Meeting Points – Using Thread Criticality to Adapt Multicore Hardware to Parallel Regions

QIONG CAI, JOSE GONZALEZ, RYAN RAKVIC, GRIGORIOS MAGKLIS, PEDRO CHAPARRO & ANTONIO GONZALEZ, PACT '08, PROCEEDINGS OF THE 17TH INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURE & COMPILATION TECHNIQUES,PAGES 240-249

Ishan Dalal

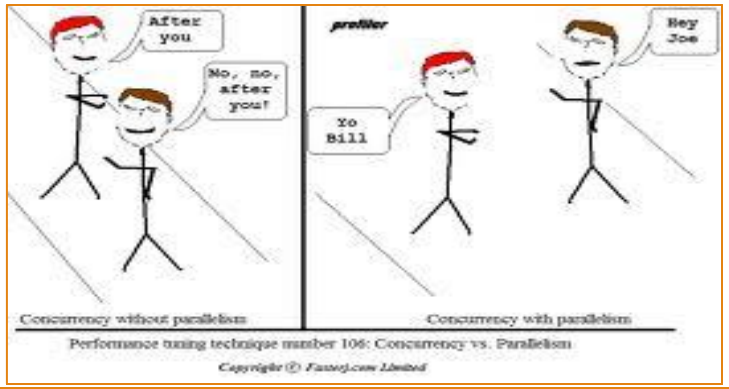
Introduction

- Multicore Systems – ICs containing two or more processors for enhanced performance, reduced power consumption and more efficient simultaneous processing of multiple tasks (parallel processing).

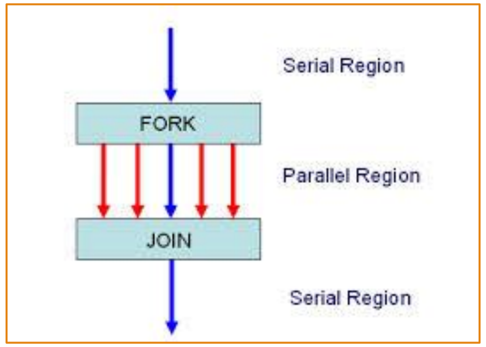
Each Core contain one or more threads. Threads are an independent smallest sequence of programmed instructions managed by OS.

On single Processor, multithreading achieved by time-division multiplexing i.e. Context Switching.

On Multi-Core Systems, Parallelism (True!) achieved by running threads simultaneously on each of the cores or processors.


After you
No, no, after you!
profiler
Yo Bill
Hey Joe
Concurrency without parallelism
Concurrency with parallelism
Performance tuning technique number 106: Concurrency vs. Parallelism
Copyright © Fasterj.com Limited

Challenge with Chip Multiprocessors (CMPs)



- High Energy Consumption – Major hurdle in design of systems.
  - Workload Imbalance among cores is source of energy inefficiency.

Example – In OpenMP's fork-join parallel execution model, there is a barrier at the joint point of the parallel loop that synchronizes all threads.

- If Lucky, all cores reach this barrier at the same time.
- In normal cases, some threads reach the barrier earlier than others and spend a large amount of time waiting for slower ones.
- Fast threads have been executed at maximum possible speed and power consumption which leads to the inefficiency.

Possible Solutions

- Put fast threads to sleep as soon as they arrive at the barrier and then shut down the core.
  - Feasible Approach if the waiting time is long enough so that the energy saved in sleep mode pays off the energy/performance wasted by putting the cores to sleep and waking them up.
- If the thread is known beforehand to reach the synchronization point early, the voltage and frequency of the core running that thread could be reduced dynamically without compromising the performance.
  - Better than former because Dynamic Voltage Frequency Scaling (DVFS) achieves greater energy reduction compared to putting a core to sleep due to cubic relationship of power to voltage/frequency.

Approaches & its challenges

- Detection of Critical and Non-critical threads.
  - Used Slack as proxy to know the Criticality level of parallel thread.
  - Slack is the amount of time a thread can be delayed with no impact on final performance.
  - Critical thread is one with zero slack while other threads have non-zero slack.
  - Thus Other threads can be delayed as long as they don’t impact the performance.
  - Slack determines the level of criticality of each thread.

- **Challenge – Detecting critical threads and level of criticality since one could not know a priori whether a thread is going to be the last one to reach the barrier.**

Meeting Point Thread Characterization (MPTC)

- Each thread has a counter to accumulate the number of iterations executed for the parallel loop.
- At specific intervals of time, all threads broadcast the information so they can know the number of iterations being executed by each one of them.
- Slack can be calculated from its own iteration counter and counter of the slowest one.

❑ Applications done by the Study Using this Optimization

1) **Thread Delaying (CMP)**
2) **Thread Balancing (SMT – Simultaneous Multi-threading)**

1) Thread Delaying

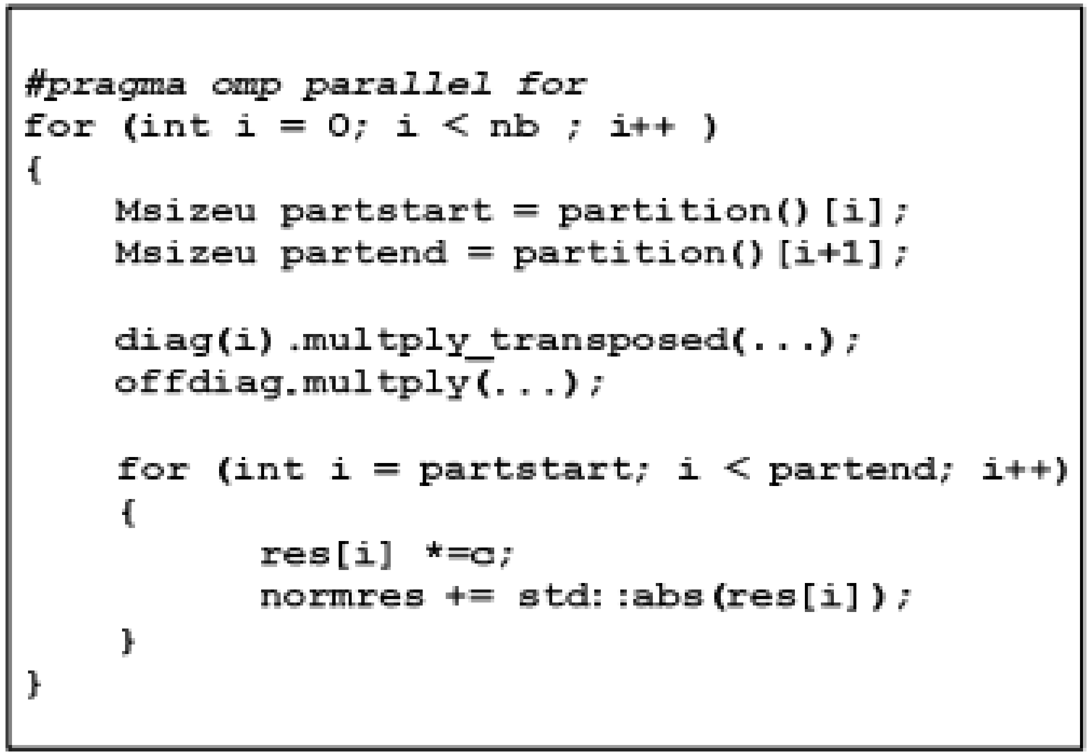
- Goal : To reduce overall energy consumption by dynamically scaling down the voltage and frequency of the cores executing non-critical threads.
- At specific intervals of code execution, each core utilizes the MPTC to estimate the slack of the parallel thread.
- It computes the voltage/frequency depending on the current slack for the next interval of time so that the energy is minimized but the expected arrival time to the barrier does not exceed that of the current critical thread.

2) Thread Balancing

- Goal : To reduce the overall execution time by speeding up the Critical thread.
- Used for simultaneous multithreading processors running parallel threads.
- Thread balancing does sharing of resources by issuing slots in such a way that if both threads have ready instructions, both are allowed to issue the same number of them.
- But when a critical thread is identified, it is given priority in the utilization of the issue slots.

Note : Threads come from the same parallel application.

