Energy-Efficient Compilation of Irregular Task-Parallel Loops

RAHUL SHRIVASTAVA and V. KRISHNA NANDIVADA, IIT Madras

Energy-efficient compilation is an important problem for multi-core systems. In this context, irregular programs with task-parallel loops present interesting challenges: the threads with lesser work-loads (*non-critical*-threads) wait at the join-points for the thread with maximum work-load (*critical*-thread); this leads to significant energy wastage. This problem becomes more interesting in the context of multi-socket-multi-core (MSMC) systems, where different sockets may run at different frequencies, but all the cores connected to a socket run at a single frequency. In such a configuration, even though the load-imbalance among the cores may be significant, an MSMC-oblivious technique may miss the opportunities to reduce energy consumption, if the load-imbalance across the sockets is minimal. This problem becomes further challenging in the presence of mutual-exclusion, where scaling the frequencies of a socket executing the non-critical-threads can impact the execution time of the critical-threads. In this article, we propose a scheme (X10Ergy) to obtain energy gains with minimal impact on the execution time, for task-parallel languages, such as X10, HJ, and so on. X10Ergy takes as input a loop-chunked program (parallel-loop iterations divided into chunks and each chunk is executed by a unique thread). X10Ergy follows a mixed compile-time + runtime approach that (i) uses static analysis to efficiently compute the work-load of each chunk at runtime, (ii) computes the "remaining" work-load of the chunks running on the cores of each socket at regular intervals and tunes the frequency of the sockets accordingly, (iii) groups the threads into different sockets (based on the remaining work-load of their respective chunks), and (iv) in the presence of atomic-blocks, models the effect of frequency-scaling on the critical-thread. We implemented X10Ergy for X10 and have obtained encouraging results for the IMSuite kernels.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; • **Software and its engineering** → **Compilers**;

Additional Key Words and Phrases: Irregular task parallel programs, multi-socket-multi-core systems, DVFS

ACM Reference format:

Rahul Shrivastava and V. Krishna Nandivada. 2017. Energy-Efficient Compilation of Irregular Task-Parallel Loops. *ACM Trans. Archit. Code Optim.* 14, 4, Article 35 (November 2017), 29 pages.
https://doi.org/10.1145/3136063

1 INTRODUCTION

The advent of multi-core systems has brought the dual challenges of high performance and low-energy consumption, firmly to the front. The projected increase in expenses toward the energy needs of servers (Koomey et al. 2009) makes it quite important to reduce the energy consumption of parallel programs. For many types of workloads, dynamic voltage and frequency-scaling (DVFS) can be applied to obtain energy savings (Yuki and Rajopadhye 2013), but at the cost of

Authors' addresses: R. Shrivastava, Samsung R&D Institute Bangalore; email: rahuls@cse.iitm.ac.in; V. K. Nandivada, Dept of CSE, IIT Madras,Chennai; email: nvk@cse.iitm.ac.in.

https://doi.org/10.1145/3136063

```
finish for(k = 0; k < n; k++){
 async {
  if(V(k).whiteNeigCnt!=0){
   for(j=0;j<V(k).N2v.size();j++)
    if(Colr(V(k).N2v.get(j))==WHITE){
     //Mx=max white-neighbor count in
          N2v
    }
    if(Mx==V(k).W)//'k' a candidate?
      ...
   } }  /* async */ } /* finish */
```

(a) Input code

```
chSz=(n + nThreads - 1)/nThreads;
finish for(i=0;i<n;i+=chSz){//ACLoop
 async {//chunk
  for(k=i;k<min(i+chSz,n);k++){//chLoop
   if(V(k).whiteNeigCnt != 0){
    for(j=0;j<V(k).N2v.size();j++){
     if (Colr(V(k).N2v.get(j))==WHITE){
     //Mx=max white-neighbor count in N2v
     }
     if(Mx==V(k).W)//'k' a candidate?
       ...
    } } } /* chLoop */
 } /* chunk */
}  /* ACLoop */
```

(b) Chunked version

Fig. 1. A sample parallel-loop from the DS kernel of IMSuite and its chunked version. V is the array of vertices.

increased execution time. In this article, we present a scheme to reduce the energy consumption of task-parallel programs running on multi-core systems, without significantly increasing the execution time.

