Energy-Efficient Work-Stealing Language Runtimes

Haris Ribic and Yu David Liu
SUNY Binghamton
Binghamton NY 13902, USA
{hribic1,davidL}@binghamton.edu

Abstract

Work stealing is a promising approach to constructing multi-threaded program runtimes of parallel programming languages. This paper presents HERMES, an energy-efficient work-stealing language runtime. The key insight is that threads in a work-stealing environment – thieves and victims – have varying impacts on the overall program running time, and a coordination of their execution "tempo" can lead to energy efficiency with minimal performance loss. The centerpiece of HERMES is two complementary algorithms to coordinate thread tempo: the *workpath-sensitive* algorithm determines tempo for each thread based on thief-victim relationships on the execution path, whereas the *workload-sensitive* algorithm selects appropriate tempo based on the size of work-stealing deques. We construct HERMES on top of Intel Cilk Plus's runtime, and implement tempo adjustment through standard Dynamic Voltage and Frequency Scaling (DVFS). Benchmarks running on HERMES demonstrate an average of 11-12% energy savings with an average of 3-4% performance loss through meter-based measurements over commercial CPUs.

1. Introduction

Work stealing is a thread management strategy effective for maintaining multi-threaded language runtimes, with a specific target at parallel architectures and with a primary goal of load balancing. In the multi-core era, work stealing received considerable interest in language runtime design. With its root in Cilk [6, 16], work stealing is widely available in industry-strength C/C++/C#-based language frameworks such as Intel TBB [20], Intel Cilk Plus [21], and Microsoft .NET framework [25]. The core idea of work stealing has also made its way into mainstream languages such as Java [24], X10 [10, 23, 30], Haskell [28], and Scala [32]. There is an active interest in research improving its performance-critical properties, such as adaptiveness [2, 18], scalability [13], and fairness [14].

In comparison, energy efficiency in work-stealing systems has received little attention. At a time where power-hungry data centers and cloud computing servers are the norm of computing infrastructure, energy efficiency is a first-class design goal with direct consequences on operational cost, reliability,usability, maintainability, and environmental impact. The lack of energy-efficient solutions for work stealing system is particularly unfortunate, because the platforms which work stealing is most promising to make impact on – systems with a large number of parallel units – happen to be high on power consumption and require more sophisticated techniques to achieve energy efficiency [8, 12, 17, 22, 27, 33, 38].

HERMES is a first step toward energy efficiency for work stealing runtimes. Program execution under HERMES is *tempo-enabled*[1]: different threads may execute at different speeds (*tempo*), achieved by adjusting the frequencies of host CPU cores through standard DVFS. The effect of DVFS on energy management is widely known. The real challenge lies upon how to balance the trade-off between energy and performance, as lower frequencies may also slow down program execution. The primary design goal of HERMES is to apply the characteristics inherent and unique in the work stealing runtime to make judicious DVFS decisions, maximizing energy savings while minimizing performance loss. Specifically, HERMES is endowed with two algorithms:

- **workpath-sensitive tempo control**: thread tempo is set based on control flow, with threads tackling "immediate work" [7] executing at a faster tempo. As it turns out, this design corresponds to a key design principle in work stealing algorithm: the work-first principle.
- **workload-sensitive tempo control**: thread tempo is set based on the number of work items a thread needs to tackle – indicated by the size of the deque in work stealing runtimes – and threads with a longer deque execute at a faster tempo.

HERMES unifies the two tempo control strategies in one. Our experiments show the two strategies are highly *complementary*. For instance, on a 32-core machine, each strategy can contribute to 6% and 7% energy savings respectively, whereas the unified algorithm can yield 11% energy savings. In the same setting, each strategy incurs 6% and 5%

[1] The term is inspired by music composition, where each movement of a musical piece is often marked with a different tempo – *e.g. allegro* ("fast") and *lento* ("slow") – to indicate the speed of execution.

performance loss respectively, whereas the unified algorithm incurs 3% loss.

This paper makes the following contributions:

1. The first framework that we know of addressing energy efficiency of work stealing systems. The framework achieves energy efficiency through thread tempo control.
2. Two novel, complementary tempo control strategies: one workpath-sensitive and one workload-sensitive.
3. A prototyped implementation and experimental evaluation demonstrating an average of 11-12% energy savings with 3-4% performance loss over work stealing benchmarks. The results are stable throughout comprehensive design space exploration.

2. Background: Work Stealing

Work stealing was originally developed in Cilk [6, 16], a C-like language designed for parallel programming. The main appeal of work stealing is that it offers a synergic solution spanning the compute stack, bridging the gap between abstraction layers such as architectures, operating systems, compilers and program runtimes, and programming models.

From one perspective, work stealing is a load balancing scheduler for multi-threaded programs over parallel architectures. The program runtime consists of multiple threads called *workers*, each executing on a host CPU core (or hardware parallel unit in general). Each worker maintains a queue-like data structure – called a *deque* – each item of which is a *task* to be processed by the worker. When a worker completes the processing of a task, it picks up one more from the deque and continues the execution for that item. When the deque is empty (we say the worker or its host core is *idle*), the worker *steals* a task from the deque of another worker. In this case we call the stealing worker a *thief* whereas the worker whose item was stolen a *victim*. The selection of victims follows the principles observed by load balancing and may vary in different implementations of work stealing.

What sets work stealing apart from standard load balancing techniques is how the runtime structure described above corresponds to program structures and compilation units. First, each task on the deque turns out to be a block of executable code – or more strictly, a program counter pointing to the executable code – demarcated by the programmer and optimized by the compiler. In that sense, to have a worker “pick up a task” is indeed to have the worker continue its execution over the executable code embodied in the task. To describe the process in more detail, let us use the following Cilk example:

```
 L1 cilk int f()
 L2 {   int n1 = spawn f1();
 L3       ...  // other statements
 L4 }
 L5 cilk int f1() {
 L6      int n2 = spawn f2();
 L7      ... // other statements
 L8 }
 L9 cilk int f2() {
L10      ... // other statements
L11 }
```

Logically, each spawn can be viewed as a thread creation. On the implementation level however, a work stealing runtime adopts Lazy Task Creation [31], where for each spawn, the executing worker simply puts a task onto its own deque, either later to be picked up by itself or stolen by some other worker. This strategy aligns thread management with the underlying parallel architecture: a program that invokes f above 20 times but runs on a dual-core CPU can operate only with 2 threads (workers) instead of 40.

Work-First Principle The non-trivial question here is what the item placed on the deque should embody. For instance, when L2 is executed, one tempting design would be to consider f1 as the task placed on the deque. The Cilk-like work stealing algorithm takes the opposite approach: it places the *continuation* of the current spawn statement onto the deque. In the example, it is the program counter pointing to L3. The current worker continues to invoke f1 as if spawn were elided.

This design reflects a fundamental principle well articulated in Cilk: the *work-first principle*. The principle concerns the relationship between the parallel execution of a program and its corresponding serial execution. (A logically equivalent view for the latter would be to have the parallel program execute on a single-core machine.) Let us revisit the example above. If it is executed on a single-core machine, f1 is the “immediate” task when L2 is reached, and hence carries more urgency. For that reason, f1 should be immediately executed by the current worker, whereas the continuation is not as urgent and is hence placed on the deque.

Work-first principle plays a pivotal role in the design of work stealing systems. In Cilk, it further leads to a compilation strategy known as fast/slow clones, and distinct solutions for locking [16].

Deque Management One natural consequence of placing continuations onto the deque is that the order of tasks on the deque reflects the immediacy of processing these items as defined by the work-first principle: the earlier the item is placed, the less immediate it is. For example, if the control flow of a worker reaches L10, two tasks are placed on the deque, the program counter for L3 (when the spawn in L2 is executed) and the program counter for L7 (when the spawn in L6 is executed). In a serial execution, L3 will only be encountered after L7.