MPTC - Mechanism



```
#pragma omp parallel for
for (int i = 0; i < nb ; i++ )
{
    Msizeu partstart = partition()[i];
    Msizeu partend = partition()[i+1];

    diag(i).multply_transposed(...);
    offdiag.multply(...);

    for (int i = partstart; i < partend; i++)
    {
        res[i] *=c;
        normres += std::abs(res[i]);
    }
}
```

Figure (a)

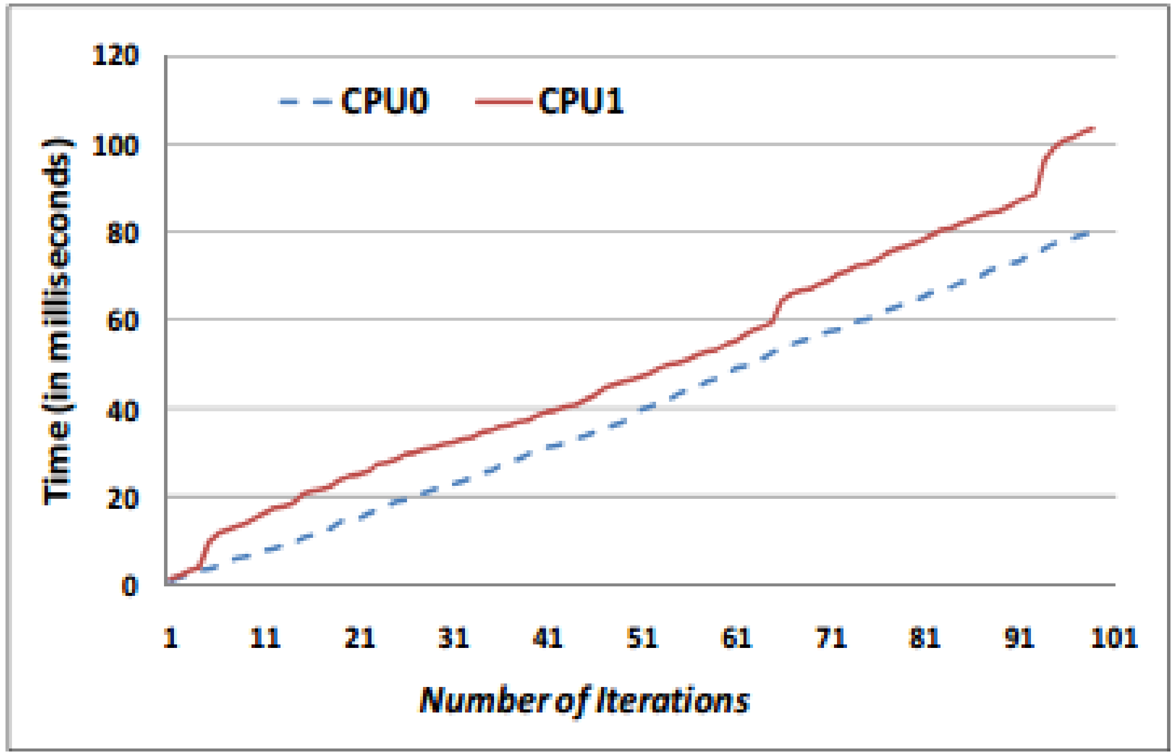


Figure (b)

Code Performance

- Figure (a) shows code written in such a way that the input data set is partitioned to achieve workload balance.
- However there exists workload imbalance on a two-core system as shown in Figure (b).
- In figure (b), x-axis represents the number of iterations of the outermost loop that each core executes. Y-axis represents the cumulative execution time of this parallel loop for each core.
- We can see, core 1 is slower than 0.

Probable Reasons :

- Core 1 suffers many more cache misses than core 0.
- Parallel threads follow different control path in the parallel loop or that application exploits task-level parallel, rather than loop level.

Check/Meeting Points

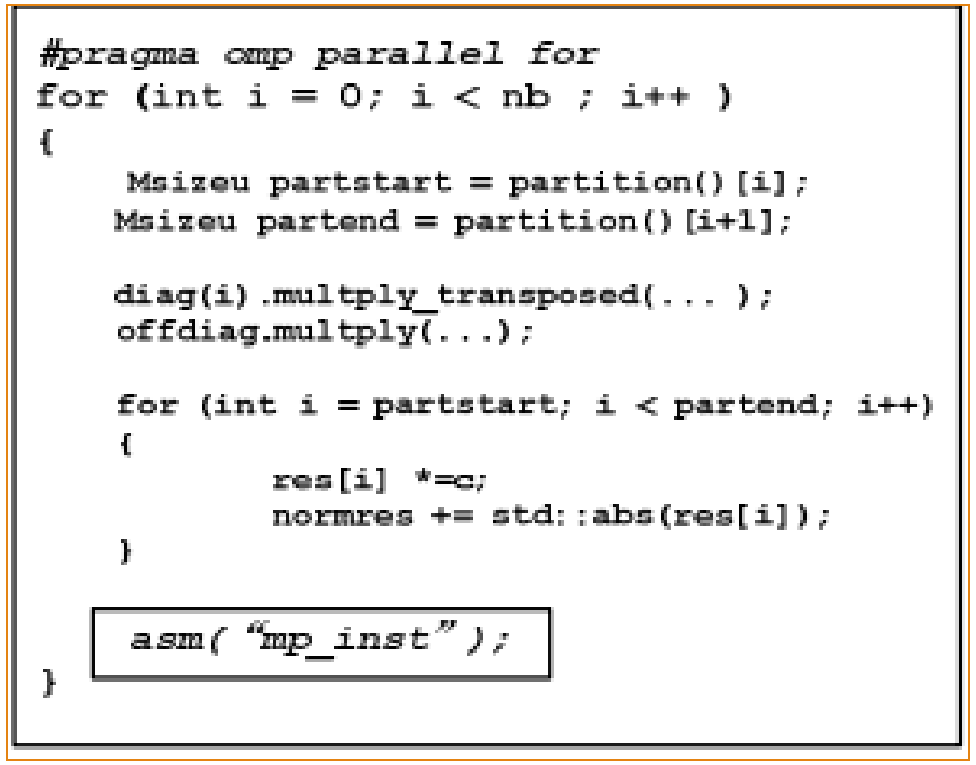
- ➢ Check the workload Balance at intermediate points of a parallel loop. These points are check/meeting points.
- ➢ Location : Back edge of a parallel loop because it is visited many times by all threads at runtime. Total amount of work assigned to each thread should be same since the total number of times each thread visits the meeting point is roughly the same.

❑ The Process of MPTC consists of the three steps :

1) Insertion of meeting points
2) Identification of Critical Threads
3) Usage of Criticality Information

1) Identification of Meeting Points

- Meeting point should be the one that is visited many number of times in the parallel region.
- As seen in the code below, last statement of the outermost loop (or the parallelized loop) satisfies the criteria.
- Insertion of meeting point can be done by the hardware, compiler or the programmer.



```
#pragma omp parallel for
for (int i = 0; i < nb ; i++ )
{
      Msizeu partstart = partition() [i];
     Msizeu partend = partition() [i+1];

     diag(i) .multply_transposed(... );
     offdiag.multply(...);

     for (int i = partstart; i < partend; i++)
     {
             res[i] *=c;
             normres += std: :abs(res[i]);
     }

      asm( "mp_inst" );
}
```

Identification of Critical Threads

- Every time a core decodes an instruction encoding a meeting point, a thread-private counter (located in the processor frontend) is incremented.
- Critical thread is the one with smallest counter while slack for other threads is calculated as the difference of its counter and the counter of the slowest counter.
- Software Only Identification Mechanism Used : Application is rewritten so that it includes an array of counters indexed by thread identifiers. Each thread increments its own counter every time it arrives the end of the parallel section.
- Current Study inserts meeting points by means of pragma & the counters are implemented in hardware.
  - The compiler translates the pragma into new instruction that once decoded, increments the private hardware counter of the thread.

Using the Criticality Information : 1) Thread Delaying

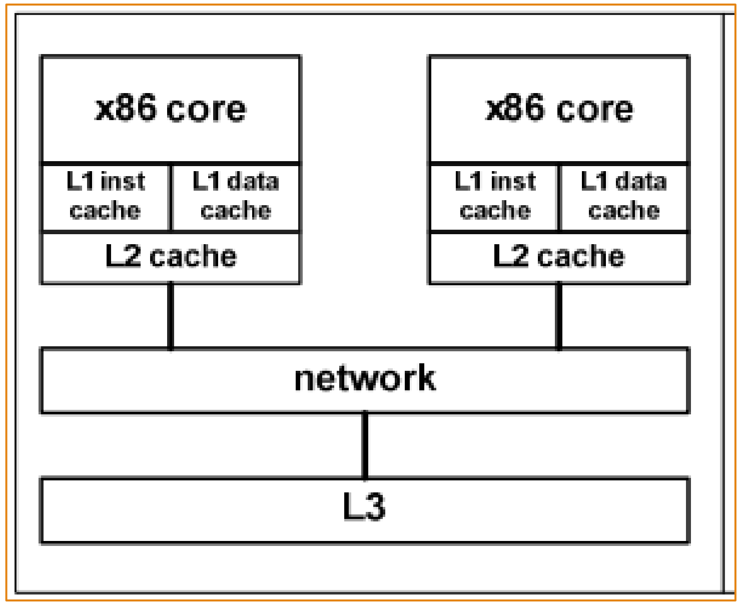
- Assume that critical thread finishes its work in **T** time units and non-critical thread finishes it work in 0.7**T** time units.

- 2 Approaches :
  - 1) Non-critical thread works at full speed for 0.7T time units and then it is put to deep sleep mode with zero energy consumption for remaining 0.3T.
    - Total Power Consumption -> $E_{fmax} = c \times V_{DD}^2 \times f_{max} \times 0.7T$.
  - 2) Core running the non-critical thread can have it frequency scaled-do to 0.7 fmax and it would meet the barrier on time anyway.
    - Total Power Consumption -> $E_{fscaled} = c \times 0.875^2 \times V_{DD}^2 \times 0.7f_{max} \times T = 0.765E_{fmax}$

- Thus DVFS is a clear winner for non-critical threads.

