Paper 5: Energy-Efficient Work-Stealing Language Runtimes

Review

Dattatreya Mohapatra[1] Viraj Parimi[2]

[1]2015021

[2]2015068

FPP, Winter 2018

Outline

Outline

Outline

- Dynamic Voltage and Frequency Scaling (DVFS) – a framework to change the frequency and/or operating voltage of a processor(s) based on system requirements.

- Dynamic Voltage and Frequency Scaling (DVFS) – a framework to change the frequency and/or operating voltage of a processor(s) based on system requirements.
- Uses CPUFreq, a linux kernel framework, that decides on whether to increase/decrease frequency of a processor(s).

Background
DVFS

- Dynamic Voltage and Frequency Scaling (DVFS) – a framework to change the frequency and/or operating voltage of a processor(s) based on system requirements.
- Uses CPUFreq, a linux kernel framework, that decides on whether to increase/decrease frequency of a processor(s).
- Consists of two elements:
    - Governor : Directs the driver to change frequency when required.
    - Driver : Performs actions based on governor's decision.

Outline

Background

Work-Stealing

- Work-Stealing
  - Work-First Principle
  - Deque Management
  - Work-Stealing Scheduler
- Covered in course lectures.

Outline

Problem Statement

Objective

Improve energy efficiency of work stealing runtime with minimal performance loss

Outline

Related Work

- Most of the related work can be summarized in two more established areas.

Related Work

- Most of the related work can be summarized in two more established areas.
- Optimization of work-stealing runtimes
  - *A-Steal*: An adaptive scheduler to take parallelism feedback into account.
  - *SLAW*: Adds adaptive scheduling policies based on locality information.
  - *AdaptiveTC*: Improves system performance through adaptive thread management.
  - *BWS*: Improves system throughput and fairness in time-sharing multicore systems.

Related Work

- Most of the related work can be summarized in two more established areas.
- Optimization of work-stealing runtimes
  - *A-Steal*: An adaptive scheduler to take parallelism feedback into account.
  - *SLAW*: Adds adaptive scheduling policies based on locality information.
  - *AdaptiveTC*: Improves system performance through adaptive thread management.
  - *BWS*: Improves system throughput and fairness in time-sharing multicore systems.
- Energy efficiency of multi-threaded programs
  - Various DVFS-based solutions have been proposed earlier.
  - *Magklis et. al.* designed a profiling-based DVFS algorithm on CPUs with multiple clock domains.
  - *Wu et. al.* designed a DVFS-based strategy where the interval of DVFS use is adaptive to recent instance issue queue occupancy.

Related Work

- Most of the related work can be summarized in two more established areas.
- Optimization of work-stealing runtimes
  - *A-Steal*: An adaptive scheduler to take parallelism feedback into account.
  - *SLAW*: Adds adaptive scheduling policies based on locality information.
  - *AdaptiveTC*: Improves system performance through adaptive thread management.
  - *BWS*: Improves system throughput and fairness in time-sharing multicore systems.
- Energy efficiency of multi-threaded programs
  - Various DVFS-based solutions have been proposed earlier.
  - *Magklis et. al.* designed a profiling-based DVFS algorithm on CPUs with multiple clock domains.
  - *Wu et. al.* designed a DVFS-based strategy where the interval of DVFS use is adaptive to recent instance issue queue occupancy.
- Other approaches include thread migration, hardware/software approximation, or a combination of all of them.

Outline

Contributions

- HERMES – The first framework addressing energy efficiency in work-stealing systems.

Contributions

- HERMES – The first framework addressing energy efficiency in work-stealing systems.
- Programs are *tempo-enabled* : Different threads may execute at different speeds(*tempo*). Two novel, complementary tempo control strategies:
  - workpath-sensitive
  - workload-sensitive

Contributions

- HERMES – The first framework addressing energy efficiency in work-stealing systems.
- Programs are *tempo-enabled* : Different threads may execute at different speeds(*tempo*). Two novel, complementary tempo control strategies:
  - workpath-sensitive
  - workload-sensitive
- A prototyped implementation demonstrating an average of 11-12% energy savings with 3-4% performance loss over work-stealing benchmarks.

Outline

Motivation

- In the multi-core era, work stealing received considerable interest in language runtime design.

Motivation

- In the multi-core era, work stealing received considerable interest in language runtime design.
- There is an active interest in research improving its performance-critical properties, such as adaptiveness, scalability, and fairness.