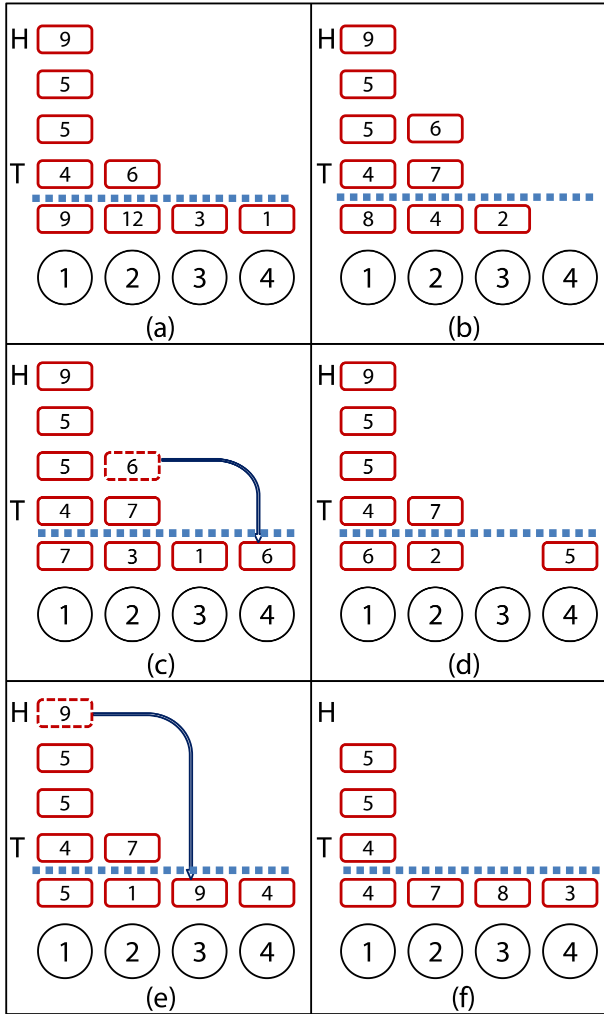

Figure 1: Work Stealing: An Illustration

With this observation, deque is designed as a data structure that can be manipulated on both ends. Let us call the *head* of the deque as the earliest item placed on the deque by the worker, whereas the *tail* of the deque as the latest. When a worker becomes idle, it always retrieves from the tail of its own deque, *i.e.* the most immediate task. On the other hand, when a thief attempts to steal from a worker, it always retrieves from the head of that worker's deque, *i.e.* the least immediate task. From now on, we call the worker placing a task to its own deque a *push*, while removing a task from its own deque a *pop*. We continue to use term *steal* to refer to a worker removing a task from another worker's deque.

Figure 1 illustrates the time sequence of a typical program execution on a 4-core CPU (numbered as 1-4 at the bottom of each sub-figure). For each pair of adjacent figures, the elapsed time is one time unit. The rectangle below the dotted line is the currently executed task, and the rectangles above form the deque for the worker on that core, with the top rectangle representing the "head" task (`H`) and the bottom representing the "tail" task (`T`). The number inside the rectangle represents the number of time units needed to complete that task if the task were to run serially. In the first elapsed time unit – from Figure 1(a) to Figure 1(b) – core 2 `spawn`'s another task with 2 time units. Its continuation,

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
```

Structures

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

structure TASK
  ... // program counter, etc
end structure
```

Other Definitions

```
procedure WORK(w, t)
// worker w runs task t
procedure SELECT()
// select and return a victim
procedure YIELD(w)
// yield worker w
procedure LOCK(w)
procedure UNLOCK(w)
// lock/unlock w
```

