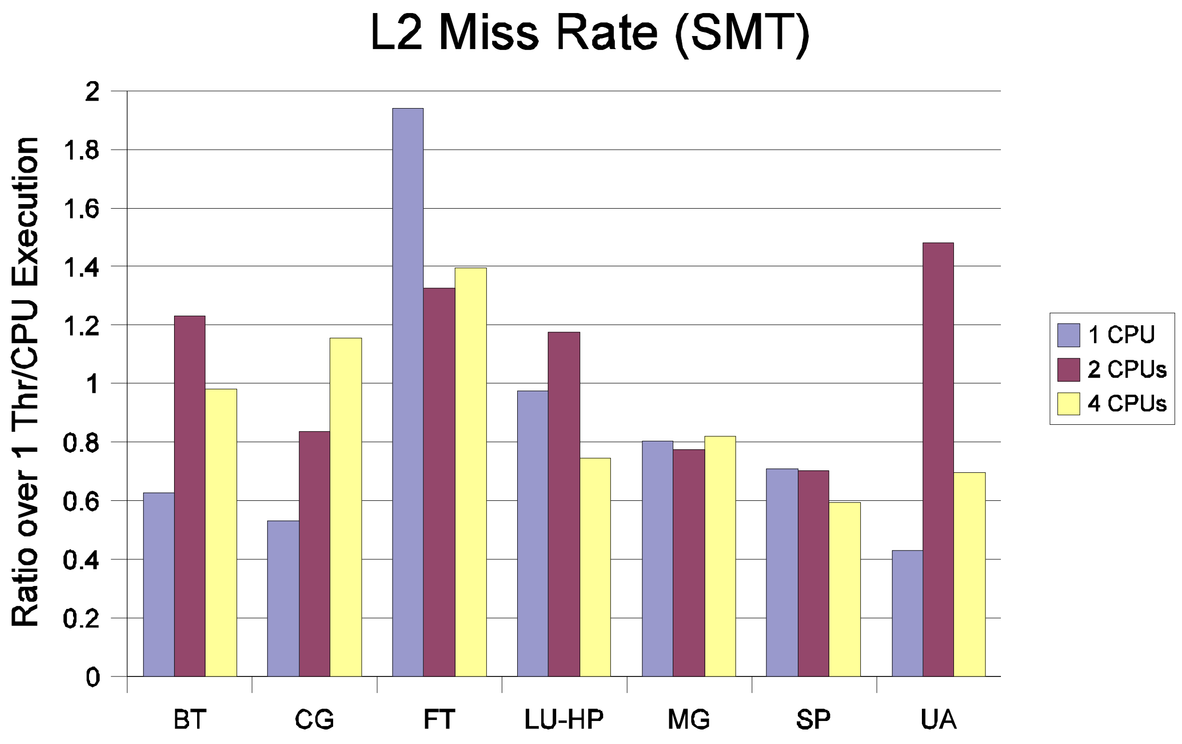




On SMT, two threads executing causes L2 references to go up by 42%

On CMP, running two threads causes L2 references to go down by 37%

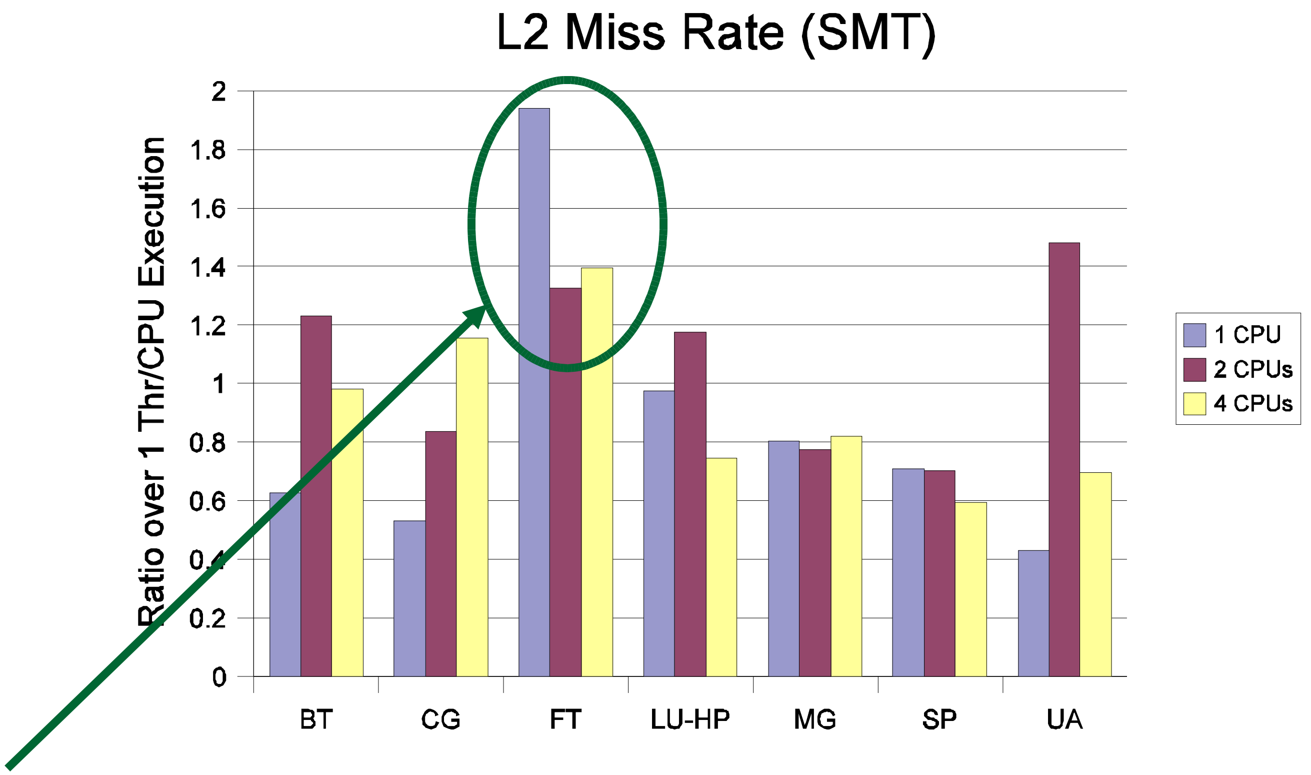
L2 Miss Rate SMT



L2 miss rate highly dependent upon application characteristics

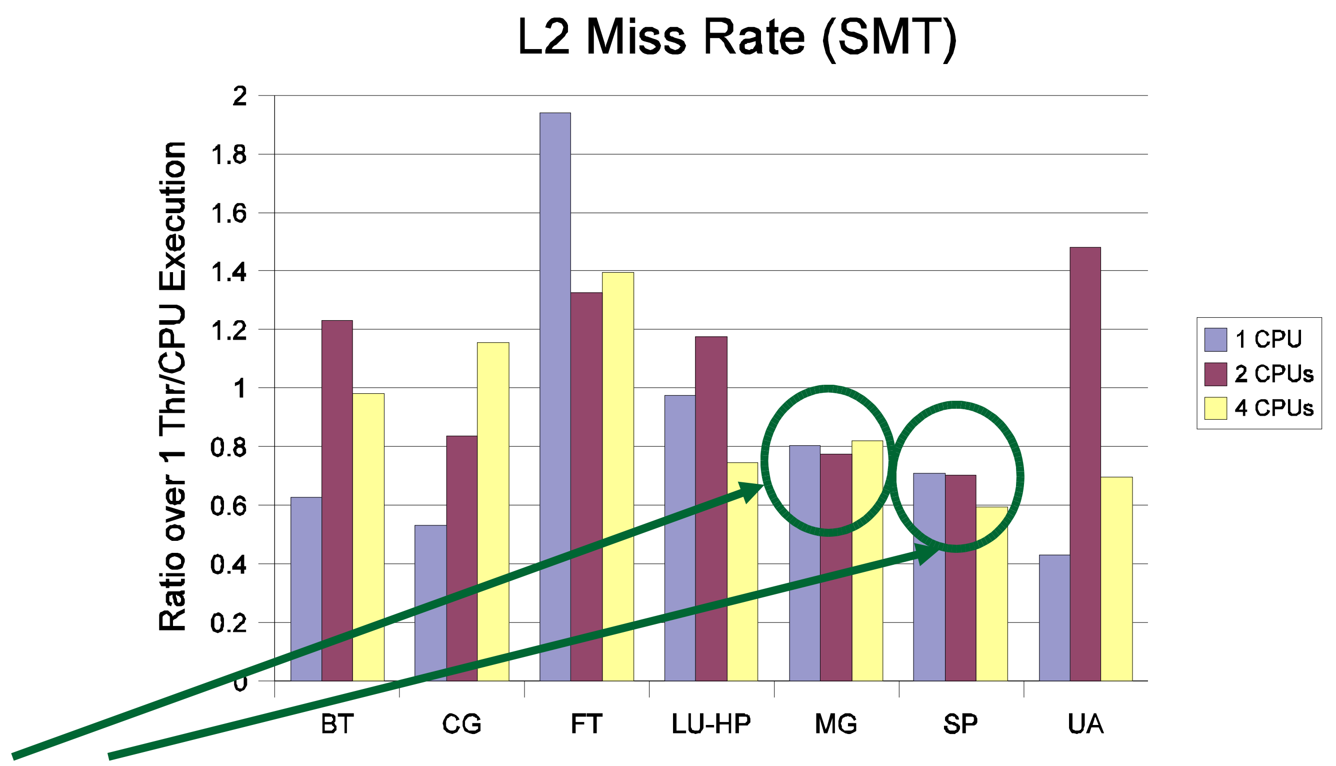
William & Mary

L2 Miss Rate SMT



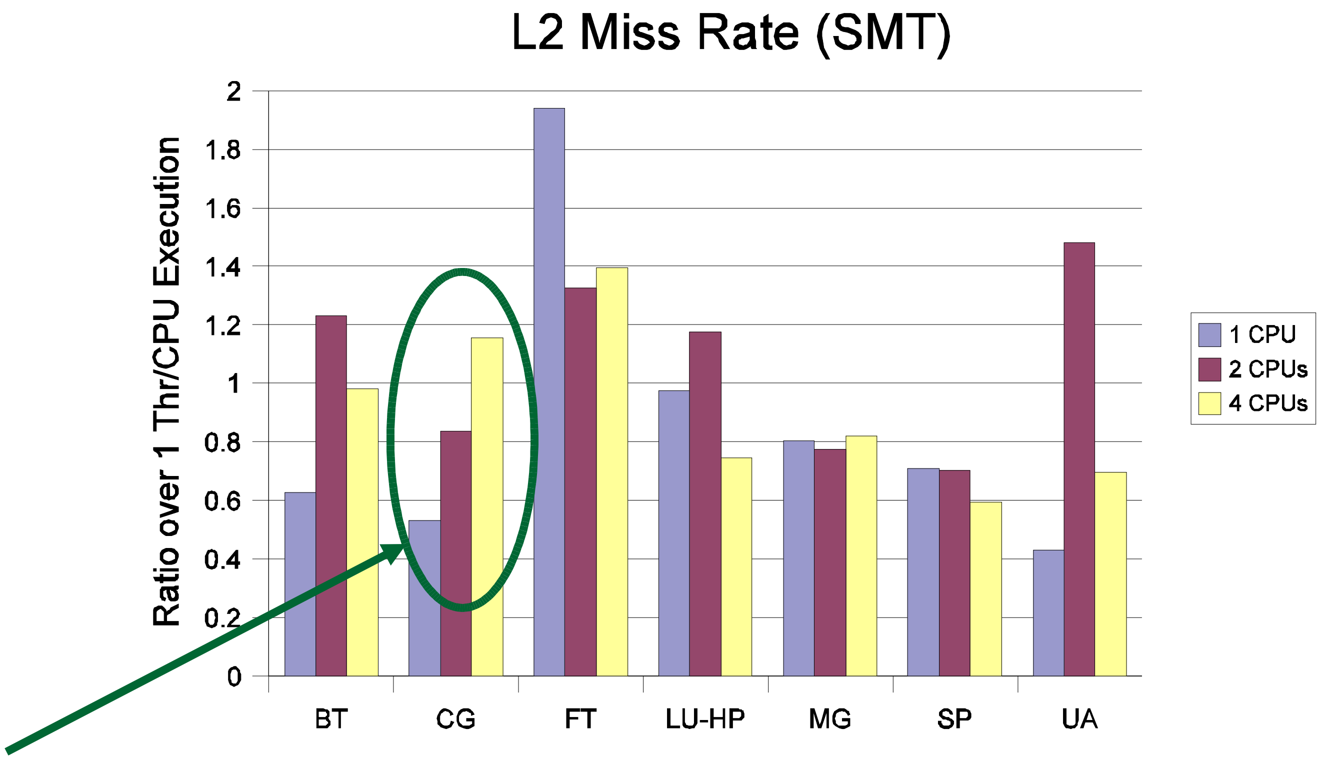
If working sets of both threads do not fit into shared cache, L2 miss rate increases

L2 Miss Rate SMT



On the other hand, applications can benefit from data sharing in the shared cache

L2 Miss Rate SMT

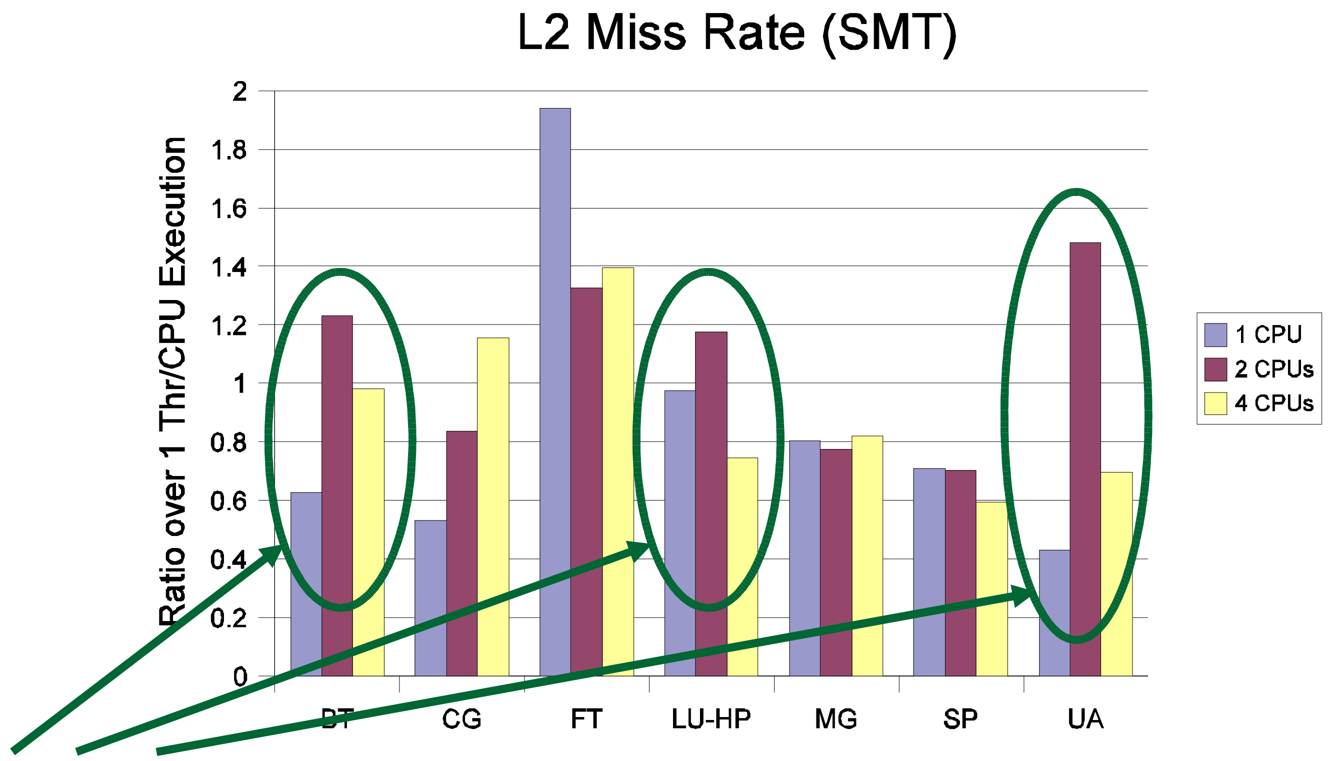


CG has a high degree of data sharing which is good with one processor but has negative consequences with more processors