Motivation

- In the multi-core era, work stealing received considerable interest in language runtime design.
- There is an active interest in research improving its performance-critical properties, such as adaptiveness, scalability, and fairness.
- In comparison, energy efficiency in work-stealing systems has received little attention, which is particularly unfortunate.

Outline

Outline

- Determines thread tempo based on control flow, with threads tackling *immediate work* executing at a faster tempo.

- Determines thread tempo based on control flow, with threads tackling *immediate work* executing at a faster tempo.
- Victim worker takes precedence over the thief worker in a thief-victim relationship.

- Determines thread tempo based on control flow, with threads tackling *immediate work* executing at a faster tempo.
- Victim worker takes precedence over the thief worker in a thief-victim relationship.
- Two important design ideas:
  - Thief Procrastination
  - Immediacy Relay

Insights

Workpath-Sensitive Tempo Control

Thief Procrastination

At the beginning of the thief-victim relationship, the tempo of the thief worker should be set to be slower than the victim worker.

Immediacy Relay

If the thief-victim relationship terminates because the victim runs out of work, the tempo of all the thiefs should be raised. This happens in a recursive fashion.

Outline

- Selects the appropriate thread tempo based on the size of work-stealing deques.

Insights

Workload-Sensitive Tempo Control

- Selects the appropriate thread tempo based on the size of work-stealing deques.
- Threads with longer deques run at faster tempo.

- Selects the appropriate thread tempo based on the size of work-stealing deques.
- Threads with longer deques run at faster tempo.
- Threshold calculation:

$$thld_i = \left(\frac{2 \times L}{K + 1}\right) \times i$$

  - $L =$ average deque size
  - $K =$ number of thresholds
  - $i \in [1, K]$

Outline

Insights
Unified Algorithm

- In workpath-alone executions, a non-immediate worker may have many deque items to work on.

- In workpath-alone executions, a non-immediate worker may have many deque items to work on.
- Similarly, in workload-alone executions, an immediate worker with fewer deque items is set to a lower tempo.

Insights

Unified Algorithm

- In workpath-alone executions, a non-immediate worker may have many deque items to work on.
- Similarly, in workload-alone executions, an immediate worker with fewer deque items is set to a lower tempo.
- Increased workpath length means increased execution time, which potentially leads to additional energy consumption.

Insights
Unified Algorithm

- In workpath-alone executions, a non-immediate worker may have many deque items to work on.
- Similarly, in workload-alone executions, an immediate worker with fewer deque items is set to a lower tempo.
- Increased workpath length means increased execution time, which potentially leads to additional energy consumption.
- In both cases, a second opinion from the other strategy can lead to better decisions.

Outline

Outline

Implementation

Order of Immediacy

- Key data-structure for workpath sensitivity.
- Double linked list across workers connected by `next` and `prev` pointers.
- When a victim terminates, immediacy relay happens through this linked list.

Implementation
Order of Immediacy

structure WORKER
  DQ // deque (array)
  H // head index
  T // tail index
end structure

structure WORKER
  DQ // deque (array)
  H // head index
  T // tail index
  next // next immediate work
  prev // prev immediate work
  thld // size thresholds (array)
  S // size threshold index
end structure

Outline

- Achieved through DVFS as modern CPUs support a limited and discrete set of frequencies.

- Achieved through DVFS as modern CPUs support a limited and discrete set of frequencies.
- Let $\{f_1, f_2, ..., f_n\}$ be set of frequencies supported by the CPU core, where $f_i > f_{i+1}$ for any $i \in [1, n-1]$

```
procedure DOWN(w, v)
    f ← frequency of core hosting v
    if f == f_i and i < N then
        ...// scale core hosting w to f_{i+1}
    end if
end procedure
```

- Achieved through DVFS as modern CPUs support a limited and discrete set of frequencies.
- Let $\{f_1, f_2, ..., f_n\}$ be set of frequencies supported by the CPU core, where $f_i > f_{i+1}$ for any $i \in [1, n-1]$

```
procedure DOWN(w, v)
    f ← frequency of core hosting v
    if f == f_i and i < N then
        ...// scale core hosting w to f_{i+1}
    end if
end procedure
```

- $N <= n$. This is called N-frequency tempo control.