Algorithm 2.2 Push

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

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 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
```

Figure 2: Work Stealing Algorithm

with 12-1-4 = 7 time units left, is pushed onto the tail of its deque. In the same elapsed time, core 4 completes its executing task. Since its deque is empty, core 4 steals from the head of the deque of core 2, as shown in Figure 1(c). Another stealing happens in Figure 1(e), after core 3 becomes idle in Figure 1(d). In Figure 1(f), core 2 completes its current task, but since its deque is not empty, it pops a task from the tail of its deque.

Work Stealing Scheduler Fig. 2 provides a simplified specification of the classic work stealing algorithm. The state of each worker thread is maintained by data structure WORKER, which consists of a deque `DQ` and two indices for its head (`H`) and (`T`) respectively. As shown in Algorithm 2.1, a worker either attempts to POP a task from its deque – or if it is not available – SELECT a victim and STEAL a task from it. Once a task is obtained, the worker WORK's on it, during which (we elide the WORK specification here) may further spawn new tasks and PUSH them to its deque. If no task is available either through POP or STEAL, the worker thread YIELD's its host core. We leave out the definition of SELECT: a typical implementation (such as in Cilk) is a randomized algorithm.

The definitions of PUSH, POP, and STEAL are predictable, with PUSH incrementing the tail index, POP decrementing the tail index, and STEAL incrementing the head index. One invariant the scheduler maintains is the head index is less than or equal to the tail index. When head index and tail index are equal, there is a possibility a thief and a victim attempt to work on the same task. To resolve potential contention, LOCK and UNLOCK are introduced. The locking strategy adopted by most work stealing runtimes are reminiscent of optimistic locking. This somewhat stylistic protocol is known as THE [16], orthogonal to the rest of the paper.

3. Energy Efficient Work Stealing

In this section, we describe how HERMES improves energy efficiency of work stealing runtimes. The overall technique of HERMES is DVFS-guided tempo control: different workers can execute at different speeds – workers tackling more urgent tasks run at the faster tempos to retain high performance, whereas others run at the slower tempos to save energy. The main challenge in this design is to determine the appropriate tempo for each worker thread, and the timing for tempo adjustment. To achieve this goal, we developed two novel algorithms.

3.1 Workpath-Sensitive Tempo Control

Our *workpath-sensitive* tempo control strategy is fundamentally aligned with the work-first principle of classic work stealing algorithms: tasks encountered earlier – if the program were to be executed serially – carry more immediacy and will be executed at the faster tempos. Recall that in work stealing systems, the order of tasks on the deque reflects the immediacy, with the head being the least immediate. Further, recall that a thief always steals from the head of a victim's deque. Hence, every worker executing a stolen task carries less immediacy than its victim worker.

The workpath-sensitve tempo control strategy says that the victim worker takes precedence over the thief worker in a thief-victim relationship, or in other words, the thief worker should be executing at a slower tempo than the victim worker. It is important to realize that the thief-victim relationship between workers has a dynamic lifespan: it is formed at steal time, and terminates when either the thief or the victim completes its current set of tasks and becomes idle again.

Specifically, workpath-sensitive tempo control entails two important design ideas:

- **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 the thief should be raised. In this case, the previous victim worker simply becomes an idle thread, and the immediacy should be "relayed" to the thief.

Intuitively, the design of **Immediacy Relay** can be analogously viewed as a relay race. When a worker finishes the tasks that carry immediacy, it needs to pass on the immediacy "baton" to the next worker.

Figure 3 demonstrates the key ideas of workpath sensitivity, with the 6 subfigures representing (not necessarily consecutive) "snapshots" of a program execution sequence. We use different gray-scales to represent different tempos. The darker the shade of the circle is, the slower tempo the hosted worker is executed at. Worker 1 starts in Figure 3(a) with a task of 100 time units. In Figure 3(b), a task with 94 time units is pushed to the deque and subsequently stolen by worker 2. According to **Thief Procrastination**, worker 2 executes at a tempo one level slower than worker 1. In Figure 3(c), worker 3 steals from worker 2 (*i.e.*, "a thief's thief") executing at a tempo further slower than worker 2. At Figure 3(d), worker 1 finishes all its tasks. According to **Immediacy Relay**, its thief (worker 2 in this case) needs to raise its tempo. Intuitively, what worker 2 currently works on is the "unfinished business" when the 100 time units started on worker 1. When worker 2 raises its tempo by one level, HERMES transitively raises the tempo of worker 2's thief. This is demonstrated in Figure 3(e). In Figure 3(f), worker 1 steals again, starting a new thief-victim relationship with worker 2, except that worker 1 this time is the thief.

3.2 Workload-Sensitive Tempo Control

HERMES is further equipped with a *workload-sensitive* strategy for tempo control. The intuition is simple: a worker needs to work faster when there are more tasks to handle.

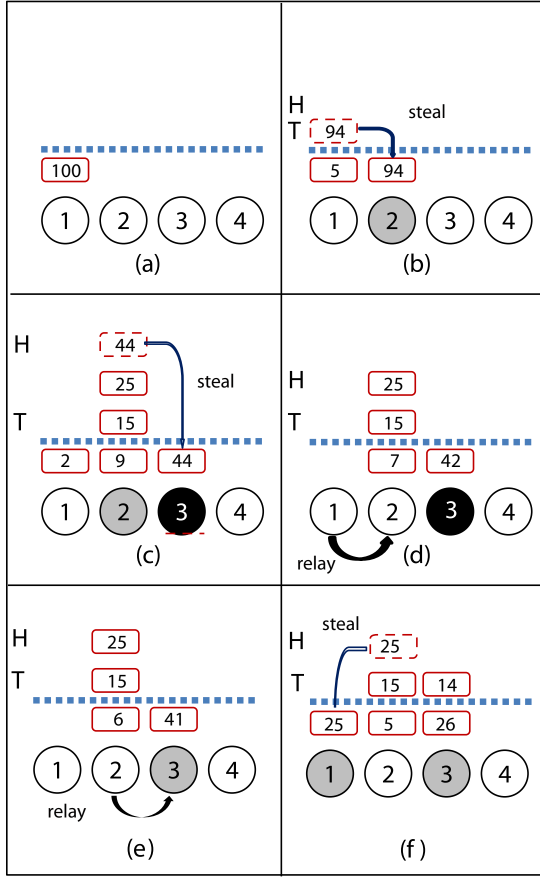

Figure 3: Example: Workpath-Sensitive Tempo Control

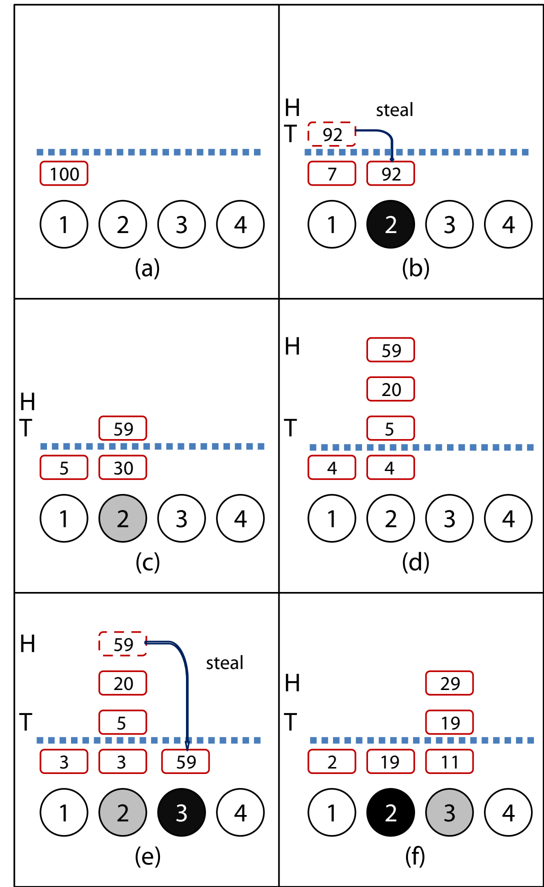

Figure 4: Example: Workload-Sensitive Tempo Control

In the case of work stealing, a natural indicator of the workload is the deque size: the number of tasks waiting to be processed by a worker.

We demonstrate the ideas of workload-sensitivity through Figure 4. Let us assume we have three tempo levels, set based on the deque size with two thresholds: 1 and 3. As a convention, HERMES always bootstraps the program execution with the fastest tempo, as in Figure 4(a). At snapshot Figure 4(b), core 2 steals one task. Since its deque is of size 0, lower than the first threshold, the tempo for worker 2 is set at the lowest one. As worker 2 progresses such as PUSH more tasks to its deque, its tempo rises to the medium level in Figure 4(c), and then fastest level in Figure 4(d). The tempo is slow downed again when worker 2 is stolen, dropping its deque size below the second threshold in Figure 4(e), and even slower when it pops more items from its own deque in Figure 4(f).

HERMES determines the thresholds through a lightweight form of online profiling. Our runtime periodically samples deque sizes, and computes the average of the last fixed number of samples. Let that average be L. In an execution with K thresholds, the thresholds for the next period are set at

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

where $1 \leq i \leq K$. For example, if the average deque size is 15 and there are 2 thresholds, we apply the fastest tempo if the deque size is no less than 20, the medium tempo for a deque size between 10 and 20, and the slowest tempo otherwise.

3.3 Unified Algorithm Specification

Figure 5 presents the pseudocode of the core HERMES algorithm. The modifications on top of the classic work stealing algorithm are highlighted, with two colors indicating workpath-sensitive and workload-sensitive support respectively.

The key data structure to support workpath sensitivity is a double-linked list across workers, connected by the `next` and `prev` pointers. The list maintains the order of immediacy: when worker `w1`'s `next` worker is `w2`, it means `w2` is processing a task immediately following the tasks processed

Algorithm 3.1 Worker

```
1:  w : WORKER
2:  procedure SCHEDULE(w)
3:    loop
4:      t ← POP(w)
5:      if t==null then
6:        w0 = w.next
7:        for w0 != null do
8:          UP(w0)
9:          w0 ← w0.next
10:       end for
11:       w.prev.next ← w.next
12:       w.next.prev ← w.prev
13:       w.next ← null
14:       w.prev ← null
15:       v = SELECT()
16:       t ← STEAL(v)
17:       if t==null then
18:         YIELD(w)
19:       else
20:         DOWN(w, v)
21:         if v.next != null then
22:           w.next ← v.next
23:           v.prev ← w.prev
24:         end if
25:         v.next ← w
26:         w.prev ← v
27:         WORK(w, t)
28:       end if
29:     else
30:       WORK(w, t)
31:     end if
32:   end loop
33: end procedure
```

Algorithm 3.2 Tempo Adjustment

```
procedure DOWN(w, v)
// set w to one tempo lower than v
procedure DOWN(w)
// set w to one tempo lower
procedure UP(w)
// set w to one tempo higher
```

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
```

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
```

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
```

Structures

```
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
```

Figure 5: Core HERMES Algorithm (X for Workpath Sensitivity and X for Workload Sensitivity)

by worker `w1`, where immediacy is defined according to the work-first principle.

When stealing succeeds (lines 20-27), the thief worker becomes the immediate `next` worker of the victim. The tempo of the thief is set to be one level slower than the victim (line 20), and the `prev` and `next` references are properly set (lines 25-26). We will detail the implementation of tempo adjustment in the next subsection. One issue to address is

that the victim might already be stolen by another thief before. In that case, the algorithm inserts the current thief ahead of the previous thief on the linked list (lines 21-24). In other words, the current thief is more immediate than the previous thief. This corresponds to how the order of tasks on the victim's deque reflects immediacy: the tasks stolen earlier are not as immediate as the tasks stolen later (recall Sec. 2).