- Inter-processor data sharing results in cache line invalidations

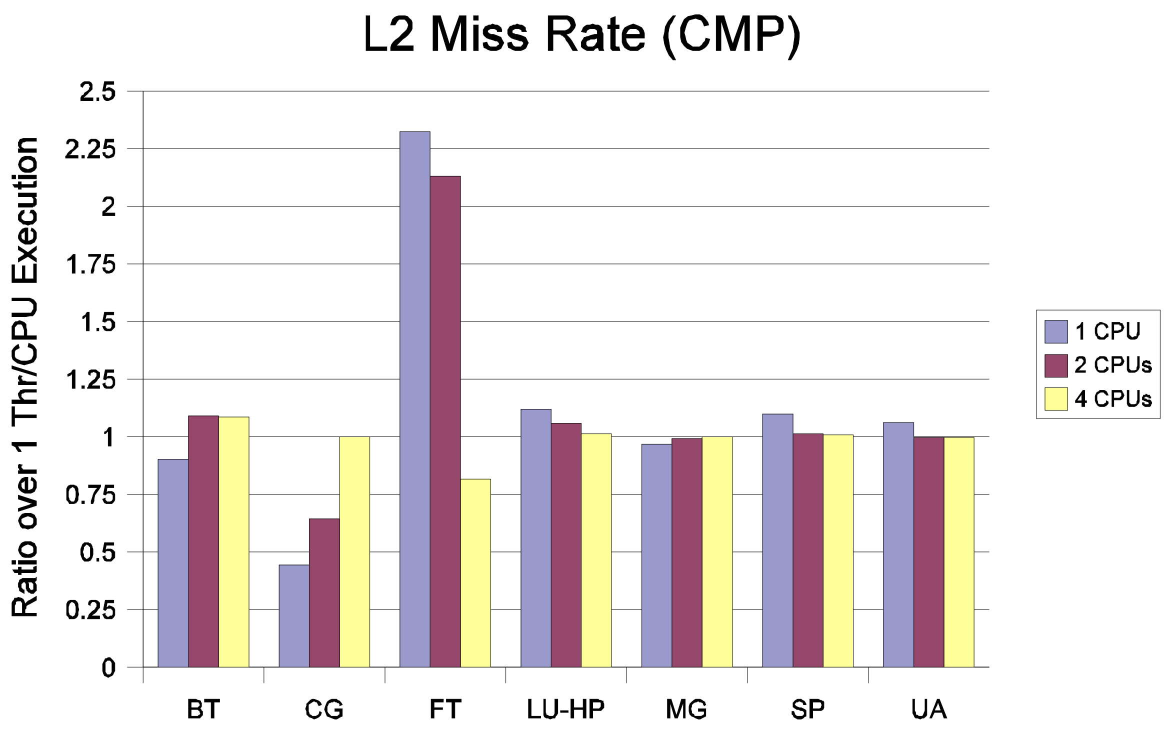
William & Mary

L2 Miss Rate SMT



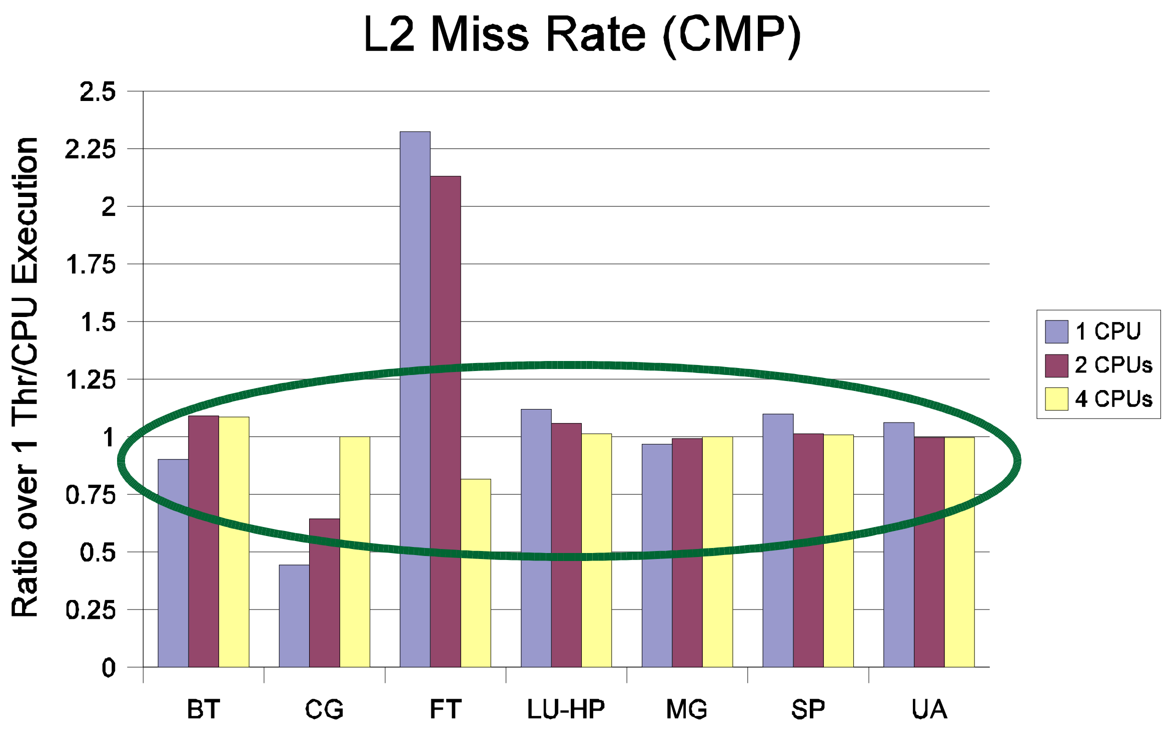
Tradeoffs between sharing in the L2 of one processor and increased cumulative L2 space from multiple processors

L2 Miss Rate CMP



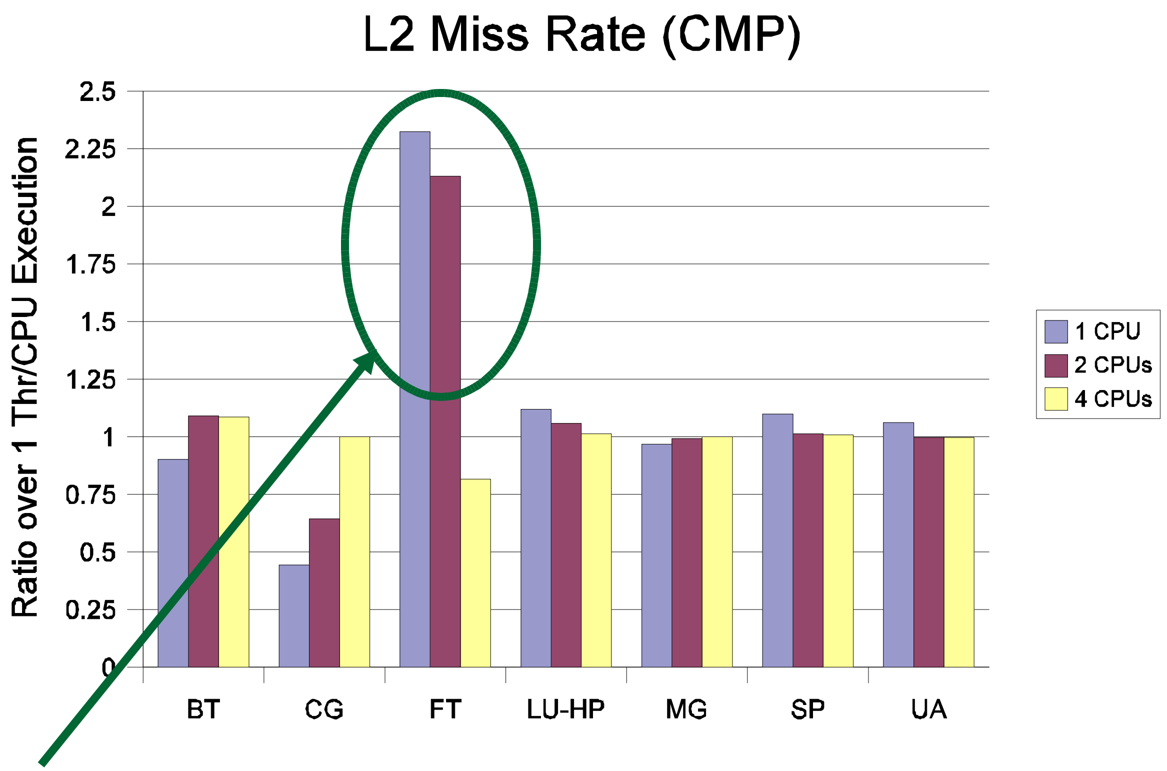
L2 miss rate much more stable on the CMP processors

L2 Miss Rate CMP



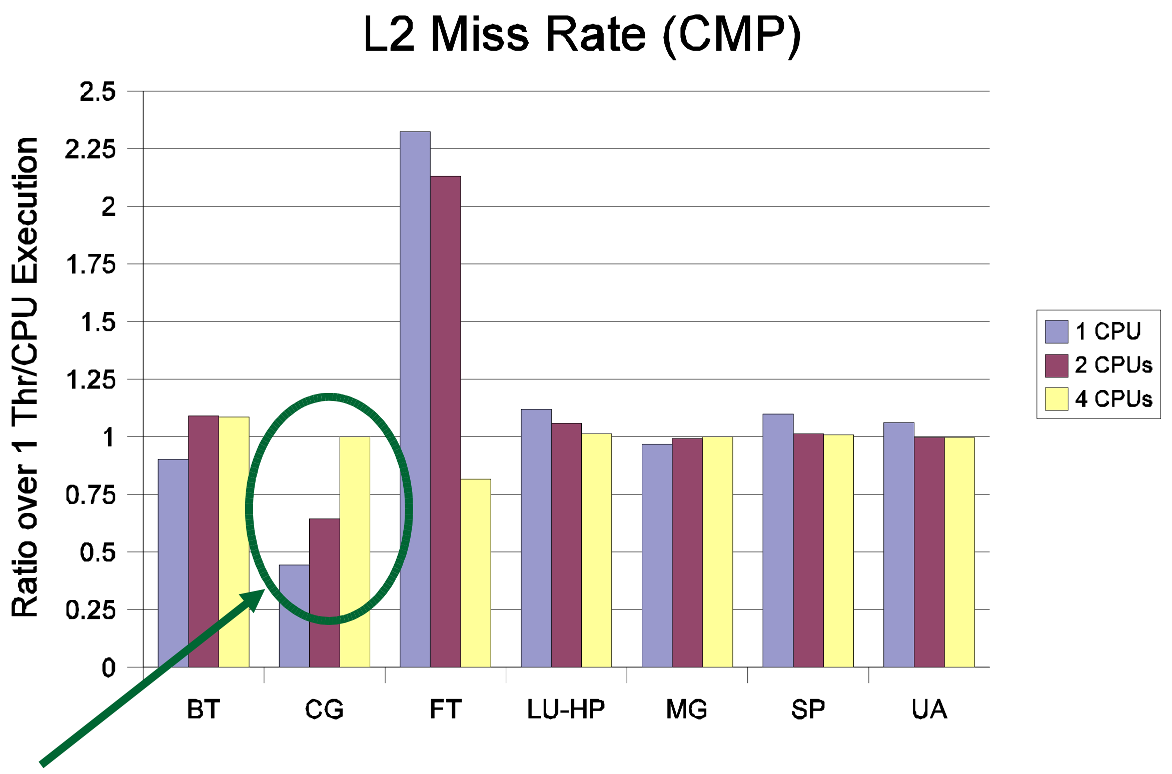
L2 miss rate generally uncorrelated to number of threads per processor

L2 Miss Rate CMP



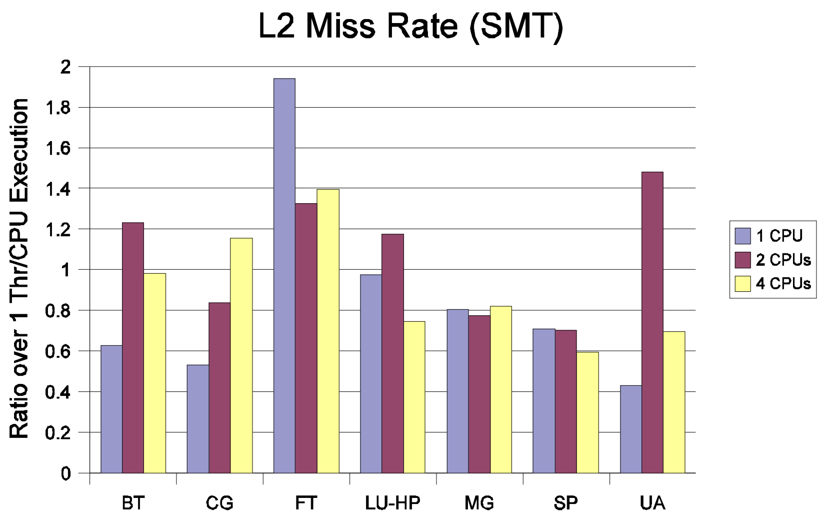
The large working set of FT is still a problem for 1 and 2 processors

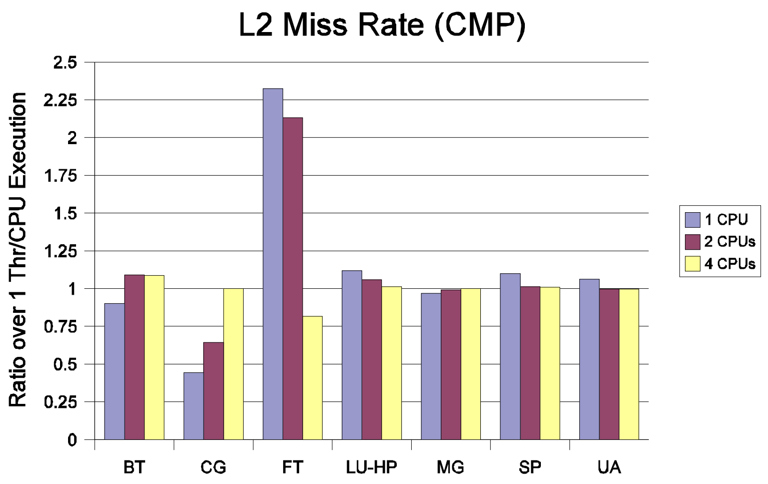
L2 Miss Rate CMP



CG retains the property observed on SMT as well

L2 Miss Rate Comparison





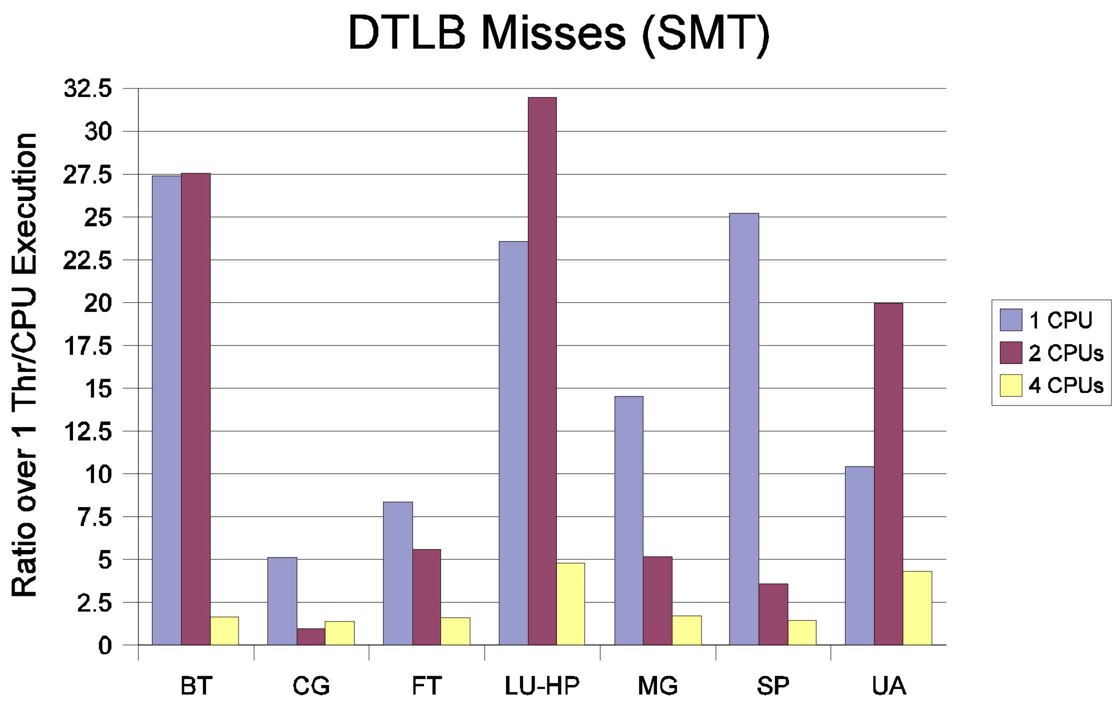
More potential for L2 data sharing on SMT, with shared L1