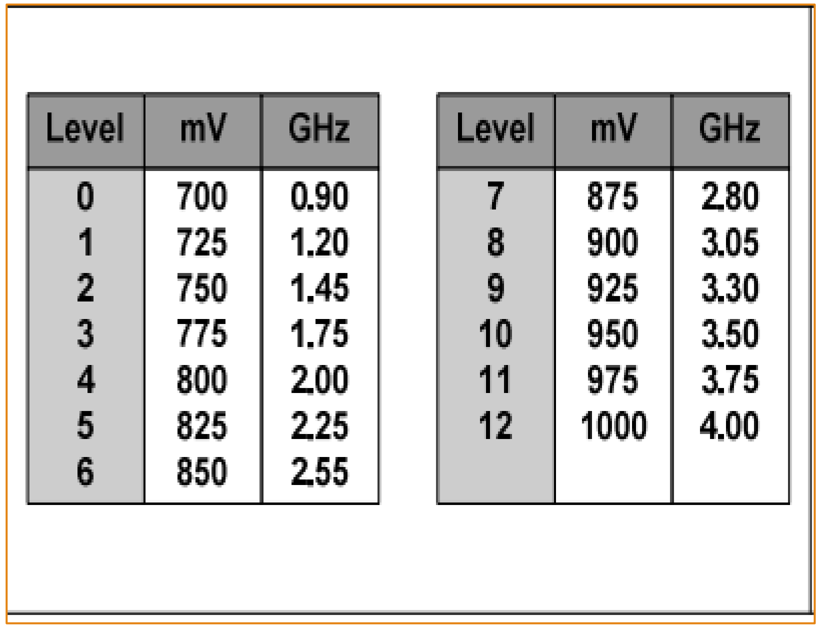
Implementation on CMP with multiple clock domains

- CMP Microarchitectures consists of many Intel64/IA32 cores and each core is single-threaded in-order Core with bandwidth of 2 instructions per cycle.
- Each core contains private first-level instruction cache and data cache and private second-level unified cache. A shared third-level cache is connected to all cores through a bus network. MESI cache controller is used to keep data coherent.
- All caches have their own clock domains. Also L3 cache also has its own clock domain.
- This allows each domain to run at different frequency and secondly adapt the frequency of each domain dynamically and independently of the others.



Voltage-Frequency Table

- Each one of microprocessor domains can operate at a distinct voltage and frequency.
- Voltage and frequency can be changed dynamically and independently for each domain.
- Having few levels allows to switch between them very quickly.
- Each domain includes an on-chip digital clock multiplier connected to the external PLL.
- Frequency changes per domain are effected by changing the multiplication factor of the domain clock multiplier; external PLL frequency is fixed. This allows extremely fast frequency changes.



Level	mV	GHz
0	700	0.90
1	725	1.20
2	750	1.45
3	775	1.75
4	800	2.00
5	825	2.25
6	850	2.55

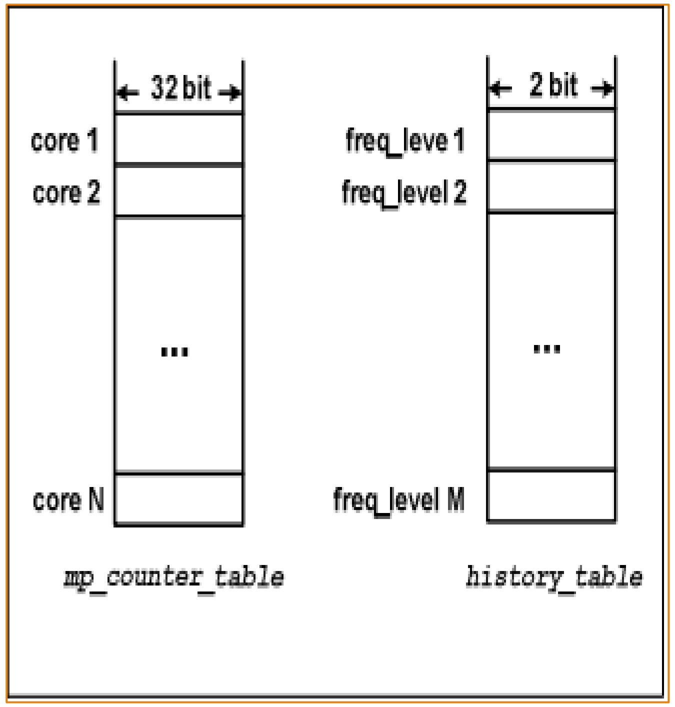
Level	mV	GHz
7	875	2.80
8	900	3.05
9	925	3.30
10	950	3.50
11	975	3.75
12	1000	4.00

Implementation of Thread Delaying

- Each Core contains two tables :

1) MP - Counter – Table
   - Contains as many entries as number of cores in the processor.
   - Each entry contains a 32-bit counter that keeps track of the number of times each core has reached the given meeting point.

2) History Table
   - Contains entry for each possible frequency level.
   - Each entry contains a two-bit up-down saturating counter used to determine the next frequency the core must run at.
   - The table is initialized so that entry corresponding to the maximum frequency level has the highest value.



Working (1/2)

- When a core decodes a meeting point, the counter corresponding to its thread is incremented by 1 in the MP-Counter-Table.
- Core broadcasts the value of counter to other cores at every 10 executions. This is done by special network message.
- When the network interface of a core receives such message, the MP-Counter-Table is accessed to increment by 10. Ten is chosen here since it gives enough precision to the thread delaying with no impact on the interconnect performance.
- For every 10 executions, processor frontend stops fetching instructions and inserts a microcode to execute the thread delaying control algorithm.
- This mircrocode takes MP-Counter-Table and History-Table as an input and its output is frequency f_i for the next interval.

Working (2/2)

- The microcode computes the frequency that better matches the current slack using the formula:
  - $f_{temp} = \frac{C_{critical}}{C_i} \times MAX\_FREQUENCY$
  - $f_i = search\_closest\_valid\_frequency(f_{temp})$

Here $C_{Critical}$ & C_i *are the counters from the critical thread and non-critical thread i.*

- *Once the frequency level for f_i is obtained, the HISTORY-TABLE is updated correctly.*
- *If the frequency level for f_i is k, entry k is incremented and every other entry is decremented.*
- *Finally the frequency chosen by the microcode for the next interval*

  is the one with the largest counter in the HISTORY-TABLE.

Using Criticality Information : 2) Thread Balancing

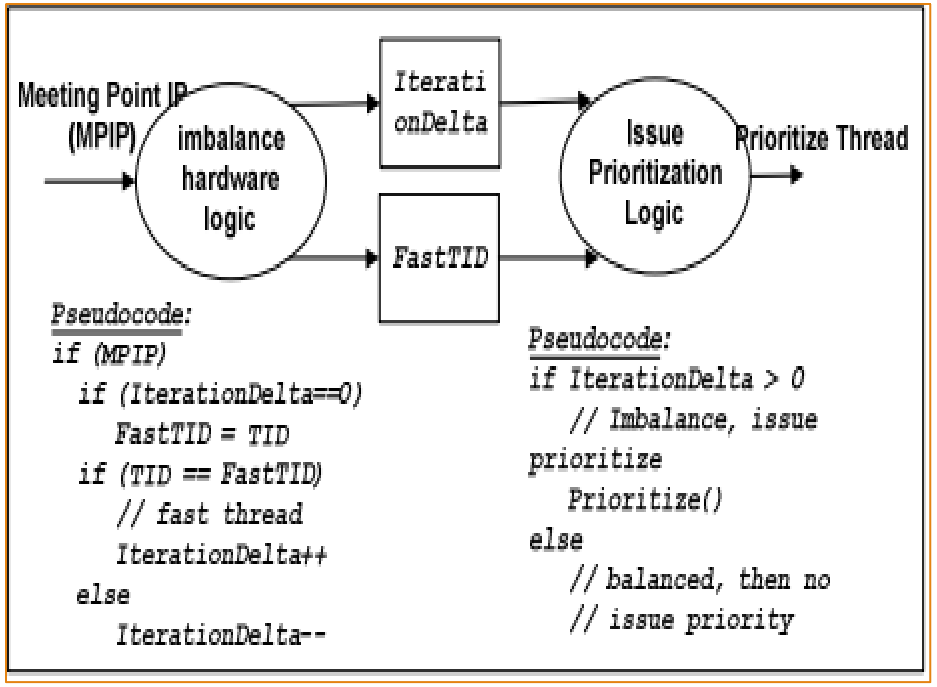
- Speeding up the critical thread for a parallel application running more than one thread on a single 2-way SMT core by accelerating the critical thread.

- Base Issue Logic:
  - If both threads have ready instructions, each one of them is allowed to issue 1 instruction.
  - If one thread has ready instructions and other does not, the one with ready instructions can issue upto two per cycle.

- But if both threads exist on same parallel application, fairness may not be the best option and priority has to be given to the critical thread.

Thread Balancing : Implementation

- Two hardware counters are located in the processor frontend to detect critical thread between two threads running in the same core.
- Counters are compared at every 10 executions and if the difference is greater than given delta, thread with lowest counter value is designated as the critical thread. This info is passed to issue logic.
- If the critical thread has two ready instructions, it is allowed to issue both instructions regardless of the ready instructions the non-critical thread has.