When a worker becomes idle again and out of work (line 6), it effectively terminates the thief-victim relationship previously developed since it became out of work last time. If the current worker is a victim, then according to the design of **Immediacy Relay**, the tempos of its thieves are raised, passing on the immediacy (lines 7-10). Note that UP(`w`) is defined to raise the tempo of `w` one level up from its current level, so in a scenario where a thief worker `w1` is further stolen by another thief `w2`, both workers will have their tempo raised by one level, and `w2` can still maintain a slower tempo than `w1`. Finally, the current worker is removed from the linked list (lines 11-14).

Workload sensitivity is relatively simple to support. Each worker maintains an array `thld` to record its thresholds, with the number of thresholds defined by constant K. The computation of the `thld` was described in Sec. 3.2. The algorithm increases the tempo when PUSH makes the deque size reach the next threshold up, or decreases the tempo when either POP or STEAL reduces the deque size to the next threshold down.

One interesting aspect of our algorithm is that workpath sensitivity and workload sensitivity work largely independently – workpath-sensitive tempo control is applied when the deque is empty whereas workload-sensitive tempo control is appplied when the deque is not – so the unification of the two is a simple matter. The only intersction of the two lies in one fact: when a worker is at the beginning of the immediacy list, we choose not to reduce its tempo even if workload sensivity advises so. This can be seen in the `w.prev!= null` condition in POP and the similar `v.prev!= null` condition in STEAL. In other words, if the task a worker processes is immediate, we still execute it with a fast tempo regardless of deque size.

3.4 Lower-Level Design Considerations

Tempo-Frequency Mapping HERMES achieves tempo adjustment through DVFS, and modern CPUs usually support a limited, discrete number of frequencies. We now define tempo adjustment in the presence of a fixed number of frequencies. Let $\{f_1, f_2, \ldots, f_n\}$ be frequencies supported by a CPU core, where $f_i > f_{i+1}$ for any $1 <= i <= n-1$. For simplicity, let us assume all cores of a CPU support the same frequencies. The algorithm in the previous section stays the same, except UP and DOWN procedures should be refined. For instance,

procedure DOWN(w, v)
  $f \leftarrow$ 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

where $N <= n$ is a constant to further restrict the range of frequencies used for the runtime. In other words, a CPU may support n frequencies, but a runtime may only choose to use the highest N-number. In practice, a subset of frequencies often strikes a better trade-off between energy and performance: they are sufficient to yield energy savings, yet without incurring significant performance penalties due to low operating CPU frequencies. We call this design N-frequency tempo control.

Worker-Core Mapping HERMES relies on the knowledge of the relationship between workers (threads) and their hosting CPU cores, information readily available in work stealing runtimes. For maintaining this mapping, we allow for two scheduling strategies in our experiments: (a) static scheduling: each worker thread is pre-assigned to a CPU core; (b) dynamic scheduling: each worker thread may migrate from one core to another during program execution. The only requirement for dynamic scheduling is that during the processing of a task (*i.e.* an invocation of the WORK procedure), a worker stays on its host core. With this, OS cannot re-assign a worker to a different core if preempted, invalidating frequency settings at context switch time. We think this is a reasonable design because (1) work stealing by design is a load balancing strategy, overlapping with the goal of OS-level load balancing; (2) work stealing tasks usually take a short amount of time to complete.

We achieve the goal of binding workers to cores through affinity setting. For dynamic scheduling, affinity is set right before each WORK invocation (line 27 and line 30) and reset at the completion of each invocation.

One scenario common in standard multi-threaded program runtimes is the support of multiple threads executing concurrently on the same core. This is a non-issue for work stealing runtimes. Lazy task creation fundamental in work stealing entails that the number of workers can be statically bound by CPU resources, not program logic.

Tempo Setting of Idle Workers/Cores HERMES does not adjust CPU frequencies when a worker becomes idle but fails to steal. This corresponds to lines 17-18 in the algorithm where YIELD happens. In work stealing systems, there are usually more tasks to keep all workers busy, either through POP or STEAL, with YIELD relatively uncommon. When a YIELD does happen, the core is often reallocated to another worker, which sets its CPU frequency based on its own workpath-sensitive and workload-sensitive rules.

Overhead The overhead of our approach comes in 3 aspects: (1) DVFS switching cost. DVFS switching time is usually in the tens of microseconds, magnitudes smaller than

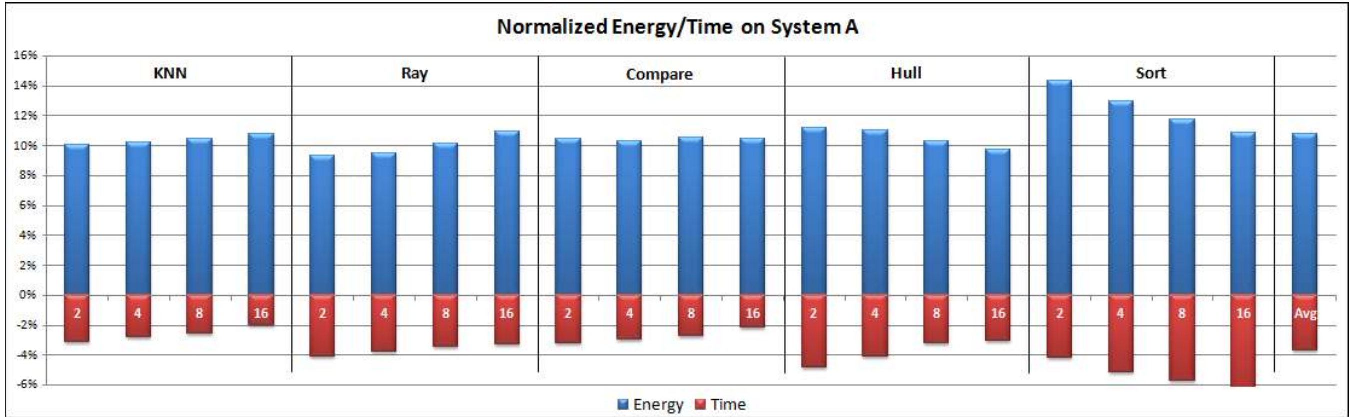

Figure 6: Normalized Energy Savings (Blue) and Time Loss (Red) of HERMES w.r.t. Intel Cilk Plus on System A

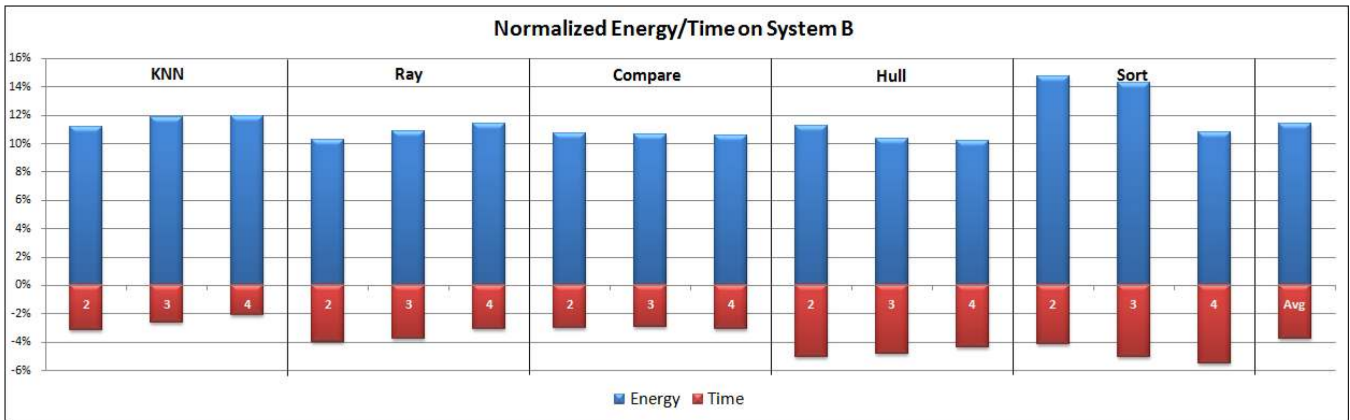

Figure 7: Normalized Energy Savings (Blue) and Time Loss (Red) of HERMES w.r.t. Intel Cilk Plus on System B