Outline

Implementation

Worker-Core Mapping

- Relationship between workers (threads) and hosting CPU cores should be known.

- Relationship between workers (threads) and hosting CPU cores should be known.
- Two scheduling strategies are used.
  - Static : each worker is pre-assigned to CPU core.
  - Dynamic : each worker thread may migrate from one core to another during program execution.

- Relationship between workers (threads) and hosting CPU cores should be known.
- Two scheduling strategies are used.
  - Static : each worker is pre-assigned to CPU core.
  - Dynamic : each worker thread may migrate from one core to another during program execution.
- Only requirement of dynamic scheduling is that the worker **must** stay on its host core.

Outline

- HERMES does not adjust CPU frequencies when a worker becomes idle (when pop and steal both fail).

Implementation

Tempo Setting of Idle Workers/Core

- `HERMES` does not adjust CPU frequencies when a worker becomes idle (when pop and steal both fail).
- `YIELD`: The core is reallocated to another worker.

- `HERMES` does not adjust CPU frequencies when a worker becomes idle (when pop and steal both fail).
- `YIELD`: The core is reallocated to another worker.
- `YIELD` happens very rarely in practice.

Algorithm 2.1 Worker

```
w : WORKER
procedure SCHEDULE(w)
  loop
    t ← POP(w)
    if t==null then
      v = SELECT()
      t ← STEAL(v)
      if t==null then
        YIELD(w)
      else
        WORK(w, t)
      end if
    else
      WORK(w, t)
    end if
  end loop
end procedure
```

Algorithm 3.1 Worker

```
w : WORKER
procedure SCHEDULE(w)
  loop
    t ← POP(w)
    if t==null then
      w0 = w.next
      for w0 != null do
        UP(w0)
        w0 ← w0.next
      end for
      w.prev.next ← w.next
      w.next.prev ← w.prev
      w.next ← null
      w.prev ← null
      v = SELECT()
      t ← STEAL(v)
      if t==null then
        YIELD(w)
      else
        DOWN(w, v)
        if v.next != null then
          w.next ← v.next
          v.prev ← w.prev
        end if
        v.next ← w
        w.prev ← v
        WORK(w, t)
      end if
    else
      WORK(w, t)
    end if
  end loop
end procedure
```

Implementation

API

Algorithm 2.3 Pop

```
w : WORKER
procedure POP(w)
  w.T– –
  if w.H > w.T then
    w.T++
    LOCK(w)
    w.T– –
    if w.H > w.T then
      w.T++
      UNLOCK(w)
      return null
    end if
  end if
  UNLOCK(w)
  return w.DQ[w.T]
end procedure
```

Algorithm 3.4 Pop

```
w : WORKER
procedure POP(w)
  w.T– –
  if w.H > w.T then
    w.T++
    LOCK(w)
    w.T– –
    if w.H > w.T then
      w.T++
      UNLOCK(w)
      return null
    end if
  end if
  UNLOCK(w)
  if w.T - w.H < w.thld[w.S] then
    if w.S > 0 then
      if w.prev ! = null then
        w.S– –
        DOWN(w)
      end if
    end if
  end if
  return w.DQ[w.T]
end procedure
```

Implementation

API

Algorithm 2.4 Steal

```
v : WORKER // victim
procedure STEAL(v)
    LOCK(v)
    v.H++
    if v.H > v.T then
        v.H– –
        UNLOCK(v)
        return null
    end if
    UNLOCK(v)
    return v.DQ[v.H]
end procedure
```

Algorithm 3.5 Steal

```
v : WORKER // victim
procedure STEAL(v)
    LOCK(v)
    v.H++
    if v.H > v.T then
        v.H– –
        UNLOCK(v)
        return null
    end if
    UNLOCK(v)
    if v.T - v.H < v.thld[v.S] then
        if v.S > 0 then
            if v.prev ! = null then
                v.S ––
                DOWN(v)
            end if
        end if
    end if
    return v.DQ[v.H]
end procedure
```

Implementation

API

Algorithm 2.2 Push

```
w : WORKER
t : TASK
procedure PUSH(w,t)
    w.T++
    w.DQ[w.T] ← t
end procedure
```

Algorithm 3.3 Push

```
w : WORKER
K: number of thresholds
t: TASK
procedure PUSH(w, t)
    w.T++
    w.DQ[w.T] ← t
    if w.T - w.H > w.thld[w.S] then
        if w.S < K-1 then
            w.S++
            UP(w)
        end if
    end if
end procedure
```