- Private L1s can reduce L2 sharing, less L2 accesses

On CMP, L2 not as affected by executing two threads per processor

William & Mary

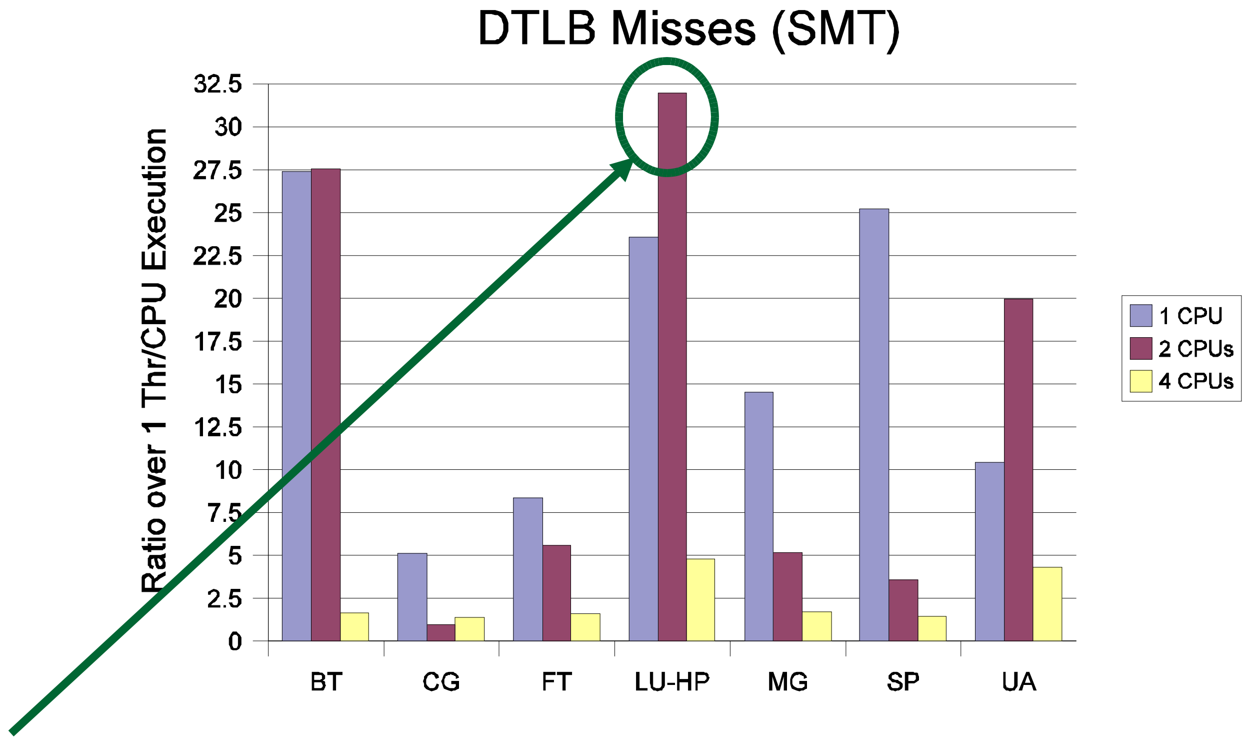
Data TLB Misses SMT



The number of DTLB misses increases dramatically with use of second execution context

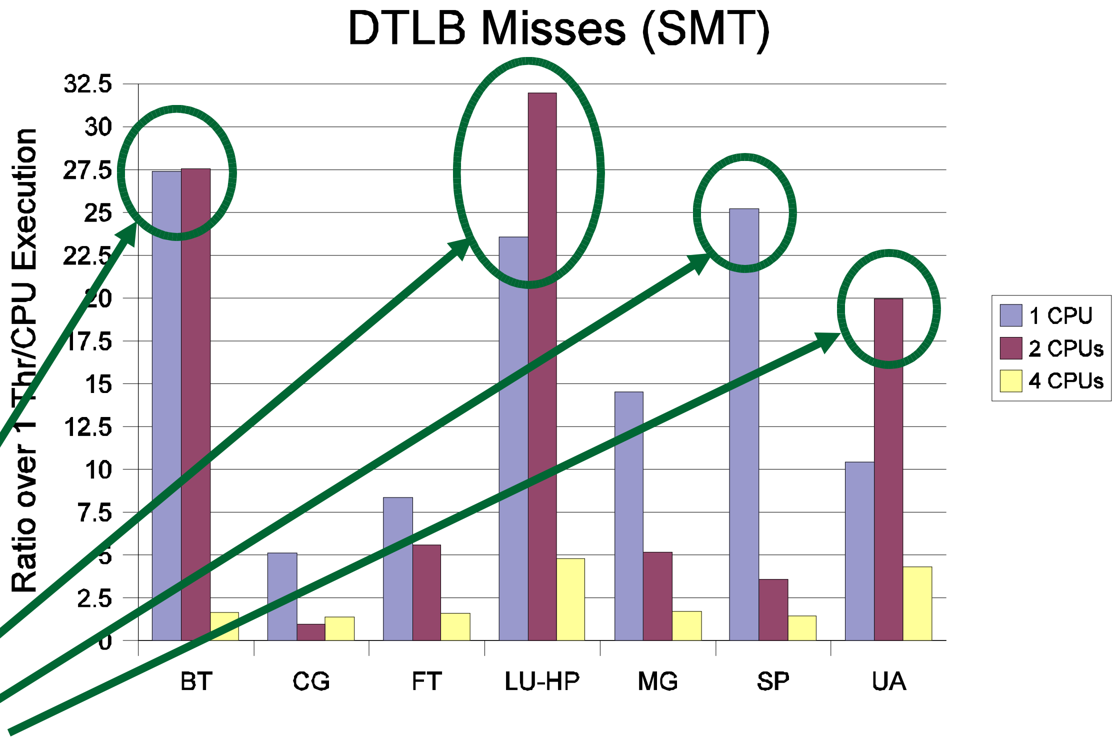
William & Mary

Data TLB Misses SMT



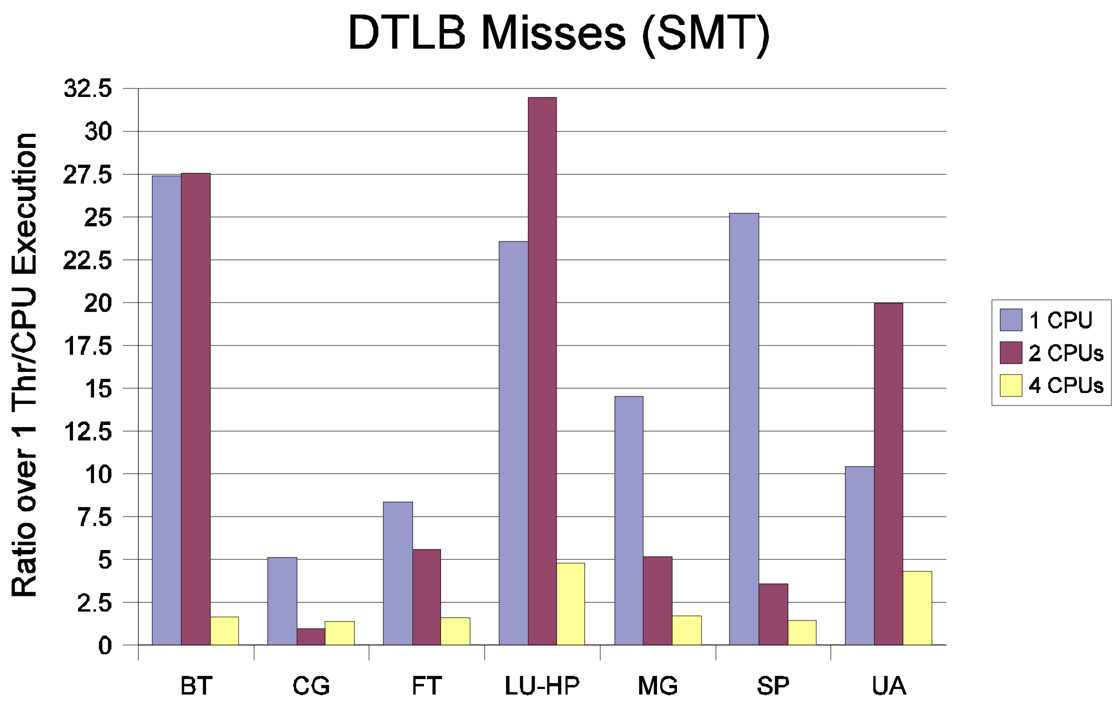
DTLB misses suffer up to a 32-fold increase

Data TLB Misses SMT



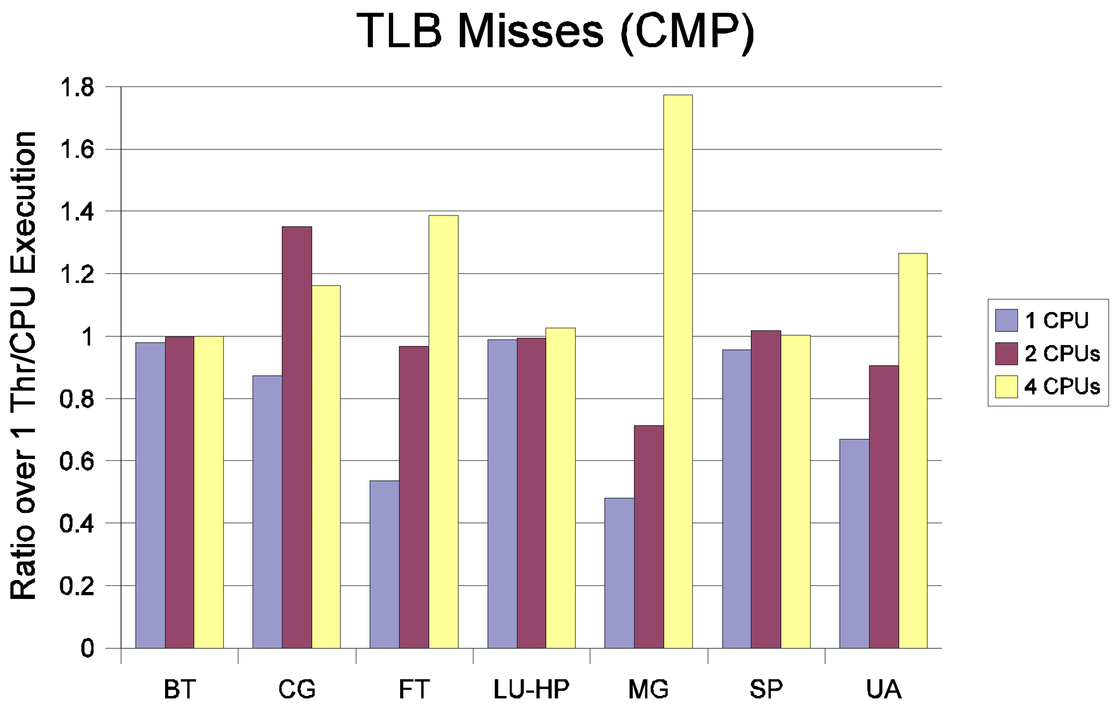
6 executions suffer a 20 or more fold increase

Data TLB Misses SMT



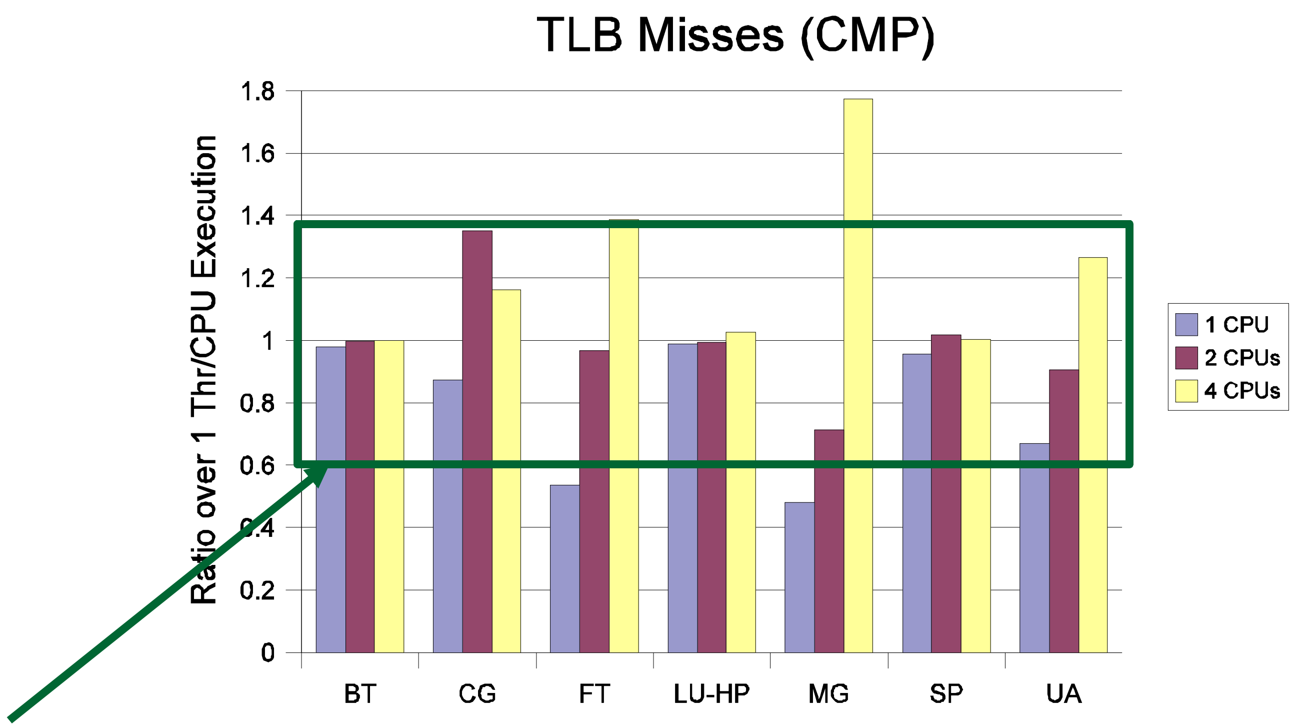
Intel's HT processor has surprisingly small DTLB -> poor coverage of the virtual address space

Data TLB Misses CMP



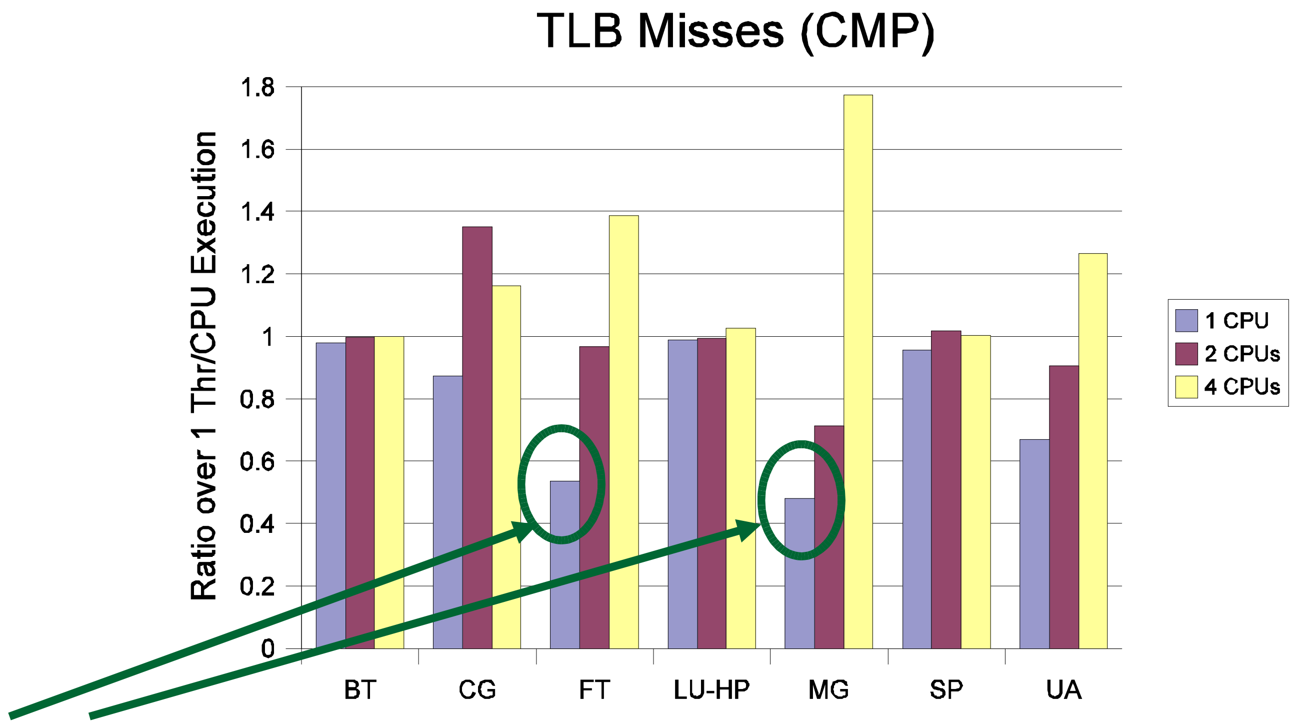
CMP provides private DTLB to each core, which results in much more stable DTLB performance