Optimizing task-parallel programs to reduce energy consumption is an interesting problem, especially in the context of *multi-socket-multi-core* (MSMC) systems, such as Sandy Bridge (Rotem et al. 2012), Ivy Bridge (Intel 2017), Opteron (AMD 2016), and so on, where all the cores on a socket run at the same frequency. This problem becomes more challenging for irregular programs with task-parallel loops (ITP programs, in short), where the work-loads of different tasks can vary significantly and it is not possible to completely reason about these work-loads statically. The resulting load-imbalance among the threads, executing these tasks on MSMC systems supporting DVFS, provides some unique opportunities for energy reduction. We will use an example to illustrate.

Figure 1(a) shows a snippet of the code (in X10 (Saraswat et al. 2014)) from the DS kernel of IMSuite (Gupta and Nandivada 2015). In X10, an `async` construct creates an asynchronous task that may run in parallel with the current task. The `finish` construct acts as a join point that waits for all the tasks spawned within the `finish`-block to terminate. The DS kernel computes a dominating-set of nodes of an input graph using a distributed greedy (approximation) algorithm (Wattenhofer 2011). It uses the intuition that if a node `k` has many neighbors as members of the dominating-set d, then `k` is less likely to be a member of d. Corresponding to each node of the graph, the algorithm maintains a set of "white" neighbors (nodes not yet processed). For each node `k`, the algorithm checks if `k`'s white-neighbor count (`V(k).W`) is more than that of any of its white neighbors at a distance of at-most two. If so, then `k` is a potential candidate to be added to the dominating-set. The snippet of the code shown in Figure 1(a) performs this check. Figure 1(b) shows the corresponding code after loop-chunking (Nandivada et al. 2013). This optimization creates as many tasks as the number of threads (*nThreads*—typically set to #hardware cores); each such task consists of non-overlapping groups of iterations of the original loop; this is an essential optimization to keep the program scalable. For example, on an Intel dual socket system (64GB RAM, 8 cores/socket, with cores 0–7 and 8–15 connected to sockets 0 and 1, respectively), running at the highest system frequency (hereafter, called as MAXFREQ) for an input graph of 32K nodes, the original program terminates with heap-overflow, but the chunked program completes successfully (takes ≈21s, and consumes 1464 Joules—measured using RAPL interface (Manual 2016)).

We now analyze the chunked code to identify opportunities for reduction in energy consumption. Even though the coarser-grained tasks (each executed by a different thread) created by loop-chunking may more or less execute the same number of iterations, these tasks may not be fully balanced with respect to the amount of work done by each of them, especially in the context of

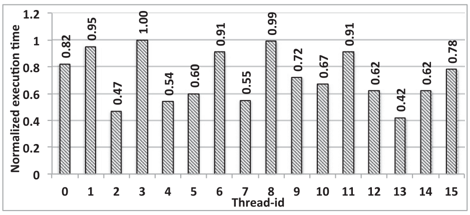


Fig. 2. Execution time of the threads normalized against that of thread-id 3 for the snippet shown in Figure 1(b).

ITP programs. Note that for the code snippet shown in Figure 1(b), different tasks may execute different number of serial for-loop iterations (Line 6), and may take different paths in if-else statements (Lines 5, 7, 10). Figure 2 shows the execution times of the threads executing each task created in Figure 1(b), normalized with respect to that of the task with the highest execution time (task executed by thread with thread-id 3). This load imbalance causes some threads that are executing tasks with smaller work-loads (called non-critical-threads) to wait at the join-point for the thread that is executing the task with the largest work-load (critical-thread), which in turns leads to wastage of energy.