Outline

1. DVFS switching cost

1. DVFS switching cost
2. Online profiling of workload threshold.

Implementation

Overheads

1. DVFS switching cost
2. Online profiling of workload threshold.
3. Affinity setting in dynamic scheduling.

Outline

Outline

- K-Nearest Neighbors (`KNN`) uses pattern recognition methods to classify ojects based on closest training examples in the feature space.
- Sparse Triangle Intersection (`Ray`) calculates for each ray the first triangle it intersects given a set of triangles contained in 3D bounding box and set of rays to penetrate.
- Integer Sort (`Sort`) is a parallel implementation of Radix Sort.
- Comparison Sort (`Compare`) is similar to `Sort` but uses sample sort.
- Convex Hull (`Hull`) is a computational geometry benchmark.

- K-Nearest Neighbors (`KNN`) uses pattern recognition methods to classify ojects based on closest training examples in the feature space.
- Sparse Triangle Intersection (`Ray`) calculates for each ray the first triangle it intersects given a set of triangles contained in 3D bounding box and set of rays to penetrate.
- Integer Sort (`Sort`) is a parallel implementation of Radix Sort.
- Comparison Sort (`Compare`) is similar to `Sort` but uses sample sort.
- Convex Hull (`Hull`) is a computational geometry benchmark.
- We use `KNN` and `Ray` for our experiments.

Outline

- Conducted experiments on two different systems to test effectiveness of the approach.
  - System A: 2 x 16-core AMD Opteron 6378 processors (Piledriver microarchitecture) with 64GB DDR3 1600 memory
  - System B: 8-core AMD FX-8150 processor (Bulldozer microarchitecture) with 16GB DDR3 1600 memory
- Both systems run Debian 3.2.46-1 x86-64 Linux (kernel 3.2.0-4-amd64) and support 5 different CPU frequencies.

Experimental Methodology

System Specs

- Conducted experiments on two different systems to test effectiveness of the approach.
  - System A: 2 x 16-core AMD Opteron 6378 processors (Piledriver microarchitecture) with 64GB DDR3 1600 memory
  - System B: 8-core AMD FX-8150 processor (Bulldozer microarchitecture) with 16GB DDR3 1600 memory
- Both systems run Debian 3.2.46-1 x86-64 Linux (kernel 3.2.0-4-amd64) and support 5 different CPU frequencies.
- Piledriver / Bulldozer were chosen as they supported *multiple clock domains*.

Outline

Experimental Methodology

Energy Consumption

- Measured through current meters over power supply lines to the CPU module.
- Data converted to NI DAQ and collected by NI LabVIEW SignalExpress with 100 samples per second.
- Supply voltage stable at 12V, so energy consumption computed as sum of current samples multiplied by 12×0.01

Outline

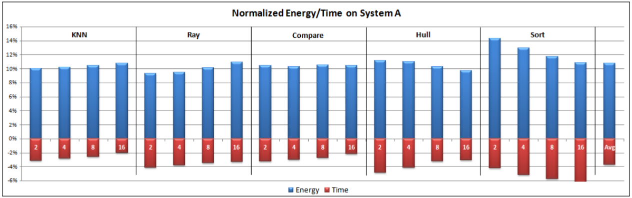
Outline

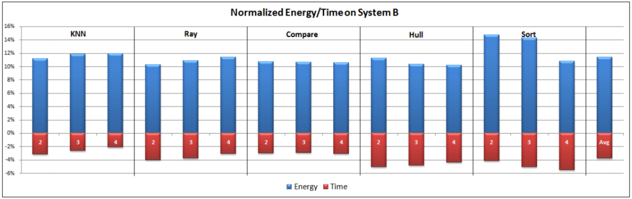
- On system A, experiments were conducted using 2, 4, 8, 16 workers.
- On system B, experiments were conducted using 2, 3, 4 workers.

- On system A, experiments were conducted using 2, 4, 8, 16 workers.
- On system B, experiments were conducted using 2, 3, 4 workers.
- In both systems, `HERMES` averages 11-12% energy savings over 3-4% performance loss.

Results

Overall Results





- Energy Delay Product(EDP): Product of energy consumption and execution time.