Data TLB Misses CMP



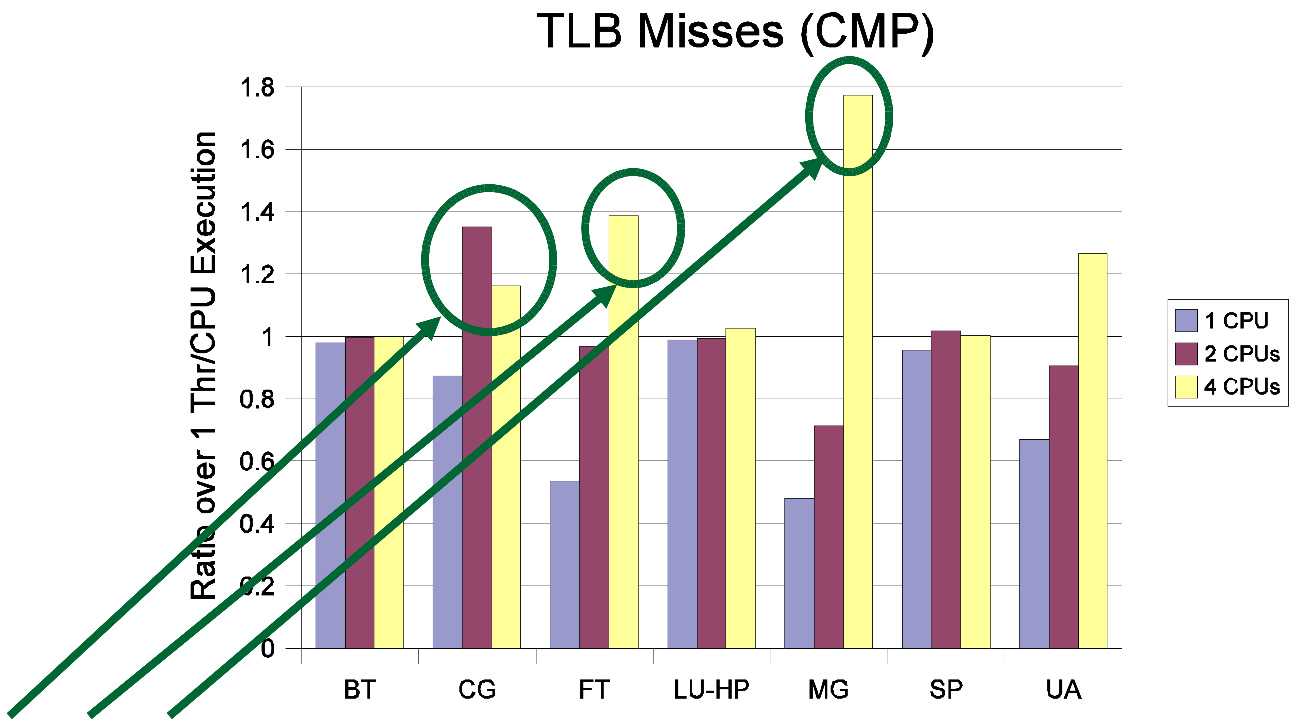
The majority of the executions experience normalized DTLB misses quite close to 1

Data TLB Misses CMP



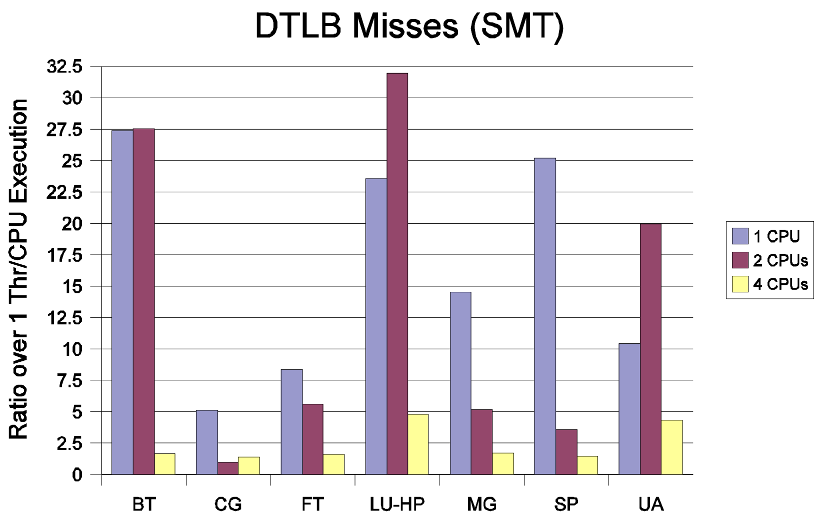
DTLB misses may decrease with 2 threads due to the cumulatively larger DTLB size from the DTLB duplication

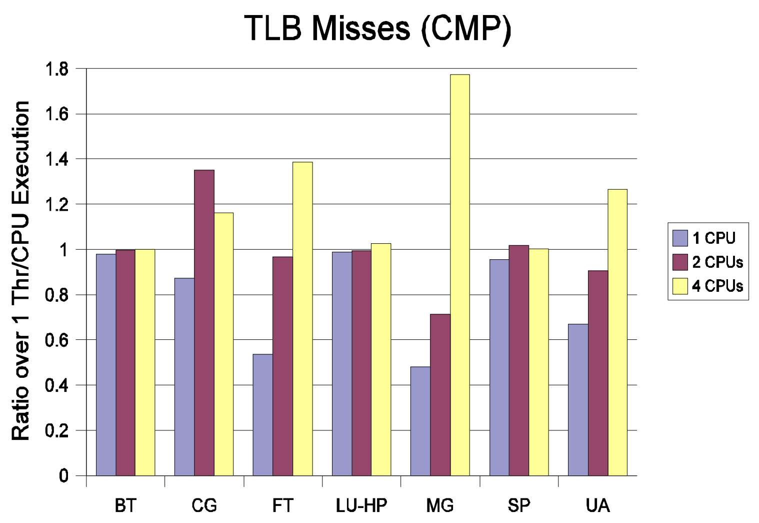
Data TLB Misses CMP



But if entries are duplicated between threads, then benefits of replicated DTLBs are reduced

Data TLB Misses Comparison





Privatizing the DTLB significantly reduces misses

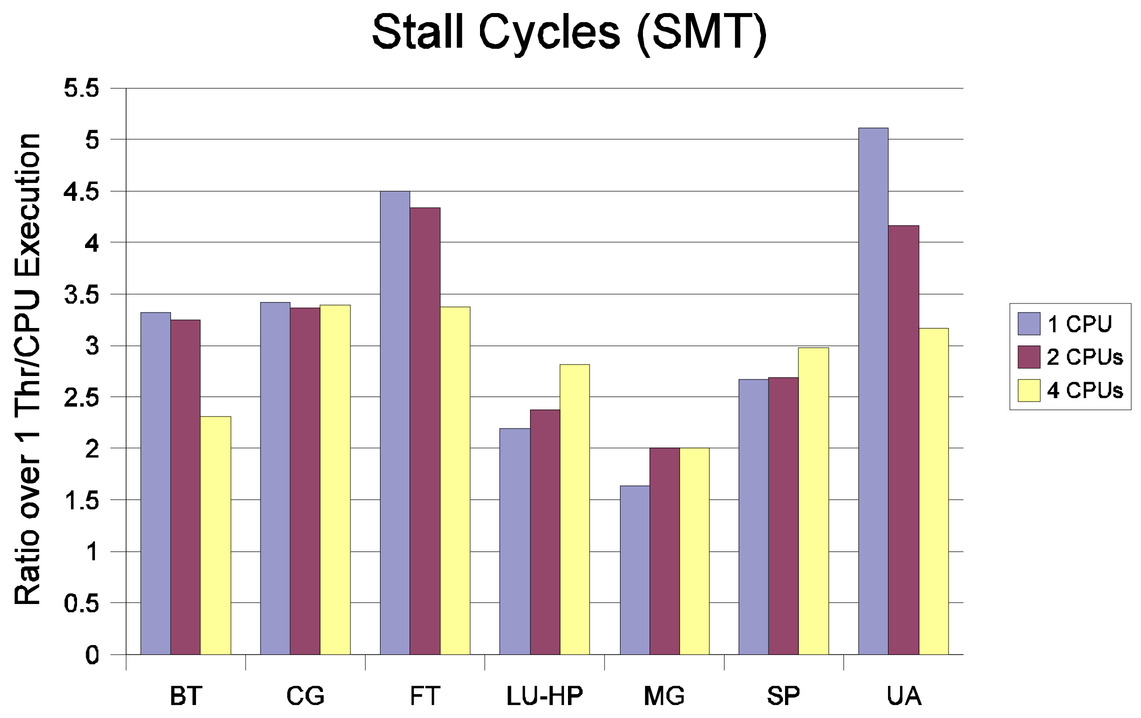
SMT average 10.8-fold increase

CMP average 0% increase

- Not very affected by multiple threads on a processor

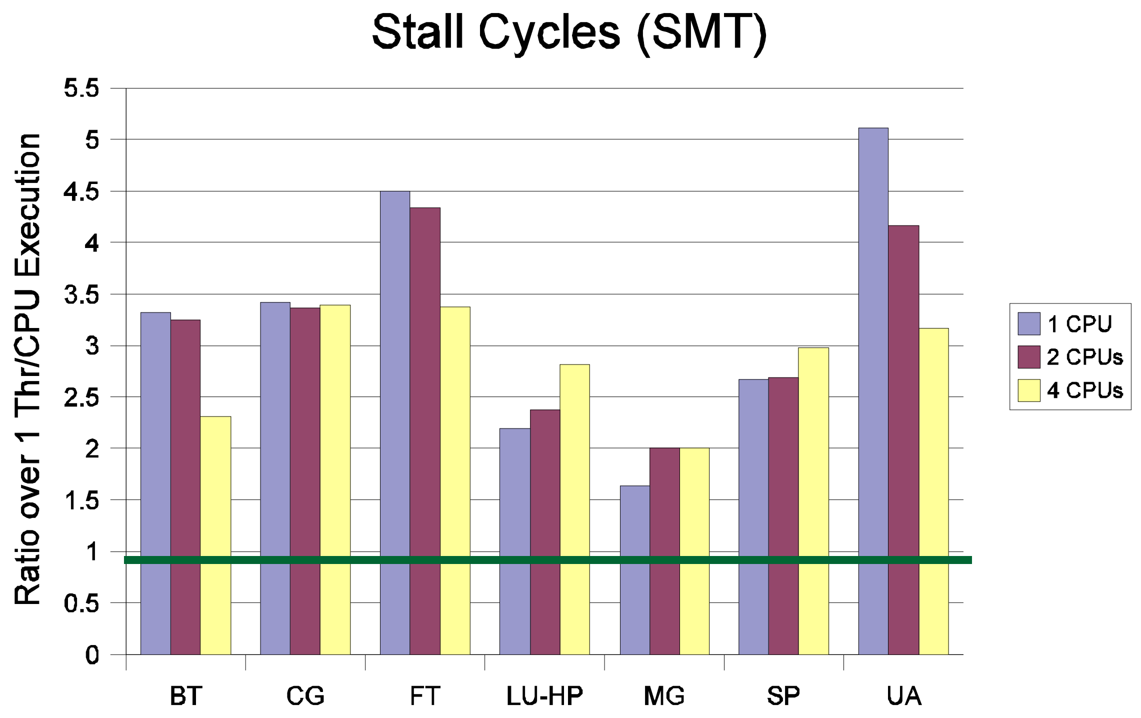
William & Mary

Stall Cycles SMT



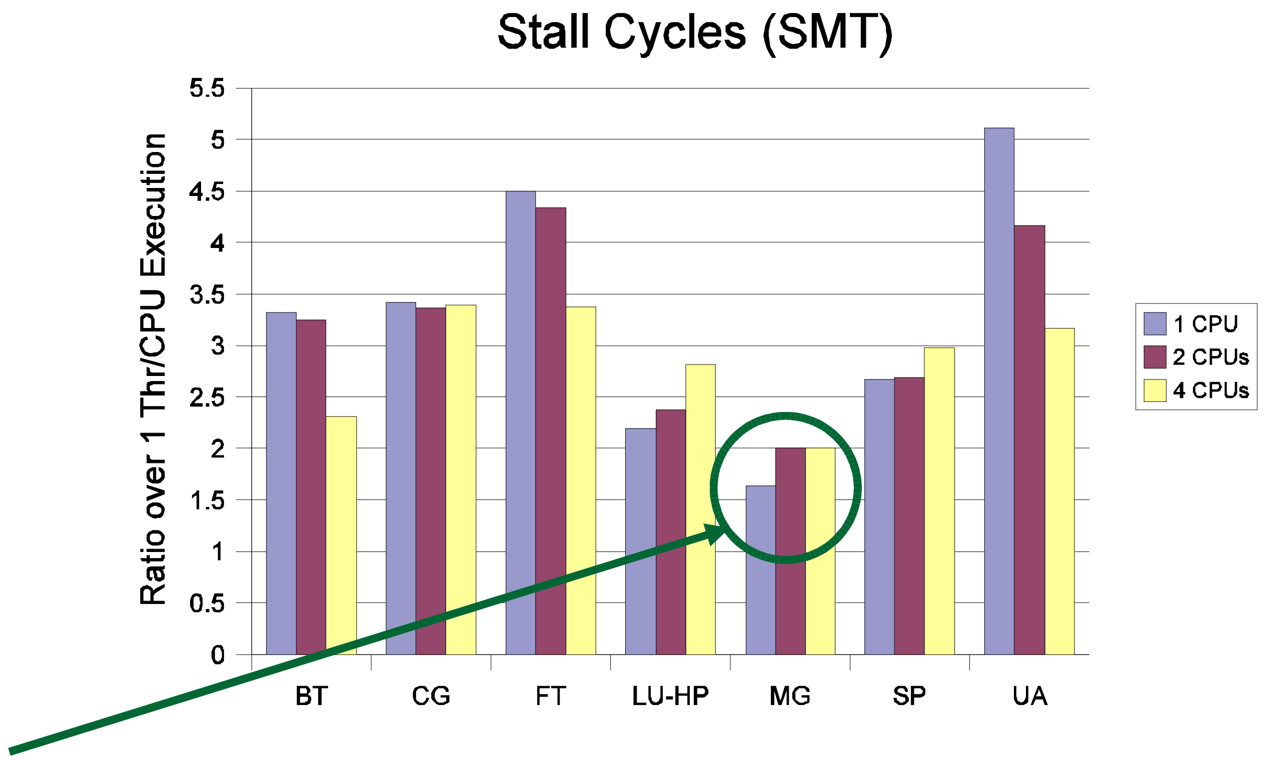
On SMT, stall cycles represent cumulative effects of waiting for memory accesses and resource contention between co-executing threads

Stall Cycles SMT



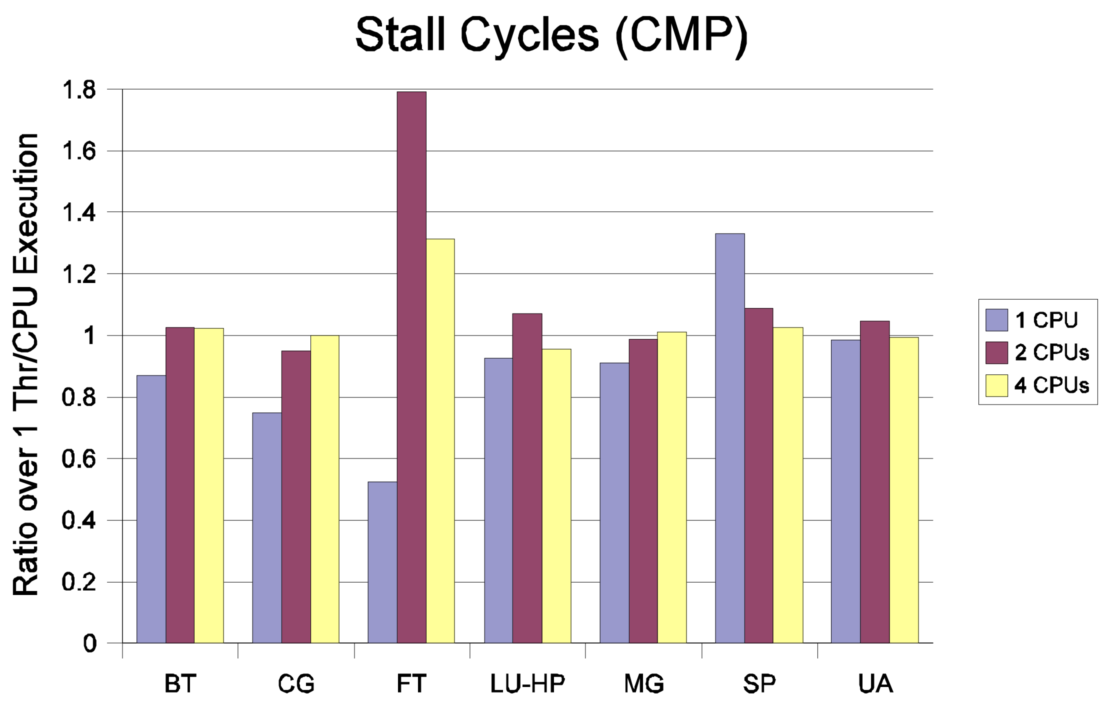
Stall cycles for all executions increase with use of second execution context