Pseudocode:
```
if (MPIP)
  if (IterationDelta==0)
    FastTID = TID
  if (TID == FastTID)
    // fast thread
    IterationDelta++
  else
    IterationDelta--
```

Pseudocode:
```
if IterationDelta > 0
    // Imbalance, issue
prioritize
    Prioritize()
else
    // balanced, then no
    // issue priority
```

Experiments : Setup (1/2)

- Functional Simulator : SoftSDV for Intel64/IA32. It can simulate not only multithreaded primitives including locks and synchronization operations but also shared memory and events.
- Guest OS : Redhat 3.0 EL
- Less than 1% of simulated instructions are from OS, this minimizing its impact on performance.
- Functional Simulator feeds Intel64/IA32 instructions into performance simulator, which provides cycle accurate simulation.
- **AGGRESSIVE BASELINE : every core is running at full speed and stops when it is completed. Once the core stops it, it consumes zero power.**
- Evaluations from experiments include dynamic energy, idle energy and leakage energy.

Experiments : Setup (2/2)

- Thread delaying is evaluated for Multi-core systems where **each core contains only one thread** while thread balancing is evaluated for a single SMT core (each core contains two threads).
- Simple In-order core is low power and is suitable for a many-core chip such as Sun’s Niagra.

Process Model	In-Order Intel64/32
L1 Instruction Cache (private)	32 KB, 4-ways
L1 Data Cache (private)	32 KB, 8-ways
L2 Cache (Unified & private)	512 KB, 16-ways
L3 Cache (Unified & shared)	8MB, 16-ways
Network Protocol	MESI

Architectural Parameters

Benchmarks used

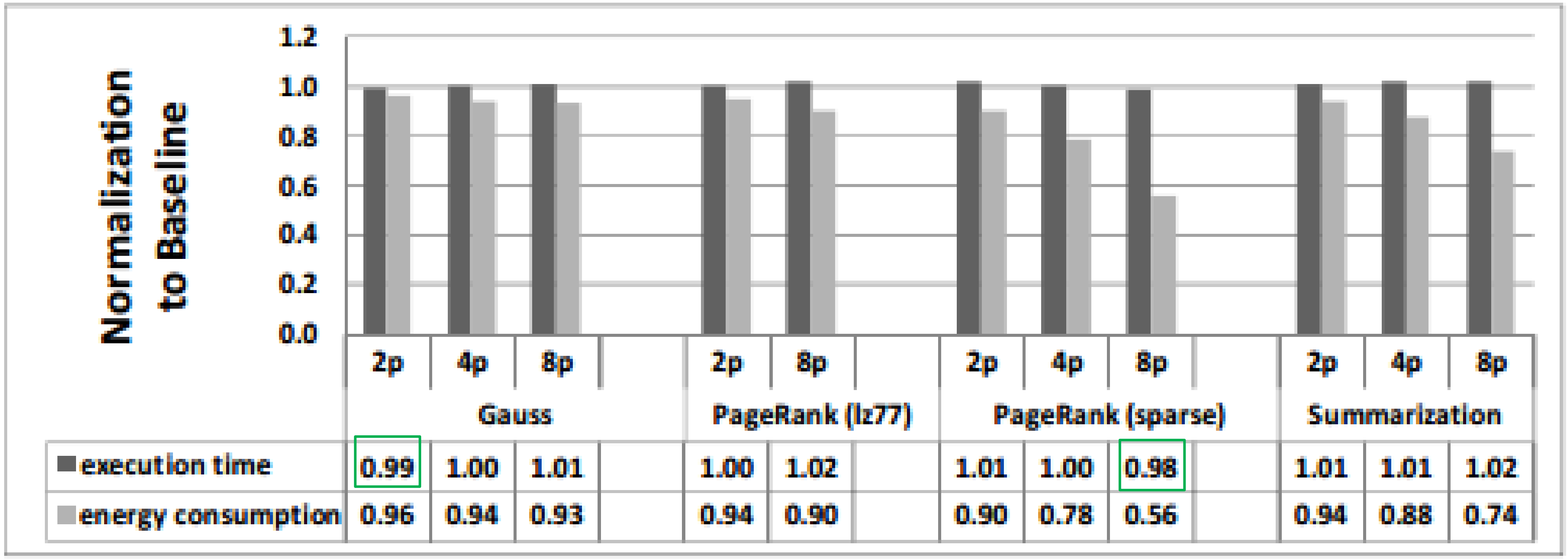
- Recognition, Mining & Synthesis (RMS) workloads from Intel are multi-threaded applications for Tera-Scale systems.
- RMS workload includes highly compute-intensive and highly parallel applications including computer vision, data mining on text and media, bio-informatics and physical simulation.

Benchmarks	Application
Gauss	Financial Analysis
PageRank (sparse)	Search Engine
PageRank (lz77)	Search Engine
Summarization	Text Data Mining
FIMI	Data Mining
Research	Bioinformatics
SVM	Bioinformatics

Characteristics of Benchmarks

- All RMS benchmarks except Gauss show workload imbalance. Gauss is chosen for testing robustness of thread delaying algorithm.
- All workloads are parallelized by using OpenMP to achieve maximal scalability but they still exhibit different degrees of workload imbalance and therefore inefficiency in the energy consumption.
- The simulated section for each benchmark is chosen by first profiling its single-threaded counterpart and then selecting the hottest region, which is a parallel loop.
- Each thread runs a fixed number of iterations and when slowest thread has executed N iterations, the simulation has finished.
- At least 100 million instructions are executed for simulation.

Performance Results for Thread Delaying (1/2)

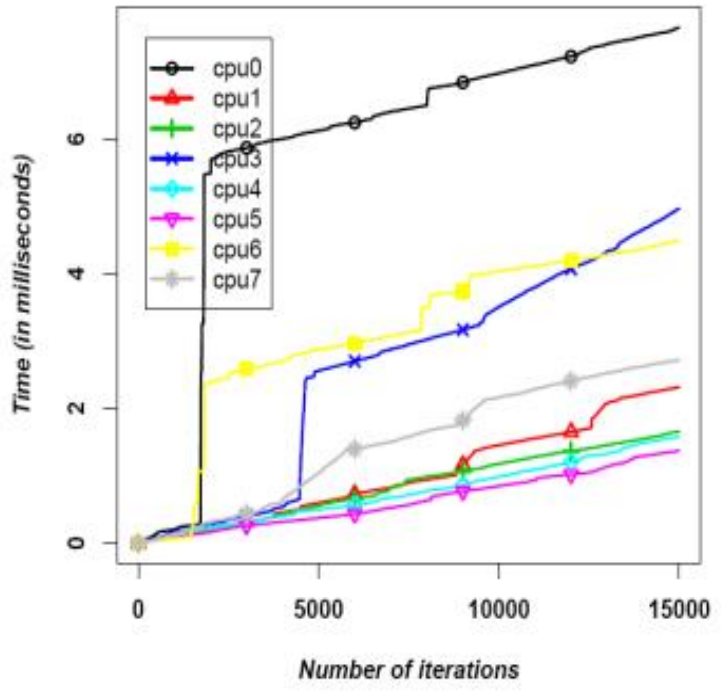


	Gauss				PageRank (lz77)			PageRank (sparse)				Summarization		
	2p	4p	8p		2p	8p		2p	4p	8p		2p	4p	8p
■ execution time	$\boxed{0.99}$	1.00	1.01		1.00	1.02		1.01	1.00	$\boxed{0.98}$		1.01	1.01	1.02
▨ energy consumption	0.96	0.94	0.93		0.94	0.90		0.90	0.78	0.56		0.94	0.88	0.74

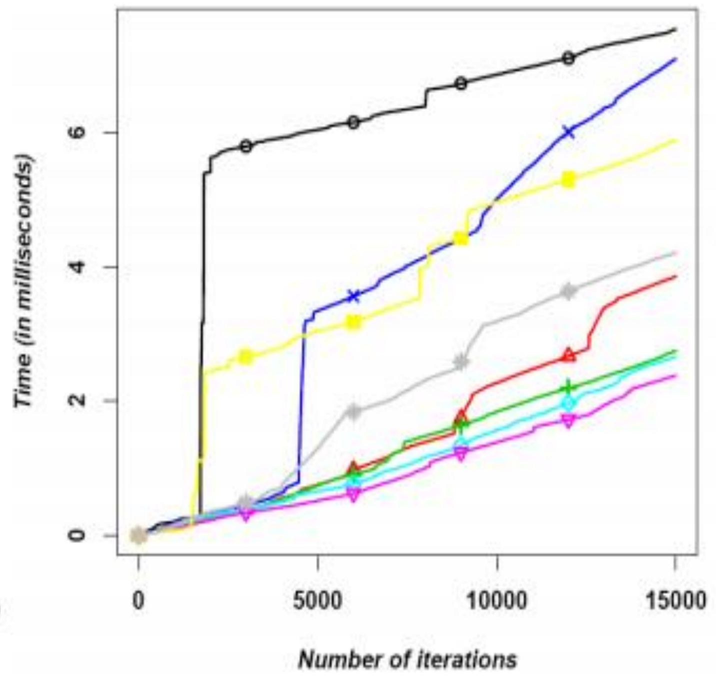
Performance Results for Thread Delaying (2/2)

- Thread Delaying achieves significant energy reductions for selected RMS benchmarks under three different hardware configurations : two, four & eight cores ranging from 4% to 44% energy savings.
- For most configurations, there is little performance loss, ranging from 1% to 2%.
- In couple of cases, thread delaying even obtains speedups !!
  - All cores except the critical thread are more spread out over time, allowing the critical thread to have more priority in the interconnection.
  - This side effect of per-core DVFS accelerates the critical thread and thus reduces the total execution time.