- Energy Delay Product(EDP): Product of energy consumption and execution time.
- EDP is often used as an indicator for demonstrating the energy/performance tradeoff.

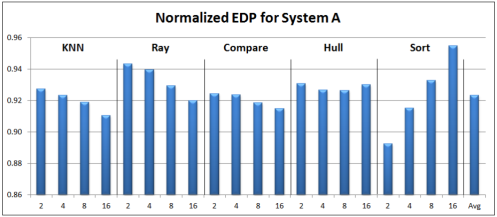
Results

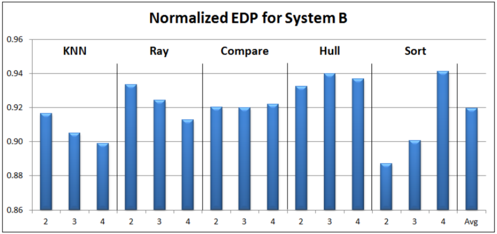
Overall Results

- Energy Delay Product(EDP): Product of energy consumption and execution time.
- EDP is often used as an indicator for demonstrating the energy/performance tradeoff.
- In both System A and System B, the average normalized EDP is about 0.92.

Results

Overall Results





Outline

- We also run benchmarks with only one of the two tempo-control strategies enabled.

Results

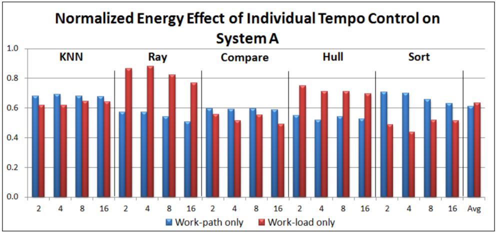
Relative Effectiveness

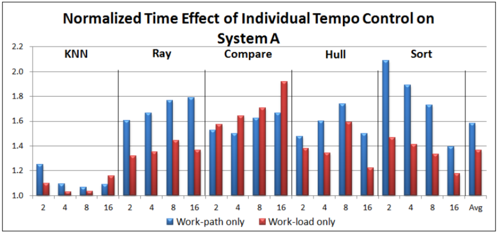
- We also run benchmarks with only one of the two tempo-control strategies enabled.
- Data are normalized w.r.t to the unified HERMES algorithm.

- We also run benchmarks with only one of the two tempo-control strategies enabled.
- Data are normalized w.r.t to the unified `HERMES` algorithm.
- The following set of figures show the complementary nature of the two strategies.

Results

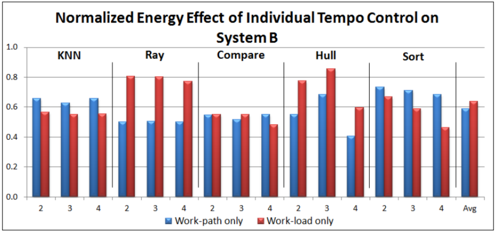
Relative Effectiveness

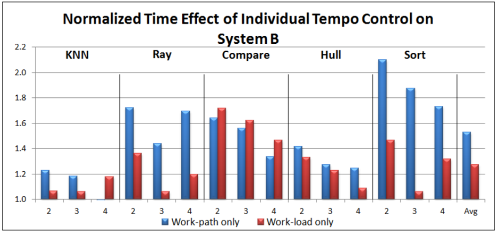




Results

Relative Effectiveness





Outline

- We experimentally evaluate the effects of different frequency mapping strategies.

- We experimentally evaluate the effects of different frequency mapping strategies.
- We only consider 2-frequency tempo control.
  - Fastest tempo is mapped to the first frequency.
  - All other tempos are mapped to the second frequency.

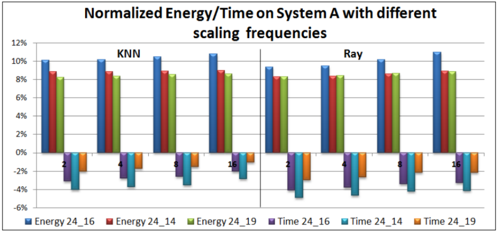
Results

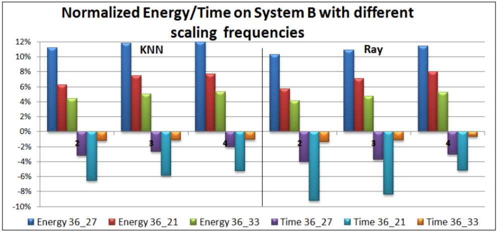
Effect of Frequency Selection

- We experimentally evaluate the effects of different frequency mapping strategies.
- We only consider 2-frequency tempo control.
  - Fastest tempo is mapped to the first frequency.
  - All other tempos are mapped to the second frequency.
- We fix the frequency for the fast tempo 2.4Ghz for System A and 3.6GHz for System B.
  - Experiment with different settings for the slow tempo.