Stall Cycles SMT



In the best case, MG, stall cycles still increase by about a factor of 2

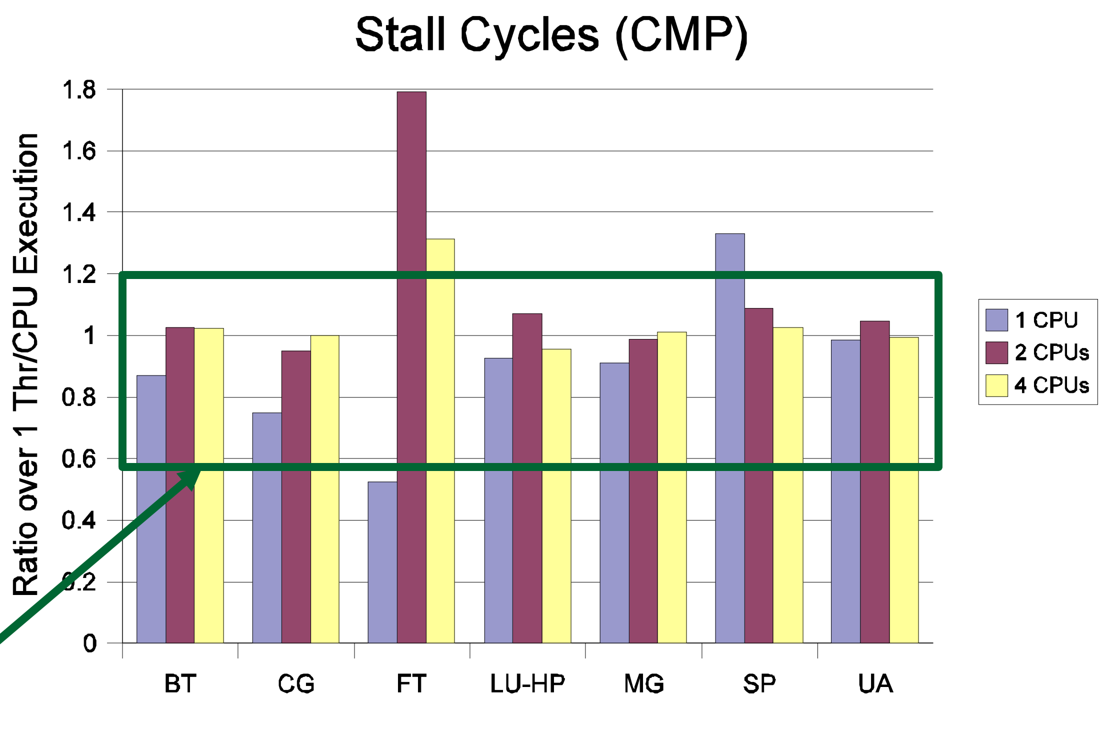
Stall Cycles CMP



CMP only shares outer levels of cache and interface to external devices, which greatly reduces possible sources of stall cycles

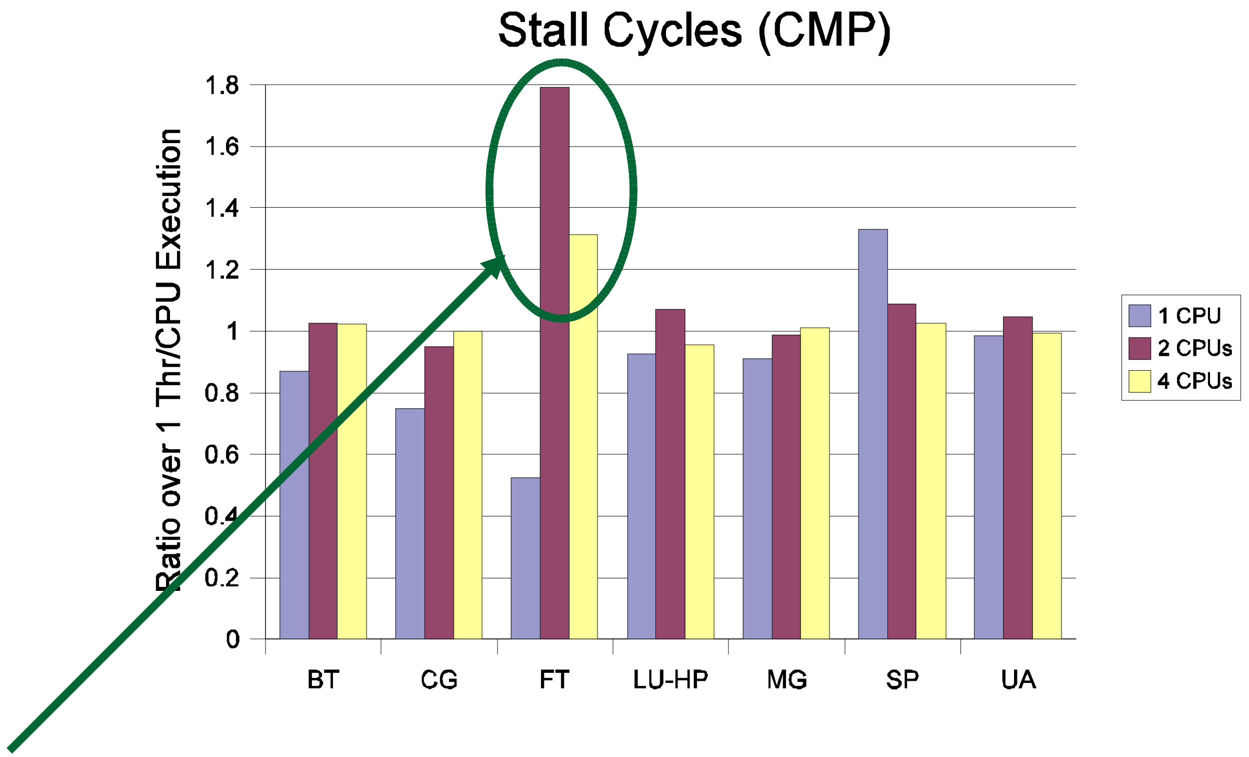
William & Mary

Stall Cycles CMP



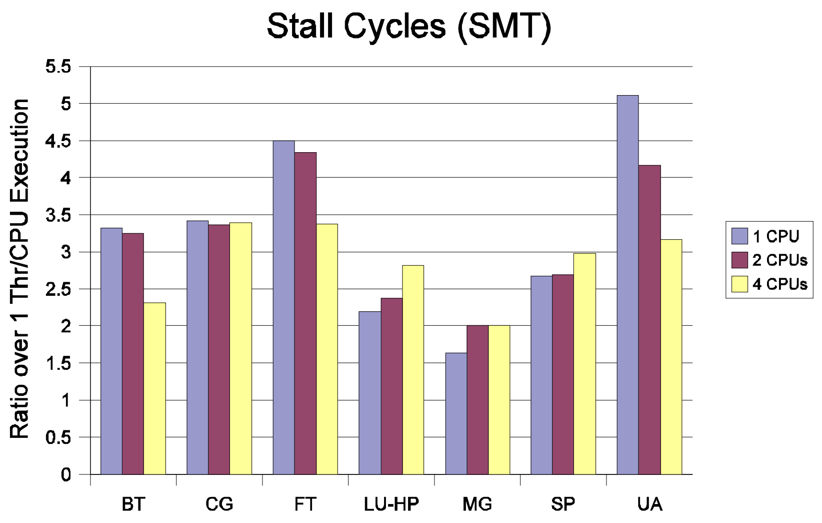
Once again, CMP's resource replication results in a stabilized number of stall cycles, close to 1

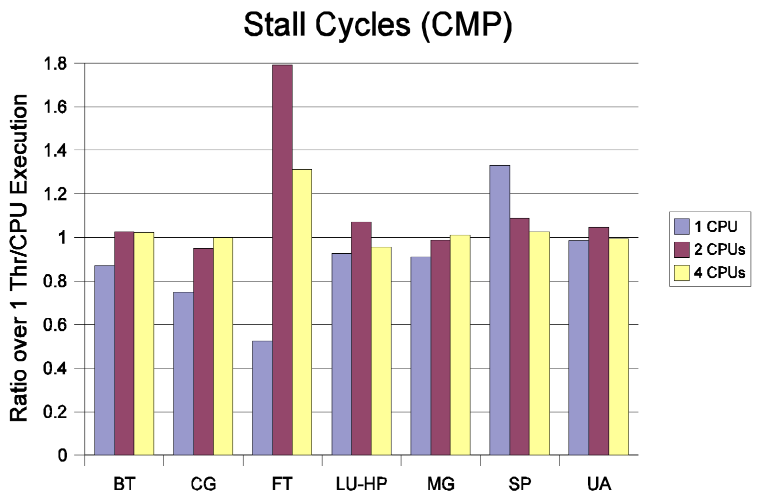
Stall Cycles CMP



FT has a relatively large increase in stall cycles
As we have already seen, it suffers from contention in the L2 and DTLB, even on the CMP architecture

Stall Cycles Comparison



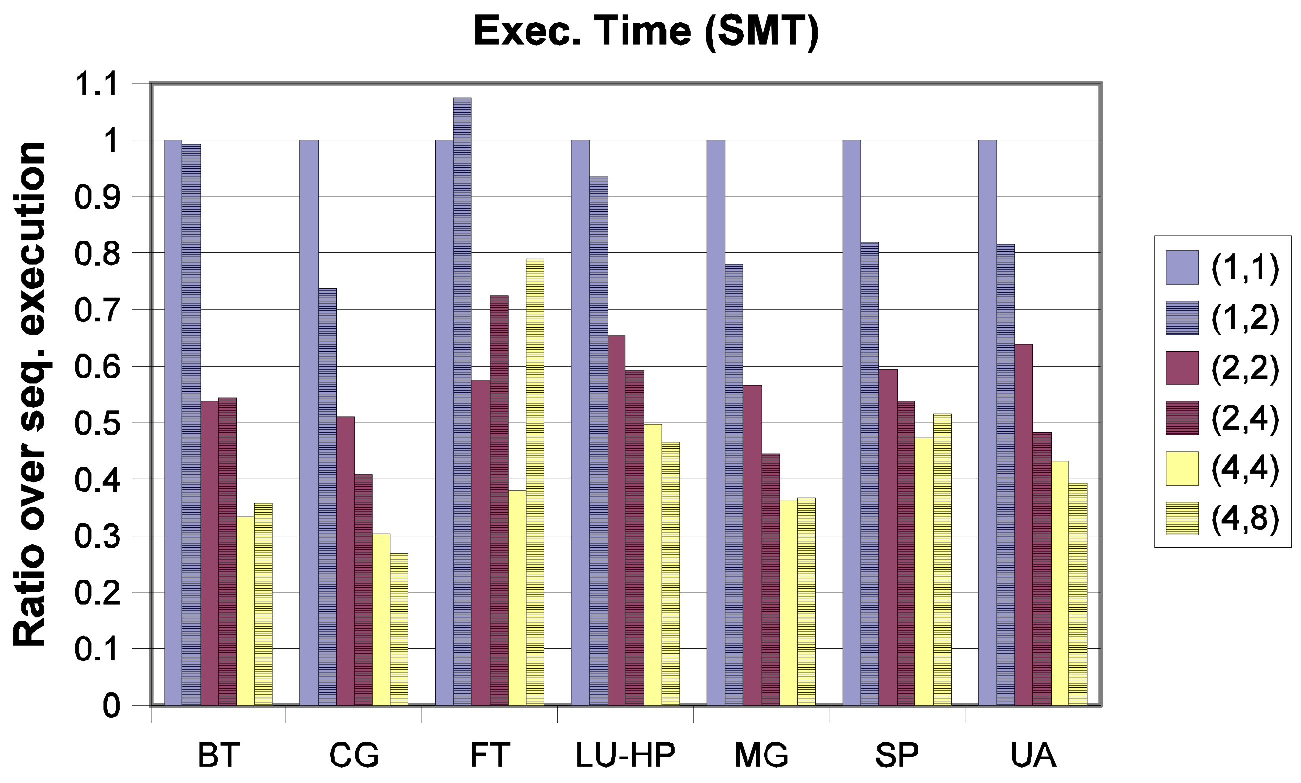


Increase of 310% for SMT vs. only 3% for CMP

Signifies that vast majority of stalls on SMT result from contention for internal processor resources

William & Mary

Execution Time SMT

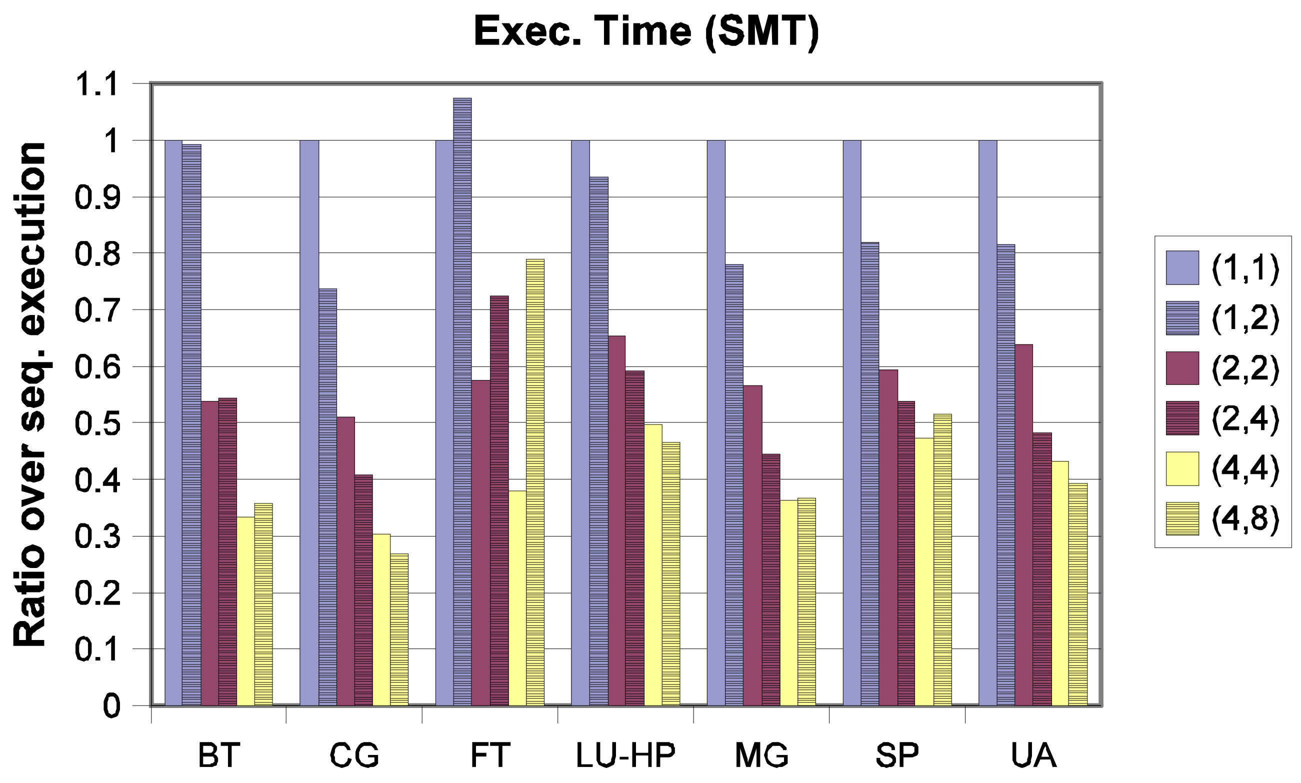


Two ways to evaluate the data:

Fixed number of CPUs, different number of threads

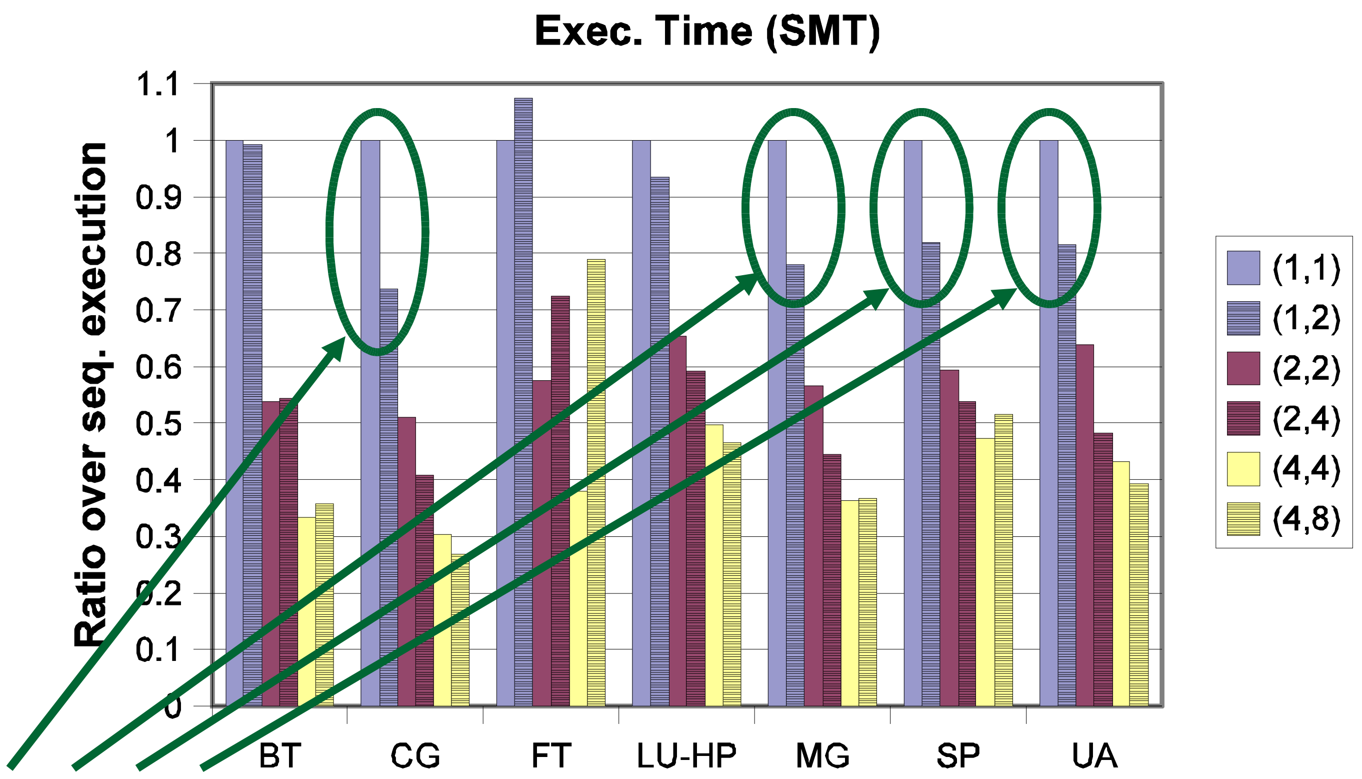
Fixed number of threads, different number of CPUs