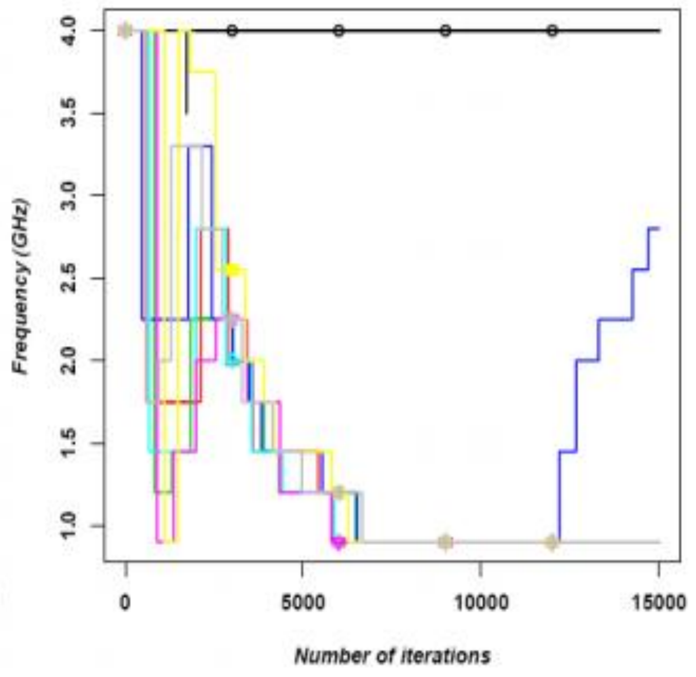
Results Analysis (1/4)



(a)



(b)

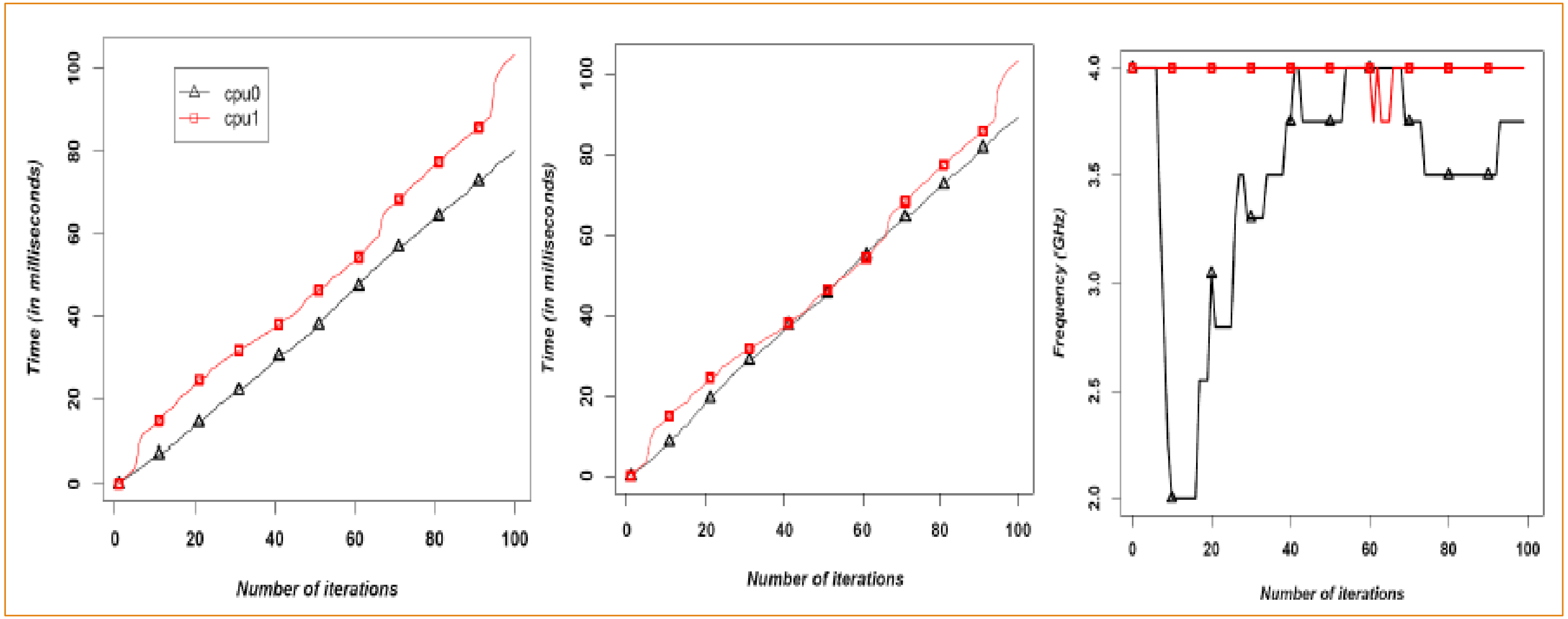


(c)

Results Analysis (2/4)

- X-axis represents number of iterations of the parallelized loop that each core executes.
- Y –axis of figure (a) & (b) represent cumulative execution time of loop while figure (c) represents frequency of the core.
- Large gaps between critical thread (cpu0) and rest of the threads.
- All non-critical threads except cpu3 stay at lowest frequency after iteration 6600.
- Big Energy savings come from large frequency decreases on non-critical threads.
- For cpu3, it says at the lowest frequency until iteration 12200 and increases, because the gap between cpu0 and cpu3 is getting smaller.

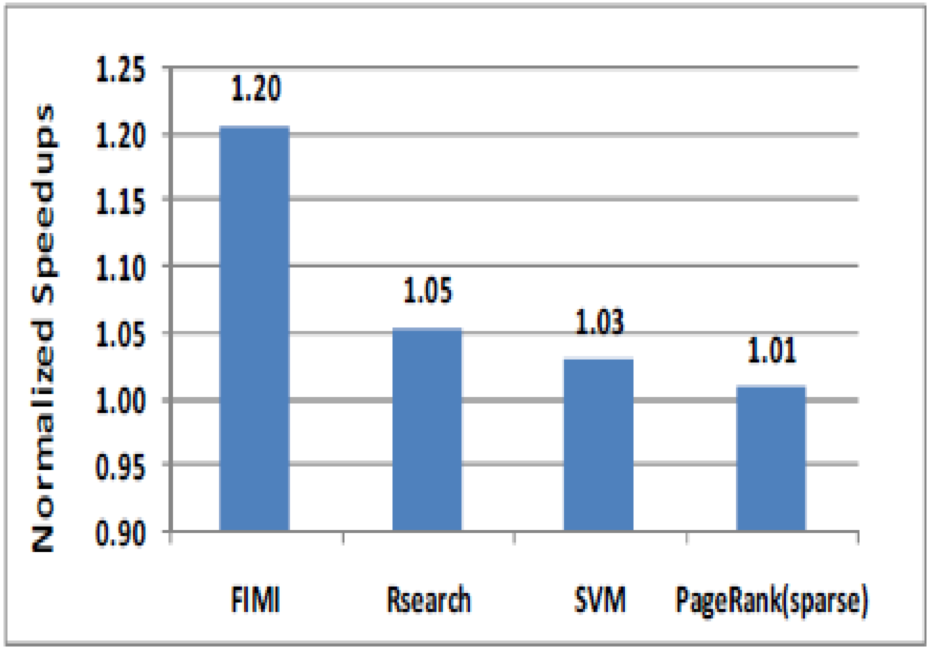
Results Analysis (3/4)



Results Analysis (4/4)

- The effectiveness of thread delaying depends on whether the algorithm can quickly adapt at runtime.
- Between Iteration 10 & 40, the time gap becomes smaller and smaller & algorithm increments the frequency of the non-critical threads slowly.
- At iteration 65, there is cache miss with latency, which results in a time difference between two threads again.
- The algorithm immediately observes the change and starts to decrement the frequency level of non-critical thread.
- Thread delaying algorithm saves reasonable amount of energy for relatively balanced workloads.
  - Gauss is balanced workload and it is hard to distinguish which threads are critical or non-critical.
  - Still, 6 % energy savings are obtained without any performance penalty.

Performance Results : Thread Balancing



Benchmark Name	Opportunity %	Correction %
FIMI	30	100
Research	24	49
SVM	56	100
PageRank (sparse)	4	1

Results Analysis (1/2)

- Performance benefit for the Four RMS workloads ranges from 1 to 20 %.
- PageRank shows huge imbalance during parallel execution and thus we have large amount of energy savings from thread delaying.
- But due to cache misses, Thread Balancing cannot give much performance improvement to PageRank.
- Prioritizing the issue of slow thread results in a shift in pipeline stalls from issue stage to the backend of the in-order core because this benchmark suffers from a significant amount of load misses.