Results

Effect of Frequency Selection





- Two scenarios:
  1. Higher frequency for slow tempo.
  2. Lower frequency for slow tempo.

Results

Effect of Frequency Selection

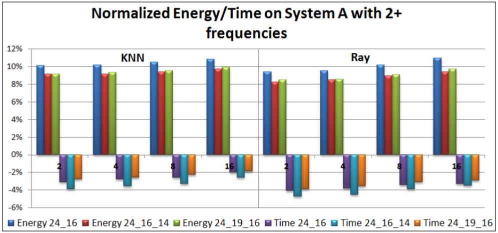
- Two scenarios:
  1. Higher frequency for slow tempo.
  2. Lower frequency for slow tempo.
- Optimal combination:
  - Frequency of slow tempo should be 60% of that of fast tempo.

Outline

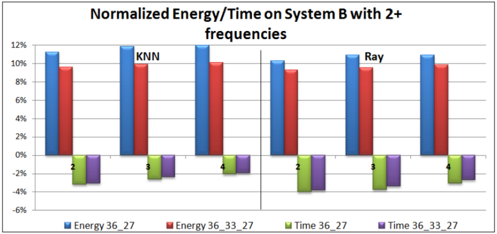
Results

N-Frequency Tempo Control

- We study how the number of frequencies impact the results.



Normalized Energy/Time on System A with 2+ frequencies
12%
10%
8%
6%
4%
2%
0%
-2%
-4%
-6%
KNN
Ray
2
4
8
16
Energy 24_16
Energy 24_16_14
Energy 24_19_16
Time 24_16
Time 24_16_14
Time 24_19_16



Normalized Energy/Time on System B with 2+ frequencies
12%
10%
8%
6%
4%
2%
0%
-2%
-4%
-6%
KNN
Ray
2
3
4
Energy 36_27
Energy 36_33_27
Time 36_27
Time 36_33_27

Results

N-Frequency Tempo Control

- A 3-frequency tempo control can sometimes incur less loss on performance.

- A 3-frequency tempo control can sometimes incur less loss on performance.
- But the 2-frequency tempo control has a slight edge on energy savings.

Results

N-Frequency Tempo Control

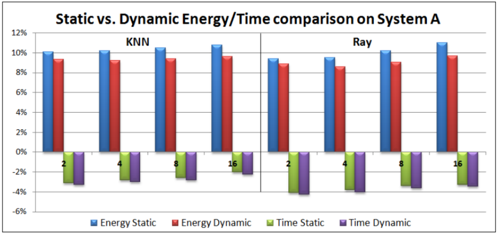
- A 3-frequency tempo control can sometimes incur less loss on performance.
- But the 2-frequency tempo control has a slight edge on energy savings.
- Maybe due to lesser overhead on DVFS.

Outline

- We study the effectiveness of HERMES under static scheduling and dynamic scheduling.

Results

N-Frequency Tempo Control

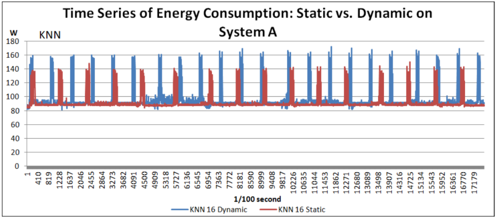


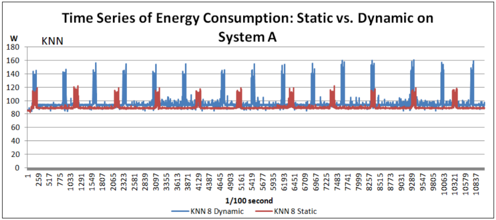
- Following figures show a more detailed analysis.
- Time series plot of power consumption.

- Following figures show a more detailed analysis.
- Time series plot of power consumption.
- The shape of the time series are dependent on the type benchmarks and their settings (such as number of workers).