Execution Time SMT



Running two threads on single CPU is not always beneficial for execution time compared to using a single thread

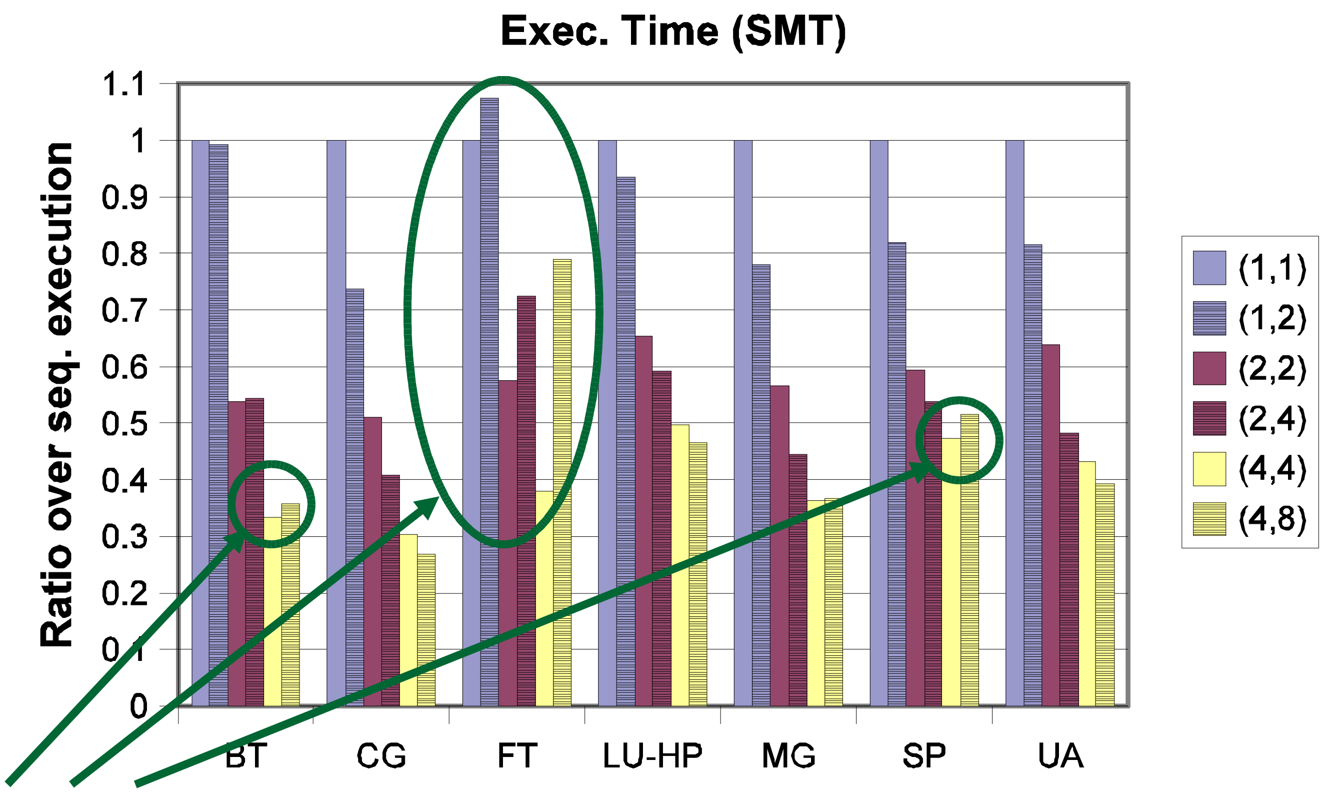
Execution Time SMT



Running two threads on single CPU is not always beneficial for execution time

Good in some cases…

Execution Time SMT

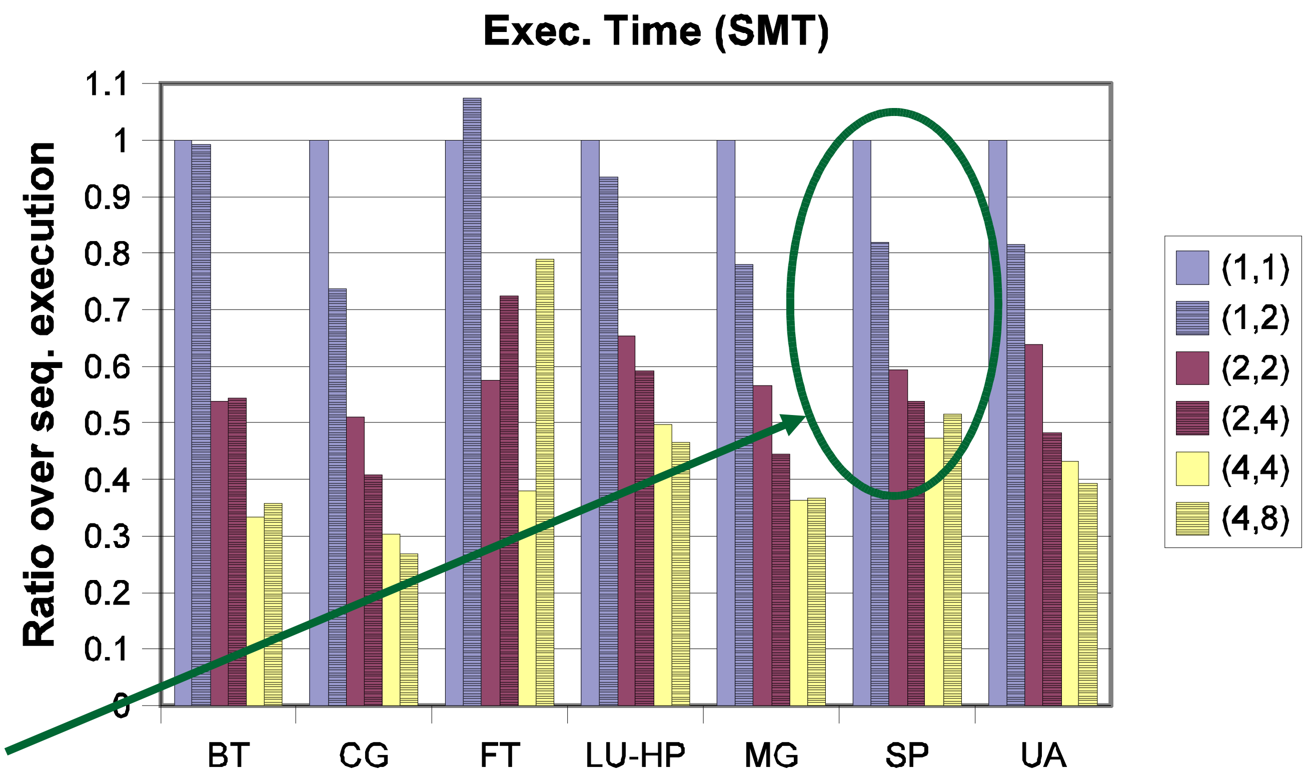


Running two threads on single CPU is not always beneficial for execution time

…Bad in others

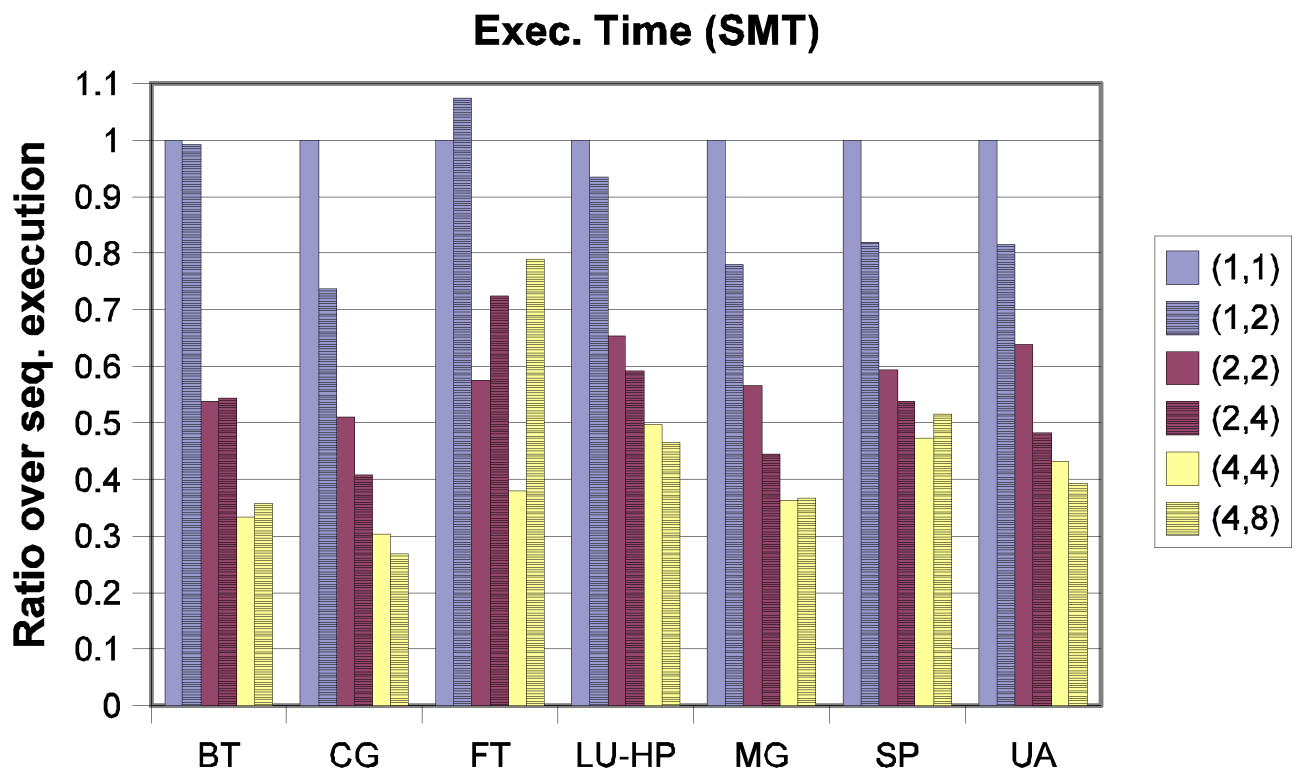
William & Mary

Execution Time SMT



Even for a given application, neither one thread nor two threads per processor is always optimal

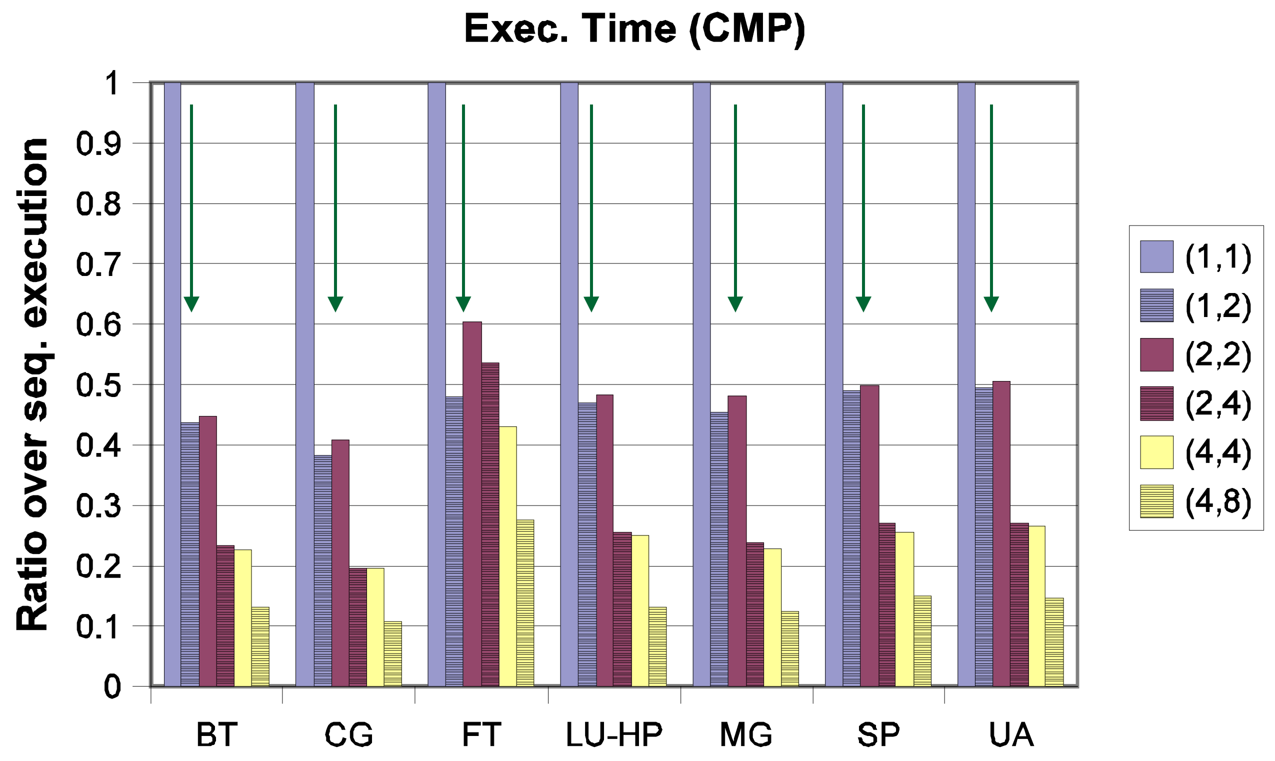
Execution Time SMT



For a fixed number of threads, it is always better to execute them on as many different physical processors as possible