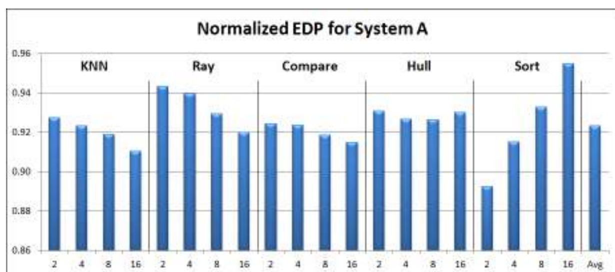

Figure 8: Normalized EDP for System A

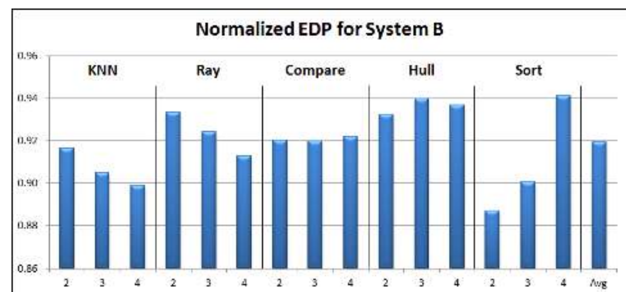

Figure 9: Normalized EDP for System B

the execution time of tasks. Our use of DVFS is relatively coarse-grained: tempo control is not applied during the execution of a task. (2) online profiling of workload threshold; (3) affinity setting in dynamic scheduling.

4. Implementation and Evaluation

HERMES is implemented on top of Intel Cilk Plus (build 2546). In this section, we present the experimental results.

4.1 Experiment Setup

We selected benchmarks from the Problem-Based Benchmark Suite (PBBS) [5]. The benchmarks support parallel programming, and our selection of the benchmarks support Cilk-like syntax such as `spawn`. K-Nearest Neighbors (`KNN`) uses pattern recognition methods to classify objects based on closest training examples in the feature space. Sparse-

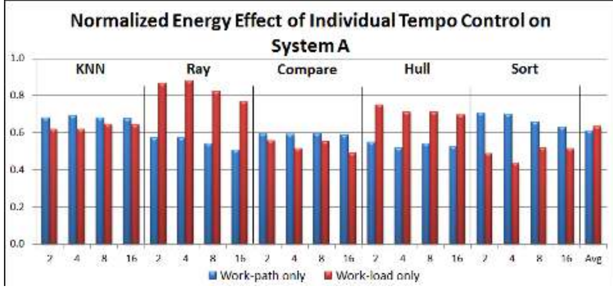

Figure 10: Energy: Workpath vs. Workload on System A

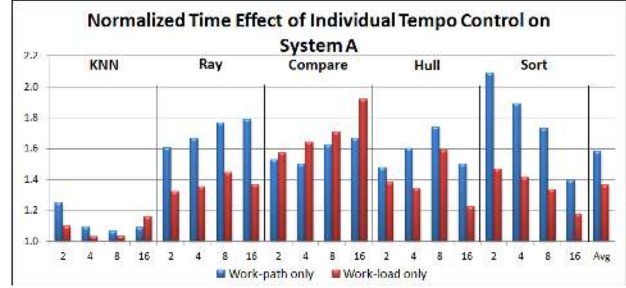

Figure 11: Time: Workpath vs. Workload on System A

Triangle Intersection (Ray) benchmark returns the first triangle each penetrating ray R intersects in a set of triangles T in a three-dimensional bounding box. Integer Sort (Sort) is an implementation of parallel radix sort. Comparison Sort (Compare) is similar to Sort but uses sample sort. Convex Hull (Hull) is a computational geometry benchmark.

To measure the effectiveness of our approach across platforms, we constructed our experiments on two systems:

- System A: a machine with 2×16-core AMD Opteron 6378 processors (Piledriver microarchitecture) running Debian 3.2.46-1 x86-64 Linux (kernel 3.2.0-4-amd64) and 64GB of DDR3 1600 memory. Each processor supports 5 frequencies: 1.4GHz, 1.6GHz, 1.9GHz, 2.2GHz and 2.4GHz.
- System B: a machine with an 8-core AMD FX-8150 processor (Bulldozer microarchitecture) running Debian 3.2.46-1 Linux (kernel 3.2.0-0.bpo.2-amd64) and 16GB of DDR3 1600 memory. The processor supports 5 frequencies: 1.4GHz, 2.1GHz, 2.7GHz, 3.3GHz and 3.6GHz.

Piledriver/Bulldozer microarchitectures are among the latest commercial CPUs that support *multiple clock-domains*, *i.e.* CPUs whose individual cores can have their frequencies adjusted independently. Specifically, in both architectures, every two cores share one clock domain. In other words, System A has 16 independent clock domains, whereas System B has 4. To avoid the undesirable DVFS interference, all our experiments are performed over cores with distinct clock domains. For example, our experiments on System A consider as many as 16 workers, and no two workers may share the same clock domain.

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

We executed each benchmark using both our HERMES scheduler as well as the unmodified Intel Cilk Plus scheduler as a control. For each benchmark, we run 20 trials and calculate the average of the trials, disregarding the first 2 trials.

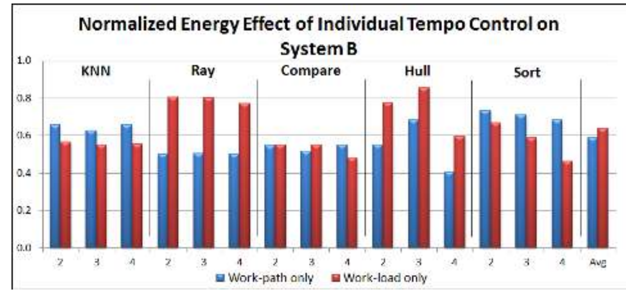

Figure 12: Energy: Workpath vs. Workload on System B

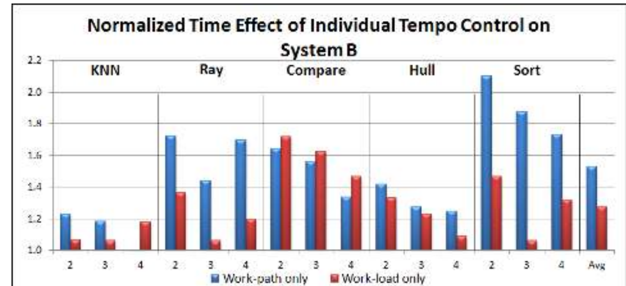

Figure 13: Time: Workpath vs. Workload on System B

4.2 Experimental Results

Overall Results Figure 6 and Figure 7 summarize the energy/performance results of HERMES on System A and System B respectively. All data are normalized against the baseline execution over unmodified Intel Cilk Plus. The blue columns are the percentage of energy savings of HERMES, while the red columns are the percentages of performance (time) loss. The results are grouped by benchmarks, and within each group, the columns show different numbers of workers. On System A, we conducted experiments using 2, 4, 8, and 16 workers (hence each group in Figure 6 has 4 columns, in that order). On System B, we conducted experiments using with 2, 3, and 4 workers (hence each group in