Results

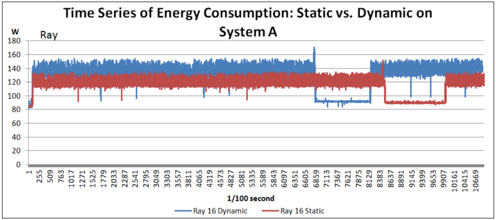
N-Frequency Tempo Control

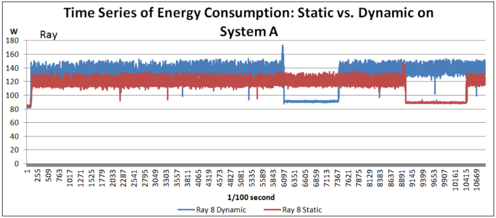




Results

N-Frequency Tempo Control





Static vs Dynamic Scheduling

- For each benchmark with the same number of workers, the executions of static scheduling vs. dynamic scheduling display similar patterns.

- For each benchmark with the same number of workers, the executions of static scheduling vs. dynamic scheduling display similar patterns.
- Dynamic scheduling incurs a slightly higher level of energy consumption.

Results

Static vs Dynamic Scheduling

- For each benchmark with the same number of workers, the executions of static scheduling vs. dynamic scheduling display similar patterns.
- Dynamic scheduling incurs a slightly higher level of energy consumption.
- Maybe due to the overhead of setting/re-setting the affinity of workers.

Outline

- Finally, we investigate the impact of naively applying frequency scaling.

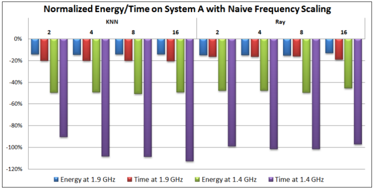
Results

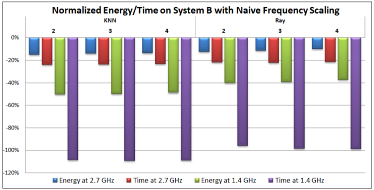
Naive Frequency Scaling

- Finally, we investigate the impact of naively applying frequency scaling.
- CPU frequencies are naively scaled down at a fixed frequency throughout the program execution.

Results

Naive Frequency Scaling





Outline

Conclusion

- Proposed HERMES – a novel and practical solution for improving energy efficiency of work-stealing applications.

Conclusion

- Proposed HERMES – a novel and practical solution for improving energy efficiency of work-stealing applications.
- Solution addresses the problem through judicious tempo control over workers, guided by a unified algorithm.

Conclusion

- Proposed `HERMES` – a novel and practical solution for improving energy efficiency of work-stealing applications.
- Solution addresses the problem through judicious tempo control over workers, guided by a unified algorithm.
- Solution is scalable as it only makes minor changes to existing work-stealing runtimes.

Conclusion

- Proposed HERMES – a novel and practical solution for improving energy efficiency of work-stealing applications.
- Solution addresses the problem through judicious tempo control over workers, guided by a unified algorithm.
- Solution is scalable as it only makes minor changes to existing work-stealing runtimes.
- HERMES is a language-level solution without the need of characterizing program executions on a per-application basis.

Outline

COTTON Integration

Work-First vs Help-First

- The COTTON runtime is help-first work-stealing runtime.
  - `HERMES` follows the work-first principle.

- The COTTON runtime is help-first work-stealing runtime.
  - `HERMES` follows the work-first principle.
- The sequence of tasks pushed in the deque remains the same.

- The COTTON runtime is help-first work-stealing runtime.
    - HERMES follows the work-first principle.
- The sequence of tasks pushed in the deque remains the same.
- No scenario yet identified where HERMES may show rogue behaviour in a help-first environment.

COTTON Integration

New Worker structure

- Member variables
  - Deque pointer
  - Linked list to maintain victim-thief and immediacy relationships
  - Frequency thresholds
- Member functions
  - `Up`(worker)
  - `Down`(worker)
  - `Down`(worker, victim)

COTTON Integration

Modified methods

- Worker routine
- Deque Push
- Deque Pop
- Deque Steal
- Deque Push

COTTON Integration

Modified methods

- Worker routine
- Deque Push
- Deque Pop
- Deque Steal
- Deque Push
- The corresponding modifications have been covered in the Implementation section.