Thus performance of slow threads is not greatly improved.

Results Analysis (2/2)

- FIMI has a large level of thread imbalance and a corresponding amount of performance improvement by administering issue priority to the slower thread. Thus performance benefit correlates with imbalance levels.
- Using the algorithm described earlier about giving the priority to the slower thread, FIMI & SVM have the most opportunity to give priority to the slow threads.
- Table shows the percentage of number of iterations that are caught up by the slow thread with thread balancing method.
- FIMI and SVM have 100% imbalance correction with the algorithm and are operating in an ideal situation.

Shortfalls !

- Thread Delaying is ineffective for SMTs for cores having multiple threads since the approach assumes only one thread per core.
- Also Thread Delaying applies DVFS at core level instead of thread level. So even if it’s applied to SMT, both critical & non-critical threads in the same core will run at same f-V resulting in no energy savings.
- In some cases when aggressive DVFS is applied, although energy savings are achieved there are performance losses.
- Study could have covered more benchmarks having balanced workloads (only Gauss is considered here) and applied Thread delaying & Thread balancing techniques to see the effect on energy savings and speedups.

Conclusion

- MPTC estimates the criticality of thread dynamically in a parallel execution which is used in thread delaying & thread balancing to save energy and improve performance.
- Thread Delaying combines per-core DVFS & MPTC in CMPs to reduce energy consumption on non-critical threads and achieves up to 40% energy savings without performance loss for different core configurations.
- Thread Balancing gives higher priority in issue queue of an SMT core to critical thread & improves performance for various RMS workloads ranging from 1% to 20%.

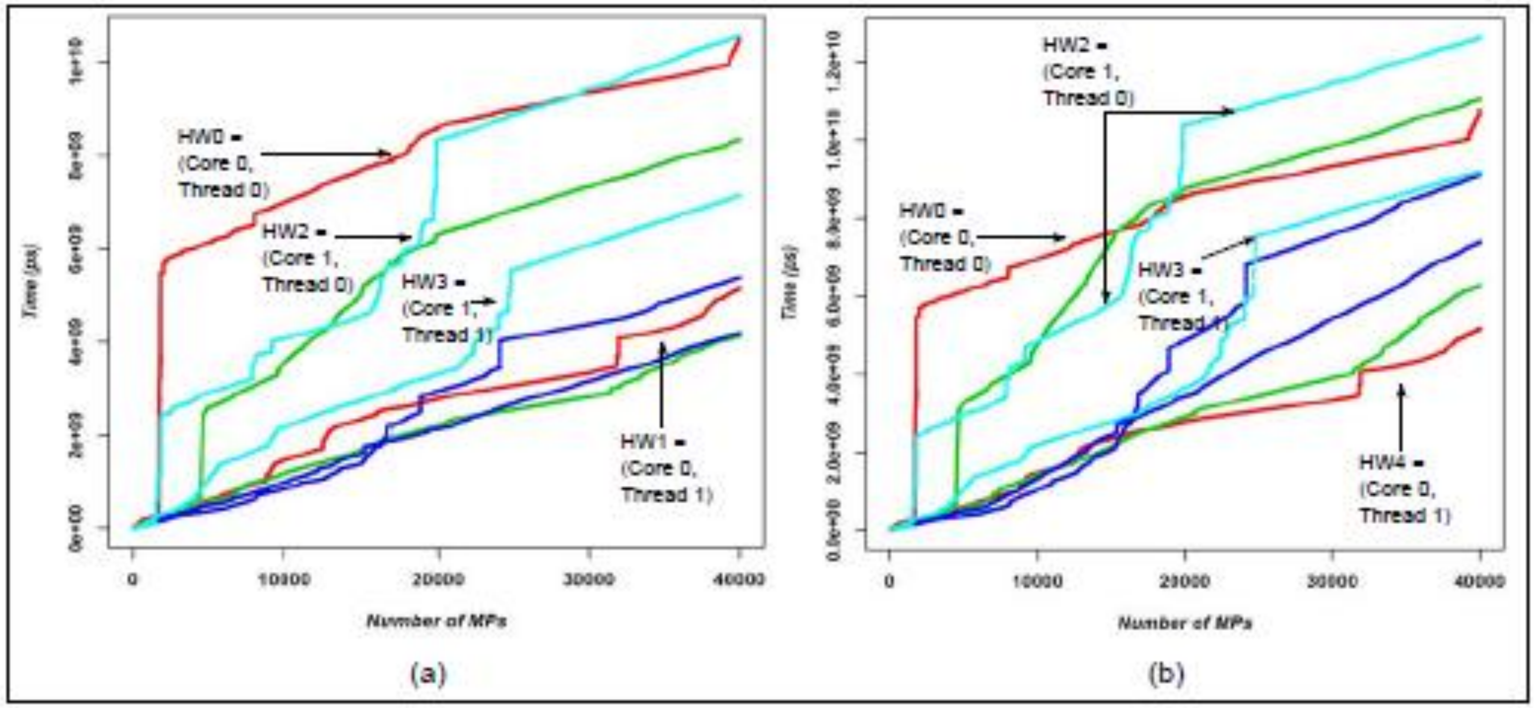
Questions??

Thread Shuffling : Combining DVFS & Thread Migration to Reduce Energy Consumption for Multi-core Systems.

QIONG CAI, JOSE GONZALEZ, GRIGORIOS MAGKLIS, PEDRO CHAPARRO & ANTONIO GONZALEZ, ISLPED '11 PROCEEDINGS OF THE 2011 INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN PAGES 379-384

Thread Delaying : Ineffective for SMTs

- Thread Delaying assumes each core can only execute one thread at a time.
- The local DVFS is applied at the core level instead of the thread level.



Example

- Run time behavior of hottest region in PageRank-sparse.
- Baseline contains four SMT cores and each core has two hardware contexts. This allows two threads to run at the same time.
- Each hardware context (HW0, HW1, HW2, HW3) is specified by its core identifier and thread identifier. HW0 denotes a hardware context in core 0 & thread 0.
- HW0 is the critical hardware context most of the time while HW1 is a non-critical hardware context all the time.
- If core contains only one hardware context, then the gap between HW0 & HW1 can be reduced after thread delaying is applied.
- But since they are on same core, when thread delaying is applied there are no energy savings as seen in second figure as the gap between them remains the same.

Another Drawback & Results

- Difficult to recover performance loss when aggressive voltage and frequency scaling is applied at the beginning of execution time.
- Slack between HW0 and HW2 is large at the beginning.
- When aggressive DVFS is applied, slack becomes very small.
- Under this situation, it’s difficult for thread delaying to react and change the voltage and frequency level of core 1 back to maximal one. This causes big performance loss.
- For PageRank example, thread delaying has 30% energy reduction but 14 % performance loss.

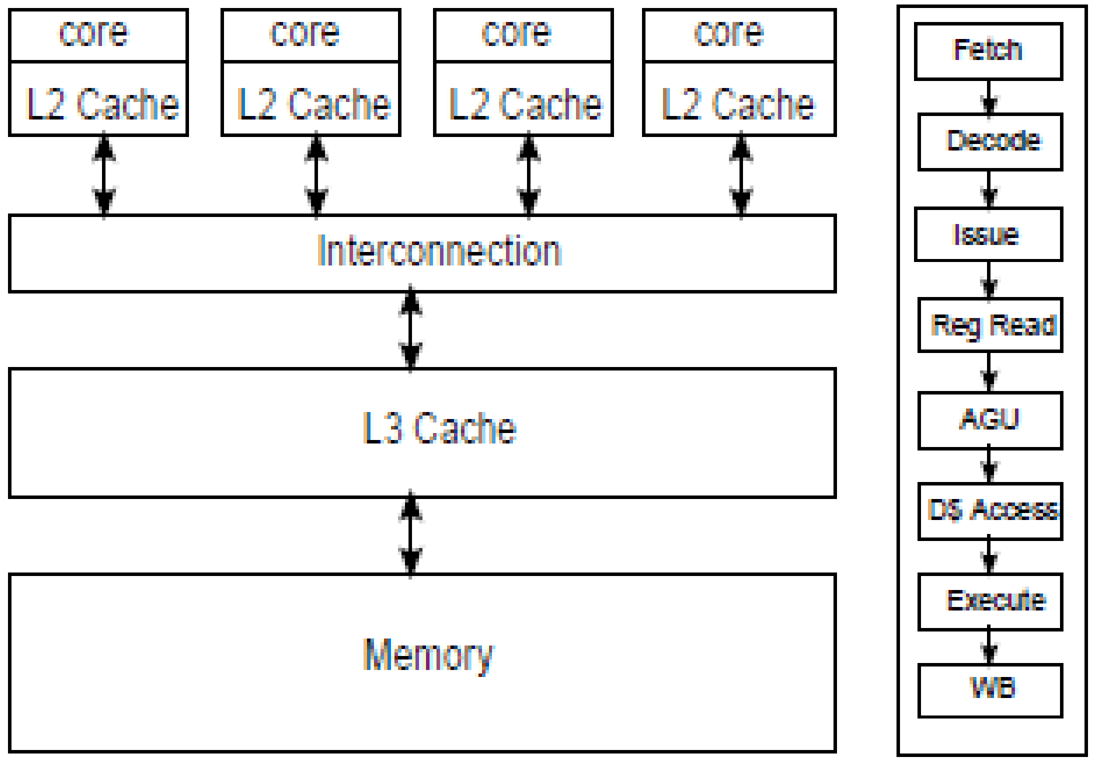
Solution : Thread Migration

- Addresses the Non-Optimal core-level DVFS and aggressive DVFS problems in SMTs.
- Idea : To map threads with similar criticality degrees into the same core through thread migration and then apply DVFS to cores containing non-critical threads.
- As in MPTC, criticality of thread is approximated as difference between its own counter and the counter of the slowest one.
- Criticality degree of thread is the position of its own counter in a reverse sorted sequence of all counters.
- If counter values are 200, 100, 300, 400 for thread 0, 1 , 2 & 3, then thread 1 is the critical thread while thread 3 is the most non-critical thread.