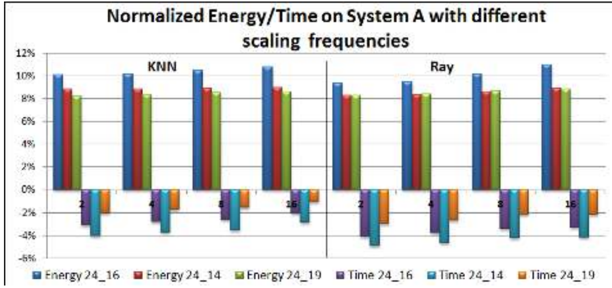

Figure 14: The Effect of Frequency Selections on System A (For each benchmark, the 4 groups are for 2, 4, 8, 16 workers respectively. Within each group, columns 1 and 4 are energy saving and time loss for frequency pair 2.4/1.6GHz; columns 2 and 5 are energy saving and time loss for frequency pair 2.4/1.4Ghz; columns 3 and 6 are energy saving and time loss for frequency pair 2.4/1.9Ghz)

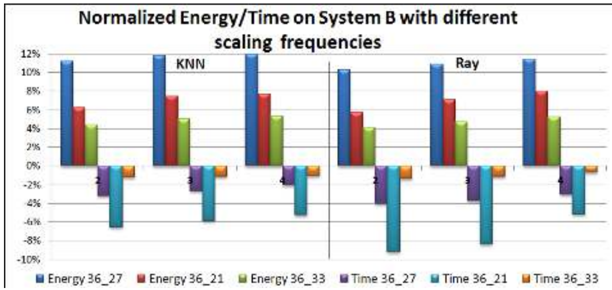

Figure 15: The Effect of Frequency Selection on System B (For each benchmark, the 3 groups are for 2, 3, 4 workers respectively. Within each group, columns 1 and 4 are energy saving and time loss for frequency pair 3.6/2.7GHz; columns 2 and 5 are energy saving and time loss for frequency pair 3.6/2.1Ghz; columns 3 and 6 are energy saving and time loss for frequency pair 3.6/3.3Ghz)

Figure 7 has 3 columns, in that order). The last columns in both Figures show the average.

In both systems, HERMES average 11-12% energy savings over 3-4% performance loss. We have further computed the Energy-Delay Product (EDP) of the benchmarking results, and the normalized results are shown in Figure 8 and Figure 9 respectively. Often used as an indicator for demonstrating the energy/performance trade-off, EDP is the product of energy consumption and execution time. A smaller value in EDP is aligned with our intuition of improved energy efficiency. In both System A and System B, the average normalized EDP is about 0.92.

HERMES shows remarkable stability across benchmarks, worker counts, and underlying systems. EDP is improved without exception. Throughout our experiments, stability is a recurring theme. This is an unexpected feature while experimenting in a highly dynamic setting.

Relative Effectiveness of Workpath vs. Workload Sensitivity To determine how much workpath sensitivity and workload sensitivity contribute to HERMES, we also run benchmarks with only one of the two strategies enabled. Figure 10 and Figure 11 shows the energy/time effects on System A, while Figure 12 and Figure 13 shows the energy/time effects on System B. To highlight the individual contributions of the two tempo control strategies to the unified HERMES algorithm, we normalize the percentage of savings/loss. For instance, if a tempo control strategy alone can lead to 6% energy savings whereas the HERMES algorithm (unified with both strategies) can lead to 12% energy savings, we record $6/12 = 0.5$ in Figure 10 and Figure 12. For another instance, if a tempo control strategy alone can lead to 6% performance loss whereas the HERMES algorithm (unified with both strategies) can lead to 3% performance loss, we record $6/3 = 2$ in Figure 11 and Figure 13. In all figures, the blue columns are the workpath-only results and the red columns are the workload-only results.

This set of figures show the complementary nature of workpath sensitivity and workload sensitivity. Take the 8-core execution of `Compare` on System A for example. In Figure 10, workpath sensitivity alone leads to around 60% energy savings relative to the unified HERMES algorithm, and workload sensitivity alone leads to around 55% energy savings relative to the unified HERMES algorithm. The overall energy saving is nearly the sum of saving from the two strategies alone. In Figure 11, again for the 8-core execution of `Compare`, the time loss of workpath-alone strategy is about 1.6 time of the time loss of the unified algorithm, and the time loss of workload-alone strategy is about 1.7 time of the time loss of the unified algorithm. In other words, the unified algorithm obtains the best of the two worlds: the unified strategy leads to more energy savings (almost the sum of the strategies alone), but incurs less performance loss (almost half of the strategies alone).

The Effect of Frequency Selection We conceptually explored the design space of tempo-frequency mapping in Section 3.4, and now experimentally evaluate the effects of different frequency mapping strategies. Figure 14 and Figure 15 are results for mapping tempos to different CPU frequencies. For simplicity, we only consider 2-frequency tempo control, where the fastest tempo is mapped to the first frequency, and all other tempos are mapped to the second frequency. In all experiments, we fix the frequency for the fast tempo – 2.4Ghz for System A and 3.6GHz for System B – and experiment with different settings for the slow tempo.

As predicted, selecting a higher frequency for the slow tempo is likely to yield less performance loss, but also fewer energy savings. This is demonstrated by columns 3 and 6 in each benchmark for both Figures, and the effect is particularly evident in System B. Selecting a very low frequency for the slow tempo (columns 2 and 5 in each benchmark for both Figures) will lead to significant performance loss. In fact,

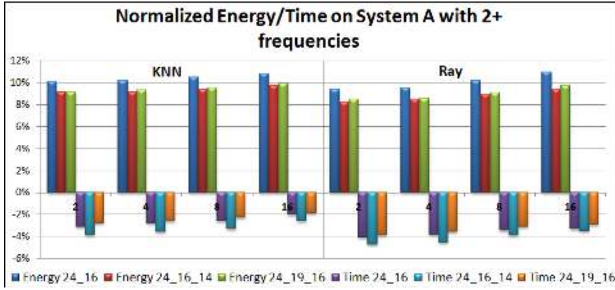

Figure 16: N-Frequency Tempo Control on System A (For each benchmark, the 4 groups are for 2, 4, 8, 16 workers respectively. Within each group, columns 1 and 4 are energy saving and time loss for 2-frequency combination 2.4/1.6GHz; columns 2 and 5 are energy saving and time loss for 3-frequency combination 2.4/1.6/1.4Ghz; columns 3 and 6 are energy saving and time loss for 3-frequency combination 2.4/1.9/1.6Ghz)

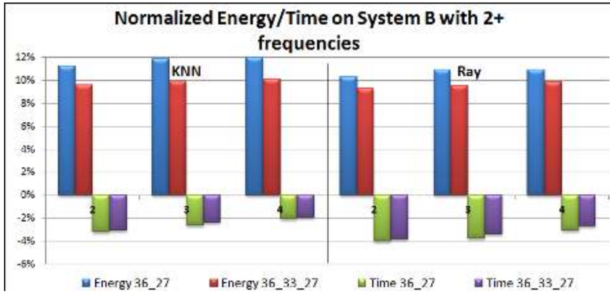

Figure 17: N-Frequency Tempo Control on System B (For each benchmark, the 3 groups are for 2, 3, 4 workers respectively. Within each group, columns 1 and 3 are energy saving and time loss for 2-frequency combination 3.6/2.7GHz; columns 2 and 4 are energy saving and time loss for 3-frequency combination 3.6/3.3/2.7Ghz)