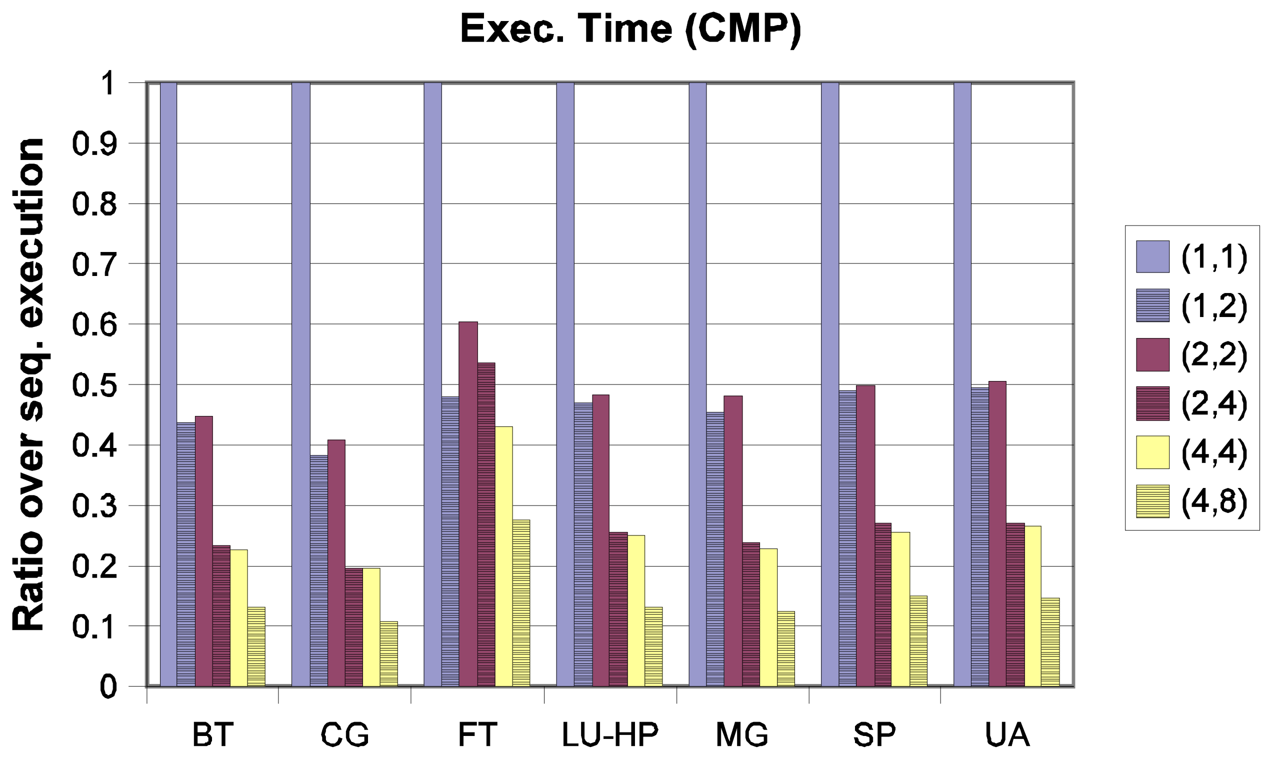
William & Mary

Execution Time CMP



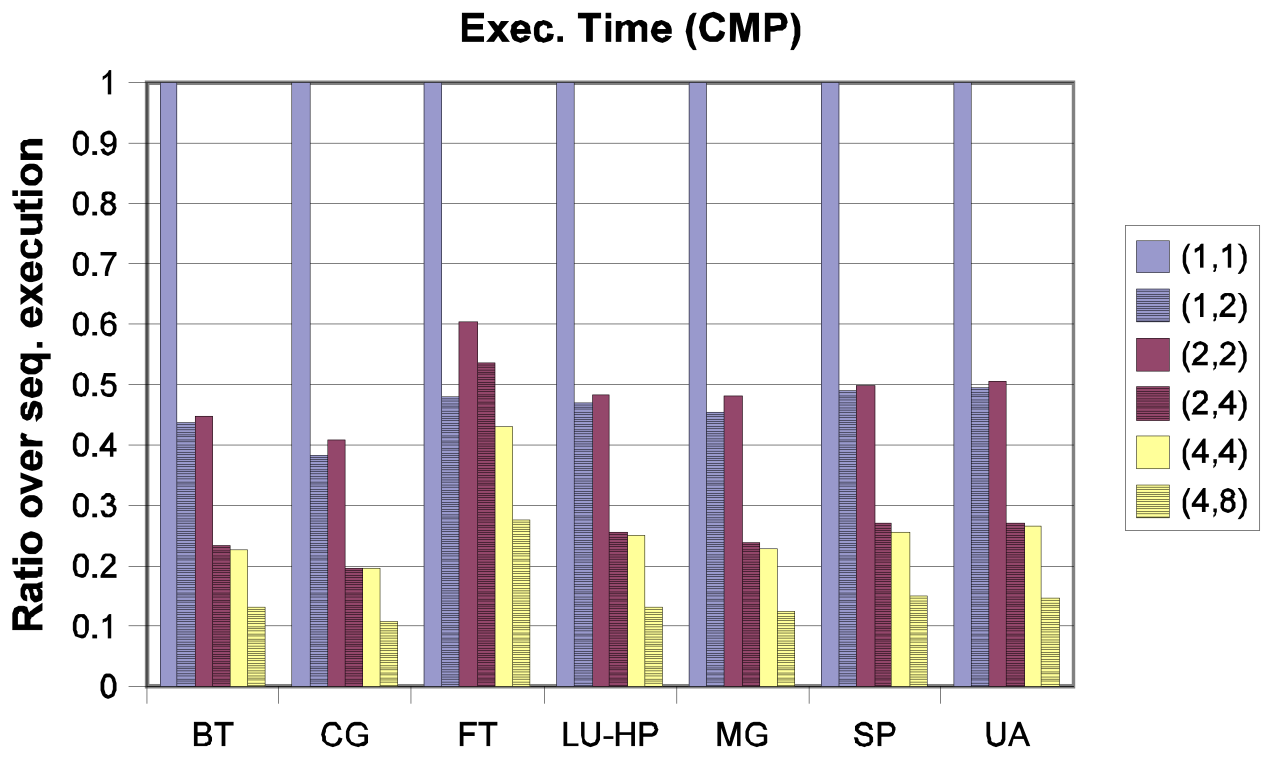
CMP, on the other hand, utilizes two threads per CPU very well

Execution Time CMP



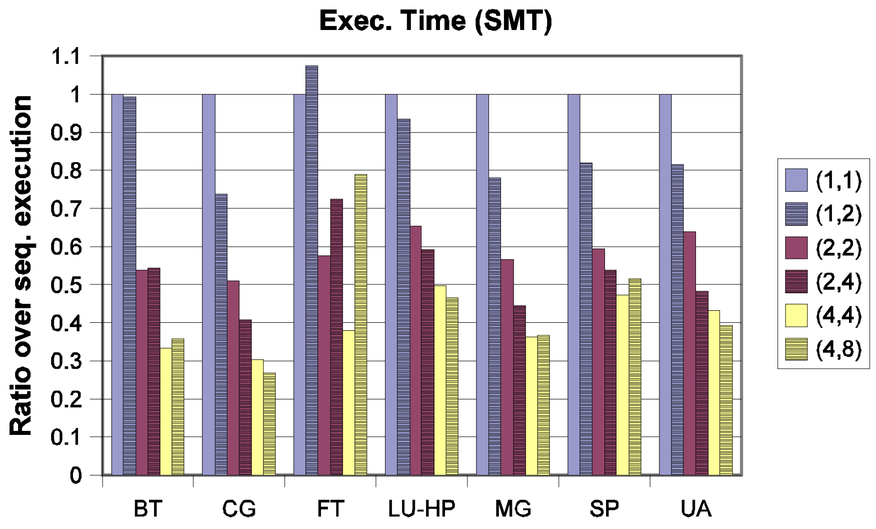
The activation of the second execution context was always beneficial

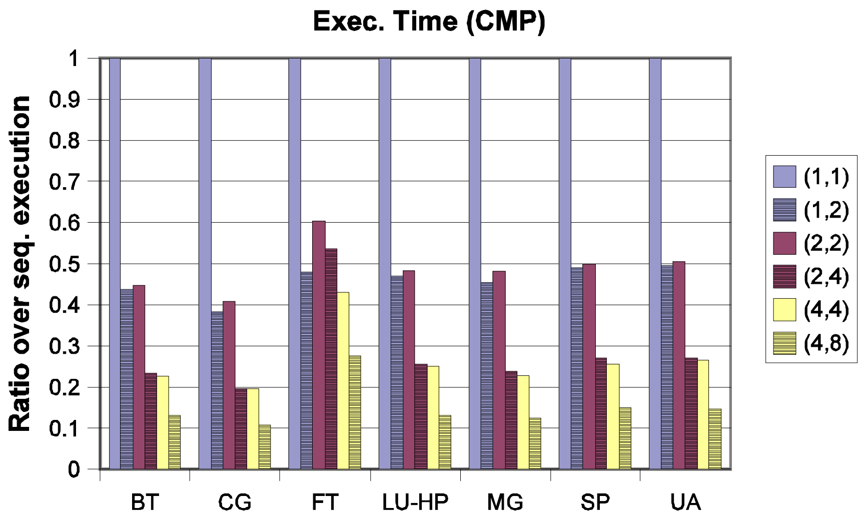
Execution Time CMP



For a given number of threads, it was often better to run them on as few processors as possible

Execution Time Comparison





- CMP handles using two threads per processor much better than SMT
  - Due to greater resource replication in CMP, which reduces contention
  - CMP is a cost-effective means of improving performance

William & Mary

Content

William & Mary

Adaptive Approach Description

- Neither 1 or 2 threads per CPU is always better
- Based on work by Zhang, et al from U. Toronto (PDCS'04) we try both and use whichever performs better
- Selection is performed at the granularity of a parallel region
- Function calls before and after each region, could be inserted by preprocessor
- We only consider number of threads, rather than scheduling policy
  - However, no manual changes to source code
  - And no modifications to compiler or OpenMP runtime

William & Mary

Description, cont.

```
Outermost Loop {
        !$OMP PARALLEL{ … }   // Parallel Region 1
        !$OMP PARALLEL{ … }   // Parallel Region 2
        …
        !$OMP PARALLEL { … }   // Parallel Region N
}
```

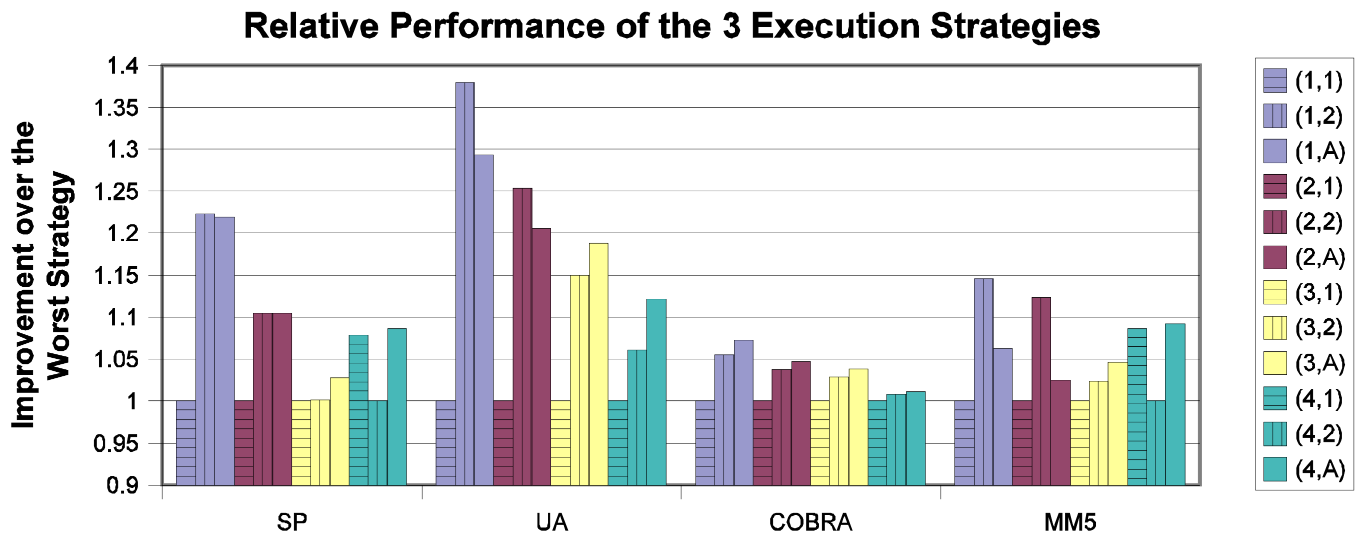
- Since NPB are iterative, we record execution time of 2nd and 3rd iterations with 1 and 2 threads
  - Ignore 1st iteration as cache warm-up
  - Whichever number of threads performs better is used when the region is encountered in the future

William & Mary

Adaptive Experiments

- Used the same 7 NPB benchmarks along with two other OpenMP codes
  - MM5: a mesoscale weather prediction model
  - Cobra: a matrix pseudospectrum code
- Ran on 1, 2, 3, and 4 processors
  - Compared adaptive execution times to both 1 and 2 threads per processor