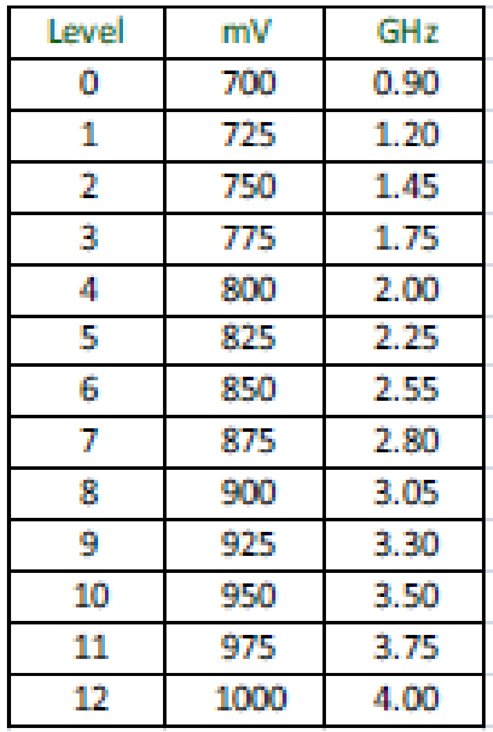
Example

- Thread shuffling addresses the problem of non-optimal DVFS in thread delaying by first grouping threads with similar criticality degrees into the same core and then apply DVFS.
- HW0 & HW2 contexts can be mapped into the same core at runtime.
- This remapping increases the chances of reducing gap between HW0 and HW2 and thus the energy reduction increases.

Multi-Core System with Multiple Clock Domains (1/2)



Block Diagram & Pipeline of SMT X86 core



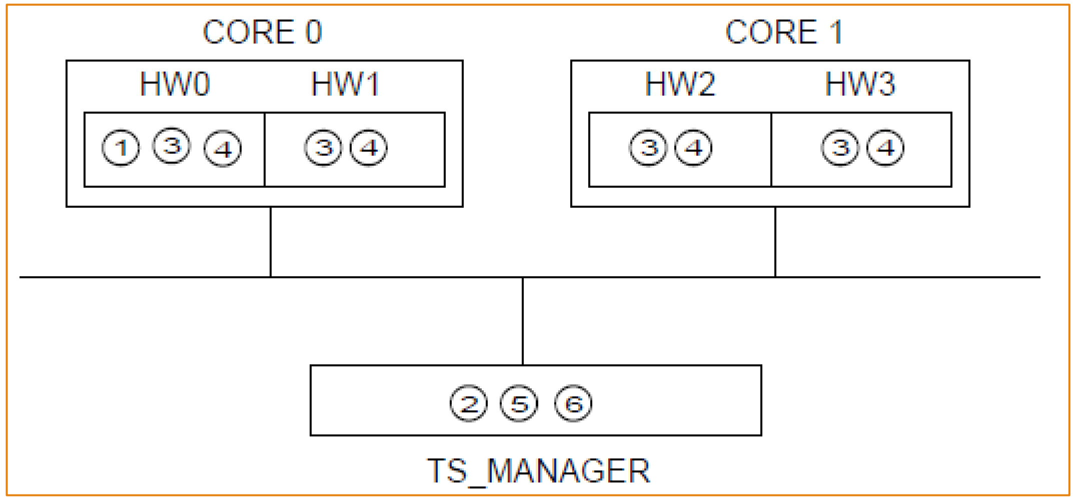
Level	mV	GHz
0	700	0.90
1	725	1.20
2	750	1.45
3	775	1.75
4	800	2.00
5	825	2.25
6	850	2.55
7	875	2.80
8	900	3.05
9	925	3.30
10	950	3.50
11	975	3.75
12	1000	4.00

Voltage-Frequency Table

Multi-Core System with Multiple Clock Domains (2/2)

- The system consists of multiple Intel64/IA32 cores. The pipeline of in-order SMT core is similar to Intel ATOM core.
- Each core has their own L1 & L2 cache which have their own separate clock domains.
- Unified L3 cache has a separate clock domain with an interconnect.
- Each clock domain has its own local clock network that receives a reference clock signal as input and distributes it to all circuits of the domain.
- Domains operate asynchronously; so interconnect communication can be synchronized correctly to avoid meta-stability.
- Mixed-clock FIFO design is used to communicate values safely between domains.

Thread Shuffling Algorithm



- Algorithm is implemented in a centralized hardware manager called TS_Manager which coordinates jobs between hardware contexts and itself.
- It then applies Thread migration and DVFS to hardware contexts and cores respectively.

Algorithm Steps

1) An Instruction encoding a meeting point is fetched in a hardware context say HW0. A TS_Request is sent to TS_Manager

2) Counters in MP_Counter_Table are sorted if necessary. TS_BK messages are sent to cores containing HW candidates for thread migration. In this example, HW0 and HW2 are candidates for thread migration. So Core1 & Core 0 need to do bookkeeping for thread migration.

3) Cores 1 & 0 do draining pipelining and save architectural states.

4) TS_BK_DONE message is sent to TS_Manager

5) Thread migration is performed & TS_MIGRATION_DONE messages are sent to cores containing HW candidates for thread migration.

6) DVFS is applied for all non-critical cores.

Condition for Thread Migration

- After TS_Manager receives shuffling request from hardware context, it checks whether following condition is true or not :

(current cycle – last_config_cycle) > MAX_CONFIG_INTERVAL

current_cycle : current cycle when manager receives request.
Last_config_cycle : cycle when last shuffling is performed &
max_config_interval : parameter to adjust frequency of thread shuffling.

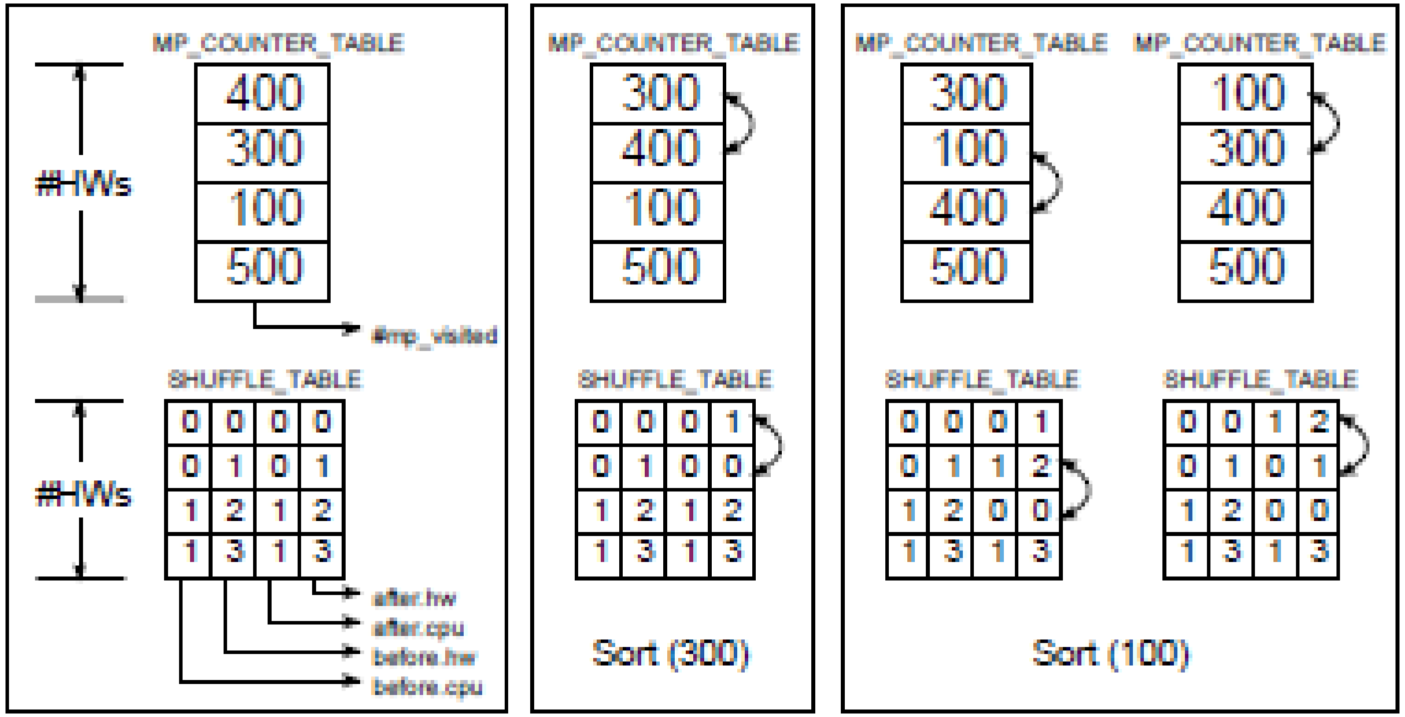
- max_config_interval is set to 5 million cycles, since thread migration is fine-grained and light-weight to be implemented in hardware.