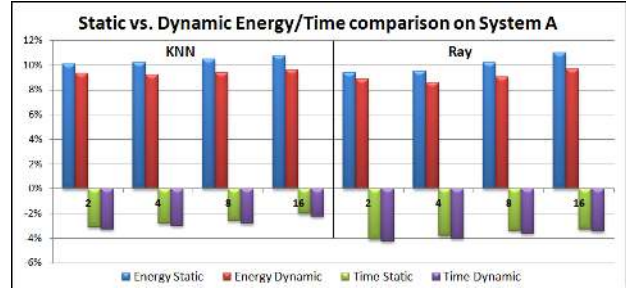

Figure 18: Static vs. Dynamic Scheduling

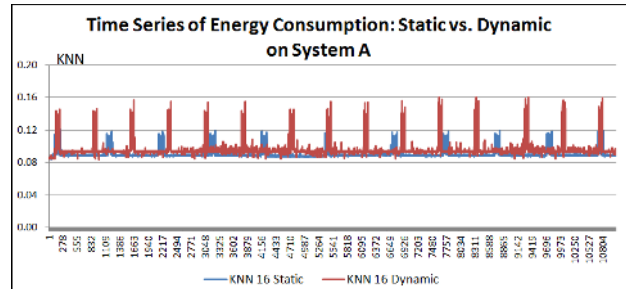

Figure 19: Static vs. Dynamic Scheduling (time series, KNN, 16 workers, System A)

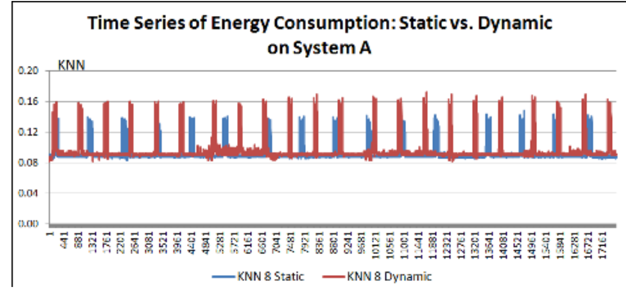

Figure 20: Static vs. Dynamic Scheduling (time series, KNN, 8 workers, System A)

such a selection is not wise for energy savings either: significant increase in the execution time may increase energy consumption, because the latter also holds a linear relationship with time. Heuristically, our experiments seem to suggest the optimal combination often comes with the golden ratio: the frequency for the slow tempo is about 60% percent of the one for the fast tempo.

N-Frequency Tempo Control In the next set of experiments, we study how the number of frequencies impact the results, demonstrated in Figure 16 and Figure 17. Overall, the results between 2-frequency tempo control and 3-frequency tempo control are similar. A 3-frequency tempo control can sometimes incur less loss on performance, as demonstrated by column 6 for each group in Figure 16 and column 4 for each group in Figure 17, but the 2-frequency tempo control has a slight edge on energy savings. We surmise the small advantage of 2-frequency tempo control on energy savings might be due to its lesser overhead on DVFS. In this context, tempo adjustment occurs less frequently.

Static Scheduling vs. Dynamic Scheduling In Section 3.4, we discussed the design choices between static scheduling and dynamic scheduling of workers. Figure 18 demonstrates the effectiveness of HERMES under static scheduling and dynamic scheduling respectively. Figures 19-22 are a more detailed analysis, demonstrating the time series of energy samples as the result of static scheduling and dynamic scheduling respectively. The "shape" of the time series are clearly dependent on the nature of the benchmarks and their settings (such as the number of workers). For each benchmark

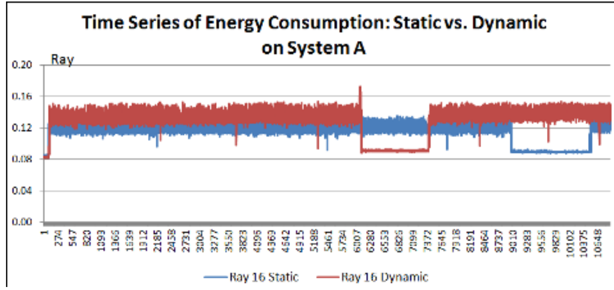

Figure 21: Static vs. Dynamic Scheduling (time series, Ray, 16 workers, System A)

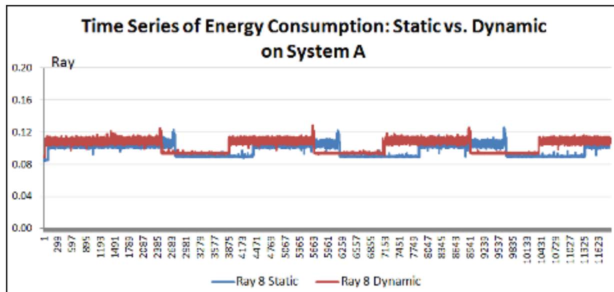

Figure 22: Static vs. Dynamic Scheduling (time series, Ray, 8 workers, System A)

with the same number of workers, the execution of static scheduling and that of dynamic scheduling do display similar patterns. Note that in each figure, the two time series are from different executions. For parallel programs with significant non-determinism, it should not be surprising that two executions of the same program do not "spike" at the same time.

As demonstrated in the figures, dynamic scheduling incurs a slightly higher level of energy consumption. We believe this is due to the overhead needed for setting/resetting affinity to workers for each WORK invocation. We discussed this topic in Sec. 3.4. We investigated into a large number of time series, but found no evidence static scheduling led to significant imbalance (*e.g.* times series with drastically different patterns from their dynamic scheduling counterparts).

5. Related Work

Energy efficiency in work-stealing run-times is an emerging problem that has so far received little attention. The only prior work we know of is a short essay [26] that called for the coordination between the thief and the victim to improve energy efficiency. We now summarize related work in two more established areas: optimization of work stealing run-times and energy efficiency of multi-threaded programs.

Improving various aspects of work-stealing system efficiency has been a central issue throughout the development of work-stealing systems. Indeed, the original work stealing algorithm [16] was designed for load balancing, with direct impact on the performance of multi-threaded systems. A-Steal [2] is an adaptive thread scheduler to take parallelism feedback into account at scheduling time. Dinan et. al [13] improved the scalability of work stealing run-times through optimizations ranging from lock reduction to work splitting. SLAW [18] is a work-stealing scheduler with adaptive scheduling policies based on locality information. AdaptiveTC [36] improves system performance through adaptive thread management at thread creation time. BWS [14] improves system throughput and fairness in time-sharing multicore systems. Kumar *et. al.* [23] applies run-time techniques such as dynamic compilation and speculative optimization to further reduce the overhead in the context of managed X10. Acar *et. al.* [1] designed a non-shared-memory model to replace the locking model based on shared memory. Performance characterization of Intel TBB with work stealing was systematically conducted by Wu *et. al.* [11]. Bender and Rabin [4] formally analyzed the performance of work stealing systems on top of a heterogeneous platform, where parallel units may operate on different, yet fixed, frequencies.