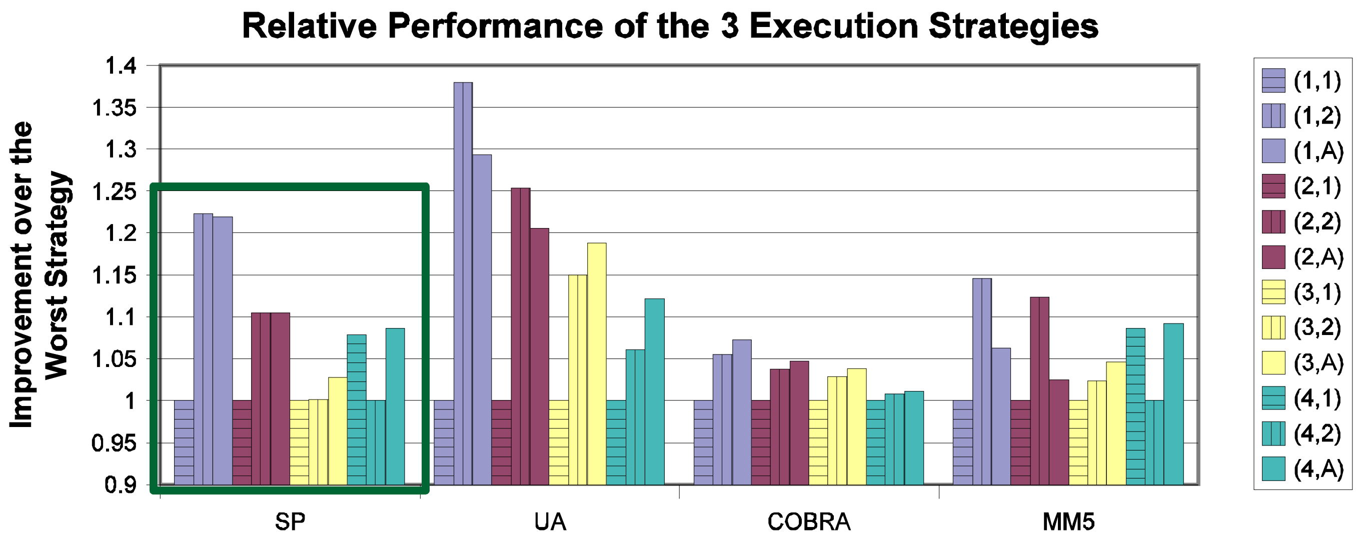
Results from Adaptation



Graph shows relative performance of each approach for 1, 2, 3, and 4 processors

- 1 thread per processor
- 2 threads per processor
- Adaptive approach

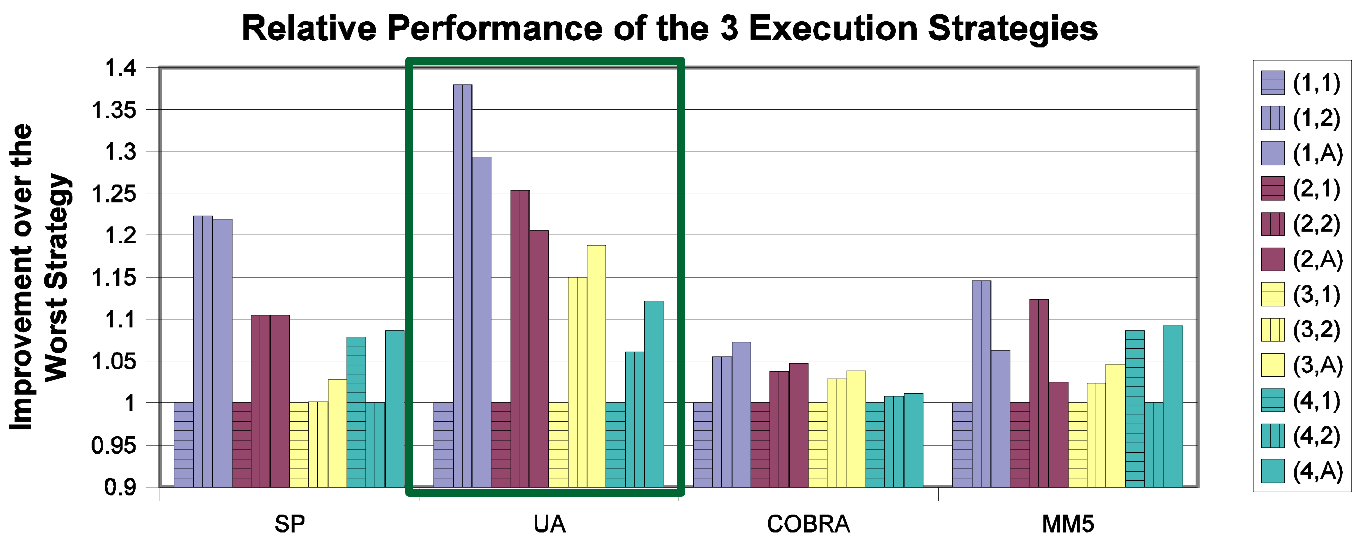
Results from Adaptation



Graph shows relative performance of each approach for 1, 2, 3, and 4 processors

- 1 thread per processor
- 2 threads per processor
- Adaptive approach

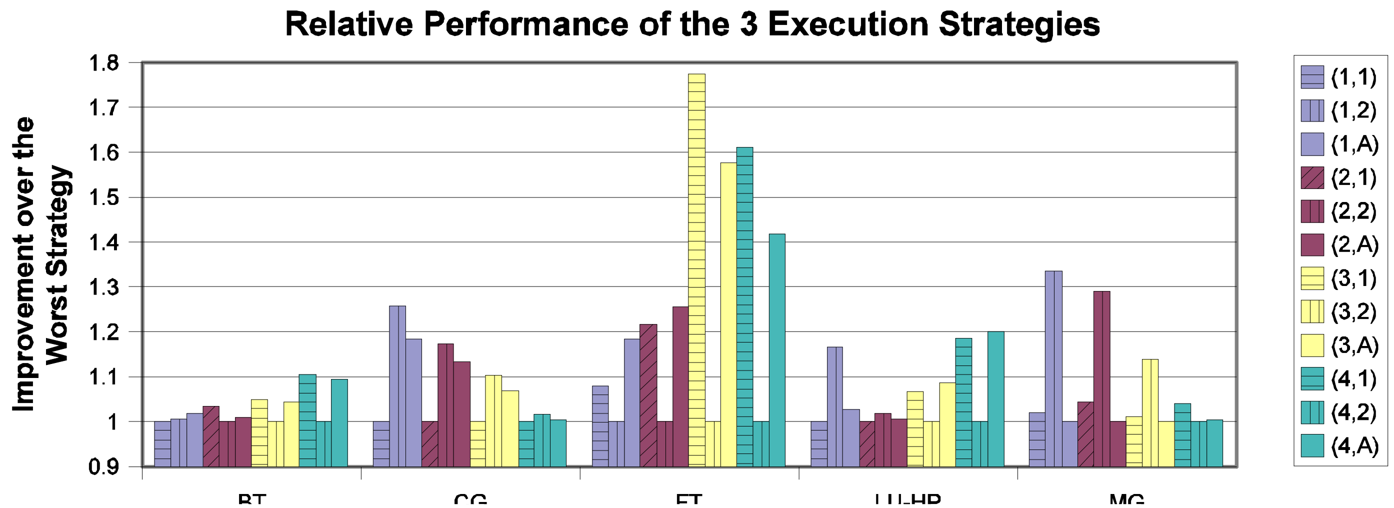
Results from Adaptation

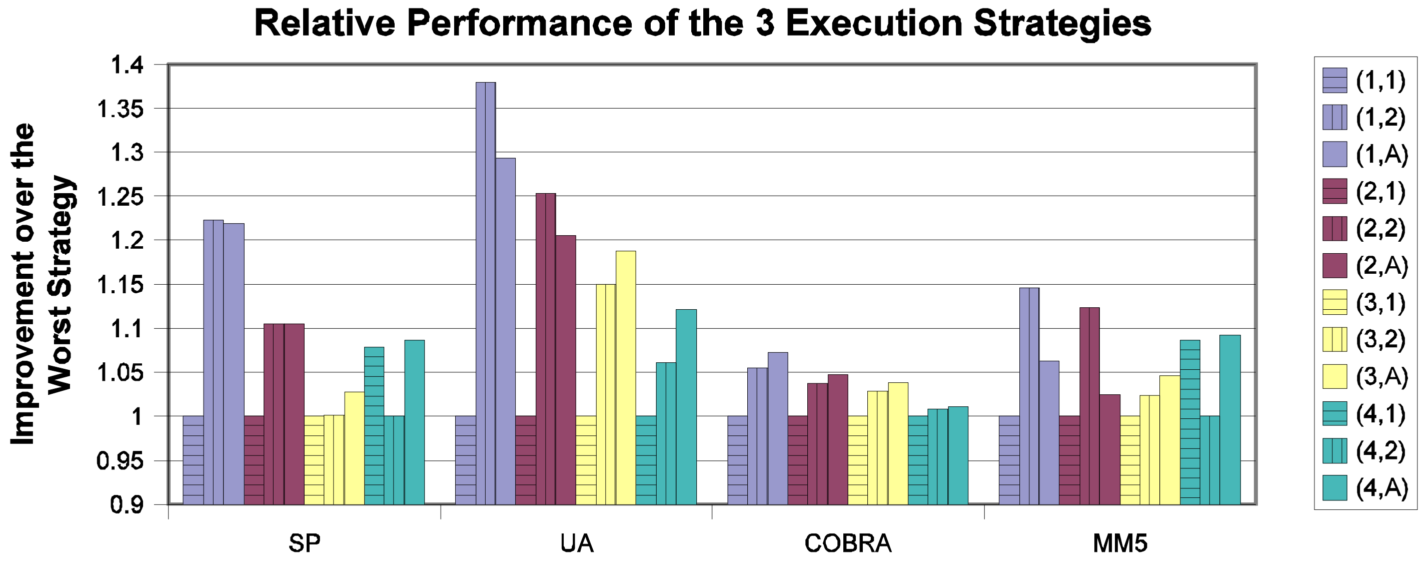


Graph shows relative performance of each approach for 1, 2, 3, and 4 processors

- 1 thread per processor
- 2 threads per processor
- Adaptive approach

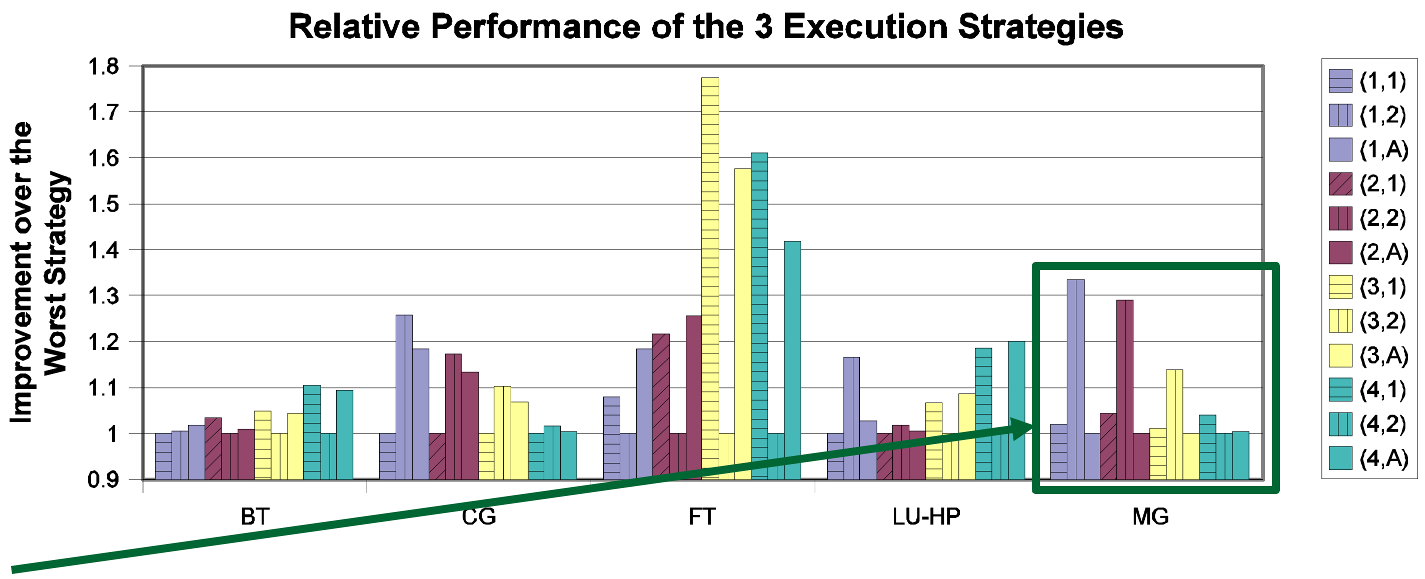
Results from Adaptation





William & Mary

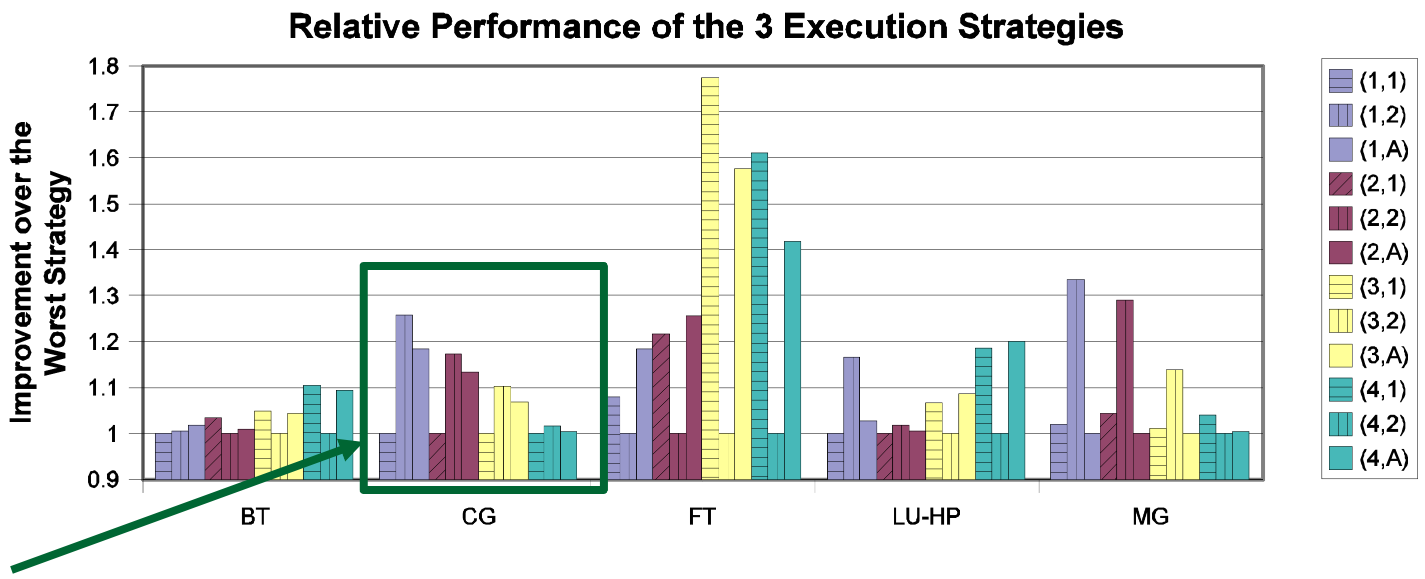
Results from Adaptation



Adaptation does not perform well for MG

MG has only 4 iterations and our approach takes 3

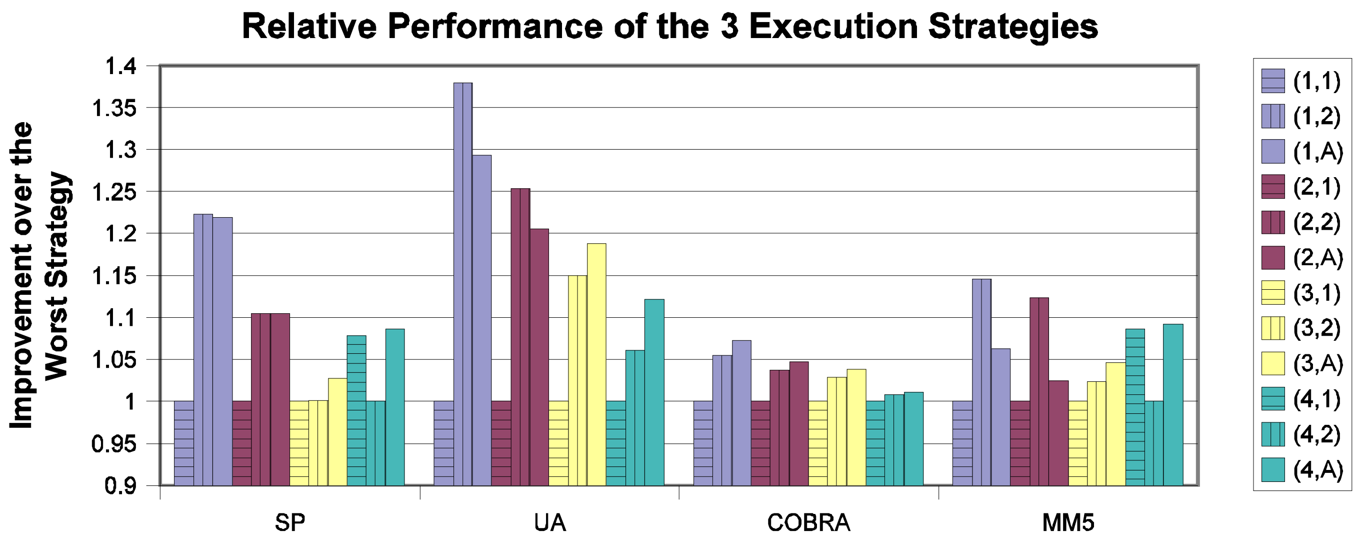
Results from Adaptation



- Adaptation does not perform well for MG
  - MG has only 4 iterations and our approach takes 3
- CG, however, performs well with only 15 iterations
  - So it does not require many iterations to be profitable

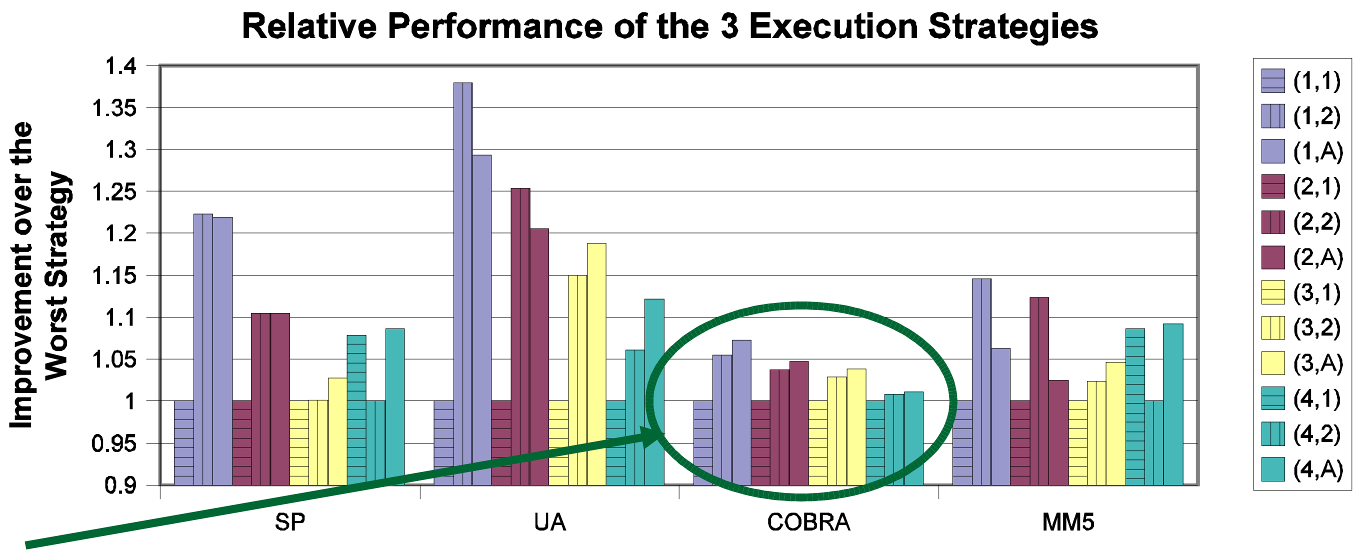
William & Mary

Results from Adaptation



In 17 of the 36 experiments, adaptation did better than either static number of threads

Results from Adaptation



In 17 of the 36 experiments, adaptation did better than either static number of threads

In Cobra, adaptation was the best for all numbers of processors

Results from Adaptation

- Compared to optimal static number of threads, adaptation was only 3.0% slower
- It was, however, 10.7% faster than the worse static number of threads
- The average overall speedup was 3.9%
- This shows that adaptation provides a good approximation of the optimal number of threads
  - Requires no *a priori* knowledge
- However, does not overcome inherent architectural bottlenecks

Content

William & Mary

Implications for OpenMP

- Our study indicates that OpenMP scales effortlessly on CMPs
- It is important to consider optimizations of OpenMP for SMT processors
  - Viable technology for improving performance on a single core
- These optimizations could come from:
  - Additional runtime environment support
  - Extensions to the programming interface

OpenMP Optimizations for SMT

- Co-executing thread identification is most important optimization
- New *SCHEDULE* clause may be used
  - Can assign iterations to SMTs
  - These iterations can then be split between co-executing threads using SMT-aware policy
- OpenMP thread groups extensions may be used
  - Co-executing threads go to same group
  - Use SMT-aware scheduling and local synchronization
  - Not necessarily nested parallelism

OpenMP Optimizations for SMT

- Necessity of thread binding
  - SMT-aware optimizations require threads to remain on the same processor
  - Some applications may benefit from running 2 threads on the same processor
- Use of proposed mechanisms, like *ONTO* clause
- However, exposing architecture internals in the programming interface is undesirable in OpenMP
- New mechanisms for improving execution on SMT processors in an autonomic manner

William & Mary

Content

William & Mary

Conclusions

- Evaluated the performance of OpenMP applications on SMT/CMP-based multiprocessors
  - SMTs suffer from contention on shared resources
  - CMPs more efficient due to greater resource replication
  - CMPs appear to be more cost effective
- Adaptively selecting the optimal number of threads helps SMT performance
  - However, inherent architectural bottlenecks hinder the efficient exploitation of these architectures
- Identified OpenMP functionality that could be used to boost performance on SMTs

William & Mary