- If above condition is true, manager starts a new thread shuffling.

Sorting Thread & Thread Migration (1/2)

Algorithm for Sorting Threads

- Reset the Shuffling table
- Apply insertion sort to MP_Counter_table and update SHUFFLE_TABLE correspondingly
- Send TS_Bookkeeping message to HW if ***SHUFFLE_TABLE[w].before cpu ! = SHUFFLE_TABLE[w].after.cpu***.



Sorting Thread & Thread Migration (2/2)

- After sorting is done, manager sends a TS_BK message to a hardware context if the following condition is true :

SHUFFLE_TABLE[hw].before.cpu != SHUFFLE_TABLE[hw].after.cpu

- **SHUFFLE_TABLE shows that hardware context 0 in cpu 0 will be replaced by hardware context 2 in CPU 1.**
- **Hardware context 0 and hardware context 2 need to be swapped to make the counters in MP_Counter_Table sorted.**
- **Above step is performed during thread migration when thread migration bookkeeping is finished.**

Local DVFS on non-critical Cores (1/2)

- For each core containing non-critical threads, algorithm computes a scaling factor based on two counter ref_counter and cmp_counter.
- ref_counter is largest counter value in critical core (Here it’s Core 0).
- cmp_counter is smallest counter value in a non-critical core.
- Scaling factor for frequency is calculated by taking ratio of ref_counter to cmp_counter.
    - In the example, ref_counter = 300, cmp_counter = 400. So scaling factor = 0.75.
    - There are 13 frequency levels and scaling factor is multiplied with 13 which results to level 10.

Local DVFS on non-critical Cores (2/2)

- Internal HISTORY_TABLE in the manager is updated after frequency is obtained.
- Each entry of the table contains a two-bit up-down saturating counter.
- The final frequency level for the next interval is the one with largest counter in HISTORY_TABLE.
- The purpose of HISTORY_TABLE is to reduce effect of temporal noise in estimation of slack.
- Final step of DVFS algorithm is to find out voltage for corresponding frequency from the voltage-frequency table.
  - Voltage 0.95 V and frequency 3.5 Ghz will be selected for core 1 in next interval of time.

Experiments : Setup (1/2)

- Functional Simulator : SoftSDV for Intel64/IA 32. It can simulate multithreaded primitives including locks, synchronization operations, shared memory & events. OS is Redhat 3.0 EL.
- Less than 1% of simulated instructions are from operating system and thus impact of the OS is negligible.
- Functional Simulator feeds Intel64/IA 32 instructions into the performance simulator.
- Performance Simulator : x86. It uses power model based on activity counters and energy access.
- **Aggressive Baseline : Assume Core is running at full speed and stops when it's completed. Once core stops, it consumes zero power.**

Experiments : Setup (2/2)

Parameter	Value
Processor	In-order x86 core, 2-way SMT
L1 Instruction Cache (private)	32 KB, 4-way, 64B
L1 Data Cache (private)	32 KB, 4-way, 64B
L2 Cache (unified & private)	512 KB, 16-way 64B
L3 Cache (Unified & shared)	8 MB, 16-way, 64B
Memory	Always hit, 500 cycles access penalty
Network Protocol	MESI

Architectural Parameters

Benchmarks

- Recognition, Mining & Synthesis benchmarks which are highly compute-intensive and highly parallel applications including data mining on text, media, bio-informatics and search engine.
- Benchmarks that clearly show workload imbalance are chosen for analysis.

Benchmark	Application Domain
PageRank-lz77	Search Engine
PageRank-sparse	Search Engine
Rsearch	Bioinformatics
Summarization	Text Data Mining

Characteristics of Benchmarks

- Kernel of PageRank performs multiple matrix multiplications on a large and sparse matrix. The matrix can be stored in memory either in a native sparse format or a compresses version.
- Compression is a simplified LZ77-based method. Research is used in bioinformatics to search a homologous RNA in a database.
- The simulated section for each benchmark is chosen by first profiling its single-threaded counterpart and then selecting the hottest region, which is a parallel loop.
- Each thread runs a fixed number of iterations and when slowest thread has executed N iterations, the simulation has finished.
- At least 100 million instructions are executed for simulation.

Performance Results (1/2)

Benchmarks	Thread Delaying vs Baseline		Thread Shuffling vs Baseline		Thread Shuffling vs Thread Delaying	
	Execution Time	Energy Consumption	Execution Time	Energy Consumption	Execution Time	Energy Consumption
PageRank-lz77	1.00	0.93	0.99	0.90	0.99	0.97
PageRank-sparse	1.14	0.71	0.98	0.49	0.86	0.69
Rsearch	1.01	0.97	0.98	0.97	0.97	1.00
Summarization	1.00	0.82	1.00	0.76	1.00	0.93

Performance Results of Thread Shuffling on 4 cores

Note : Baseline configuration is whenever a thread reaches the barrier, thread is put to sleep mode and consumes zero power.

Performance Results (2/2)

Benchmarks	Thread Delaying vs Baseline		Thread Shuffling vs Baseline		Thread Shuffling vs Thread Delaying	
	Execution Time	Energy Consumption	Execution Time	Energy Consumption	Execution Time	Energy Consumption
PageRank-lz77	1.02	0.93	1.02	0.93	1.00	1.00
PageRank-sparse	1.29	0.65	0.99	0.50	0.77	0.77
Rsearch	1.00	0.94	0.96	0.92	0.94	0.98
Summarization	1.00	0.79	1.00	0.73	1.00	0.92

Performance Results of Thread Shuffling on 8 cores

Results Analysis

- On 4 Cores, thread shuffling can obtain up to 51% energy savings w.r.t. baseline and achieve 31% energy savings w.r.t. thread delaying.
- Thread shuffling is robust and does not cause any performance slowdown across benchmarks, whereas thread delaying causes 14% & 29% slowdowns.
- Thread shuffling is scalable.
  - Performance in terms of execution time and energy consumption with respect to baseline and thread delaying is maintained when the number of cores is increased from 4 to 8.

Thread Shuffling : 3 Configurations

- By choosing largest counter value in the critical core for ref_counter and smallest counter value in the non-critical core for cmp_counter, Scaling factor is around one.
- If the Scaling factor is small, it's difficult to recover performance loss when DVFS is applied aggressively.
- Using statistical methods like maximum, minimum and average, there are nine possible combinations of ref_counter & cmp_counter. But the performance loss is huge when maximum and average methods are used cmp_counter. Only minimum method is used for cmp_counter.
- So only three configurations are used.
  - Minimum of counters in critical core and maximum counters in non-critical core.
  - Aggressive configuration where maximum of counters of both critical & non-critical cores are used.
  - Combination of average values of counters in critical core and maximum of counters in a non-critical core.

Performance Results

Benchmarks	Thread Shuffling		Thread Shuffling - Aggressive		Thread Shuffling – Midpoint	
	Execution Time	Energy Consumption	Execution Time	Energy Consumption	Execution Time	Energy Consumption
PageRank-lz77	0.99	0.90	1.03	0.87	1.03	0.87
PageRank-sparse	0.98	0.49	1.06	0.43	1.01	0.44
Rsearch	0.98	0.97	1.00	0.95	0.99	0.95
Summarization	1.00	0.76	1.02	0.71	1.01	0.70

Performance Results : Analysis

- Aggressive version achieves best energy reduction. For PageRank-sparse, it reduces up to 57% energy consumption with respect to baseline.
- But aggressive version also has 6% performance slowdown for the same benchmark.
- For other 3 benchmarks, performance loss is negligible for aggressive version.
- Midpoint version achieves best balance between performance loss and energy consumption reduction. It achieves energy consumption reduction up to 56% with negligible performance penalty.

Shortfalls !

- Thread Shuffling assumes that parallel section is statically scheduled.
  - It cannot be used for benchmarks containing loops with variable iteration times.
- Technique is implemented in hardware targeting multi-core systems. Extension to using Software level scheduling algorithm based on DVFS could have been explored for reducing energy consumptions.
- Study like the earlier one could have covered benchmarks with balanced workloads to understand the effect of algorithm on energy savings.

Conclusion

- Thread Shuffling overcomes the shortcomings of Thread Delaying using thread migration and then applying DVFS to non-critical cores.
- Technique is found to be effective for several RMS applications with energy savings up to 56% with respect to baseline & up to 38% with respect to thread delaying without any performance loss on 4 cores.
- Thread Shuffling is scaling since similar energy savings are obtained when number of cores are increased from 4 to 8.

Questions ??

Thank You !