There is a large body of work studying the energy efficiency of multi-threaded programs on parallel architectures. On the architecture level, Iyer and Marcelescu [22] studied the impact of DVFS on multi-clock-domain architectures. Wu *et. al.* [38] designed a DVFS-based strategy where the interval of DVFS use is adaptive to recent instance issue queue occupancy. Magklis *et. al.* [27] designed a profiling-based DVFS algorithm on CPUs with multiple clock domains. On the OS level, numerous efforts exist to apply DVFS for energy management, starting with the seminal work by Weiser *et. al.* [37]. Recent examples include CPU Miser [17] (DVFS based on job workload in clusters) and Dhiman *et. al.* [12] (DVFS based on online learning). Beyond DVFS, effective approaches for energy management of multi-threaded programs include thread migration [33], a combination of thread migration and DVFS [8], and software/hardware approximation [15, 34]. The boundary between architecture and OS for energy efficiency is often blurred. For example, Merkel et. al. [29] designed an energy-aware scheduling policy for thermal management, with inputs from hardware performance counters. Further afield, energy efficiency can also be achieved through compiler optimization (*e.g.*[19, 39]) and language designs (*e.g.* [3, 9, 34, 35]).

6. Conclusion

This paper introduced HERMES, a novel and practical solution for improving energy efficiency of work-stealing applications. HERMES addresses the problem through judicious tempo control over workers, guided by a unified workpath-sensitive and workload-sensitive algorithm. HERMES only requires mild changes to the work stealing runtime. With

no changes necessary for the underlying architectures, OS, or higher-level programming models, the minimalistic approach can still yield significant energy savings with little performance overhead.

References

[1] ACAR, U. A., CHARGUERAUD, A., AND RAINEY, M. Scheduling parallel programs by work stealing with private deques. In *PPoPP '13* (2013), pp. 219–228.

[2] AGRAWAL, K., HE, Y., AND LEISERSON, C. E. Adaptive work stealing with parallelism feedback. In *PPoPP '07* (2007), pp. 112–120.

[3] BAEK, W., AND CHILIMBI, T. M. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI'10* (2010), pp. 198–209.

[4] BENDER, M. A., AND RABIN, M. O. Scheduling cilk multithreaded parallel programs on processors of different speeds. In *SPAA '00* (2000), pp. 13–21.

[5] BLELLOCH, G., FINEMAN, J., GIBBONS, P., KYROLA, A., SHUN, J., TANGWONSAN, K., AND SIMHADRI, H. V. Problem based benchmark suite, 2012.

[6] BLUMOFE, R. D. *Executing Multithreaded Programs Efficiently*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA 02139, 1995.

[7] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (Sept. 1999), 720–748.

[8] CAI, Q., GONZÁLEZ, J., MAGKLIS, G., CHAPARRO, P., AND GONZÁLEZ, A. Thread shuffling: combining dvfs and thread migration toreduce energy consumptions for multi-core systems. In *ISLPED '11* (2011), pp. 379–384.

[9] COHEN, M., ZHU, H. S., EMGIN, S. E., AND LIU, Y. D. Energy types. In *OOPSLA'12* (October 2012).

[10] CONG, G., KODALI, S., KRISHNAMOORTHY, S., LEA, D., SARASWAT, V., AND WEN, T. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP'08* (2008), pp. 536–545.

[11] CONTRERAS, G., AND MARTONOSI, M. Characterizing and improving the performance of intel threading building blocks. In *IEEE International Symposium on Workload Characterization (2008)* (2008), pp. 57–66.

[12] DHIMAN, G., AND ROSING, T. S. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *ISLPED '07* (2007), pp. 207–212.

[13] DINAN, J., LARKINS, D. B., SADAYAPPAN, P., KRISHNAMOORTHY, S., AND NIEPLOCHA, J. Scalable work stealing. In *SC '09* (2009).

[14] DING, X., WANG, K., GIBBONS, P. B., AND ZHANG, X. Bws: balanced work stealing for time-sharing multicores. In *EuroSys '12* (2012), pp. 365–378.

[15] ESMAEILZADEH, H., SAMPSON, A., CEZE, L., AND BURGER, D. Architecture support for disciplined approximate programming. In *ASPLOS'12* (2012), pp. 301–312.

[16] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The implementation of the cilk-5 multithreaded language. In *PLDI '98* (1998), pp. 212–223.

[17] GE, R., FENG, X., CHUN FENG, W., AND CAMERON, K. Cpu miser: A performance-directed, run-time system for power-aware clusters. In *ICPP'07* (2007), pp. 18–18.

[18] GUO, Y., ZHAO, J., CAVÉ, V., AND SARKAR, V. Slaw: A scalable locality-aware adaptive work-stealing scheduler. In *IPDPS* (2010), pp. 1–12.

[19] HSU, C.-H., KREMER, U., AND HSIAO, M. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *ISLPED'01* (2001), pp. 275–278.

[20] INTEL. Threading Building Blocks, `http://threadingbuildingblocks.org/`.

[21] INTEL. Intel cilk plus, 2013.

[22] IYER, A., AND MARCULESCU, D. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *ICCAD '02* (2002), pp. 379–386.

[23] KUMAR, V., FRAMPTON, D., BLACKBURN, S. M., GROVE, D., AND TARDIEU, O. Work-stealing without the baggage. In *OOPSLA '12* (2012), pp. 297–314.

[24] LEA, D. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande* (2000), JAVA '00, pp. 36–43.

[25] LEIJEN, D., SCHULTE, W., AND BURCKHARDT, S. The design of a task parallel library. In *OOPSLA '09* (2009), pp. 227–242.

[26] LIU, Y. D. Green thieves in work stealing. In *ASPLOS'12 (Provactive Ideas session)* (2012).

[27] MAGKLIS, G., SCOTT, M. L., SEMERARO, G., ALBONESI, D. H., AND DROPSHO, S. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *ISCA '03* (2003), pp. 14–27.

[28] MARLOW, S., PEYTON JONES, S., AND SINGH, S. Runtime support for multicore haskell. In *ICFP '09* (2009), pp. 65–78.

[29] MERKEL, A., AND BELLOSA, F. Balancing power consumption in multiprocessor systems. In *EuroSys '06* (2006), pp. 403–414.

[30] MICHAEL, M. M., VECHEV, M. T., AND SARASWAT, V. A. Idempotent work stealing. In *PPoPP '09* (2009), pp. 45–54.

[31] MOHR, E., KRANZ, D. A., AND HALSTEAD, JR., R. H. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM conference on LISP and functional programming* (1990), LFP '90, pp. 185–197.

[32] PROKOPEC, A., BAGWELL, P., ROMPF, T., AND ODERSKY, M. A generic parallel collection framework. In *Euro-Par'11* (2011), pp. 136–147.

[33] RANGAN, K. K., WEI, G.-Y., AND BROOKS, D. Thread motion: fine-grained power management for multi-core systems. In *ISCA '09* (2009), pp. 302–313.

[34] SAMPSON, A., DIETL, W., FORTUNA, E., GNANAPRAGASAM, D., CEZE, L., AND GROSSMAN, D. EnerJ: Ap-

proximate Data Types for Safe and General Low-Power Computation. In *PLDI'11* (June 2011).

[35] SORBER, J., KOSTADINOV, A., GARBER, M., BRENNAN, M., CORNER, M. D., AND BERGER, E. D. Eon: a language and runtime system for perpetual systems. In *SenSys'07* (2007), pp. 161–174.

[36] WANG, L., CUI, H., DUAN, Y., LU, F., FENG, X., AND YEW, P.-C. An adaptive task creation strategy for work-stealing scheduling. In *CGO '10* (2010), pp. 266–277.

[37] WEISER, M., WELCH, B., DEMERS, A., AND SHENKER, S. Scheduling for reduced cpu energy. In *OSDI '94* (1994).

[38] WU, Q., JUANG, P., MARTONOSI, M., AND CLARK, D. W. Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors. In *HPCA* (2005), pp. 178–189.

[39] XIE, F., MARTONOSI, M., AND MALIK, S. Compile-time dynamic voltage scaling settings: opportunities and limits. In *PLDI '03* (2003), pp. 49–62.