There have been many efforts to balance the load among the threads (Thoman et al. 2012; Gautier et al. 2013), but their effectiveness is quite limited in the context of ITP programs. This is because in many of the ITP programs a large part of the work is done serially by a few of the tasks and it may not be semantically correct to divide those serial parts among the parallel tasks. For example, in Figure 1(a), the loop at Line 4 is the main reason for the load-imbalance among the tasks (typically a few tasks execute significantly more iterations in the loop, compared to the rest). The only way to completely balance the load among the tasks would be to divide the serial-iteration of the tasks with very large values of `N2v.size()`, among the tasks with lower values of `N2v.size()`. However, the iterations of this serial loop are not independent. Thus, it would not be semantics preserving to divide this loop among multiple tasks. This makes it hard to do load balancing, even using a dynamic scheme like work-stealing, which mainly work at a task-level granularity (not at the granularity of inner serial code). A similar argument applies to the chunked code shown in Figure 1(b). Thus, instead of balancing the work-load, many prior approaches (Liu et al. 2005; Rakvic et al. 2010; Cai et al. 2008; Ozturk et al. 2013; Chen et al. 2014; Rauber and Rünger 2015) reduce the energy consumption by reducing the frequency of the cores on which the non-critical-threads are mapped such that all the threads reach the join-point at the same time, thereby reducing the energy consumption without any increase in the execution time. Many of these schemes (Liu et al. 2005; Rakvic et al. 2010; Cai et al. 2008; Chen et al. 2014; Rauber and Rünger 2015) assume that the frequency of each core can change independently, which is not true in case of MSMC systems. We can extend these schemes to MSMC systems by assuming that the work-load of the socket is equal to that of the thread executing the task with maximum work-load on that socket, and using the respective schemes to reduce frequencies of the "non-critical"-sockets (sockets not running the critical-thread)—we call such an extension as MSMC-oblivious scheme. However, such an extension does not always translate to energy gains because of the way the threads are scheduled on the socket. For example, in Figure 2, there is an overall load-imbalance in the program but the two threads executing the tasks with the highest execution times (threads with ids 3 and 8) are mapped to two different sockets (note: here thread i is mapped to core i). Since the execution times for tasks executed by threads 3 and 8 are almost equal, the MSMC-oblivious schemes assume that there is no load-imbalance among the sockets and thus perform very minimal or no frequency-scaling. Similarly, the MSMC-oblivious scheme of Li et al. (2004) runs

Technique	Supports MSMC	Handles Atomic-blocks	Work-load estimation	Optimize for	Target systems
Ozturk et al. [2013]	✓	✗	compile-time	energy	homogeneous
Rakvic et al. [2010]	✗	✗	runtime	energy	homogeneous
Liu et al. [2005]	✗	✗	runtime	energy	homogeneous
Chen et al. [2014]	✗	✗	runtime	energy	homogeneous
Jibaja et al. [2016]	✗	✗	runtime	energy+time	heterogeneous
Jimborean et al. [2014]	✗	✗	NA	energy	homogeneous
Rauber and Rünger [2015]	✗	✗	compile-time	energy	homogeneous
Ribic and Liu [2014]	✗	✗	runtime	energy	homogeneous
Salami et al. [2014]	✗	✗	runtime	temperature	homogeneous
X10Ergy	✓	✓	compile-time+runtime	energy	homogeneous

Fig. 3. Comparison of X10Ergy with related works.

the non-critical-sockets at MAXFREQ and switches them off once all the tasks running on these sockets reach the join-point. This scheme does not lead to much benefits either as there is not much socket-level load-imbalance. Further Cai et al. (2008) and Rakvic et al. (2010) show that even in the presence of some load-imbalance, this simple scheme is not much beneficial. Thus, we see that in the context of MSMC systems, there is less scope to reduce the frequency of the non-critical-sockets, using MSMC-oblivious schemes.

The recent work of Ozturk et al. (2013) reduces the energy consumption in MSMC systems, but it assumes that the work-load is known statically (not applicable in the context of ITP programs, where the sources of load-imbalance are input dependent loop bounds, if-conditions, and so on).

In this article, we aim to reduce the energy consumption of ITP programs running on MSMC systems, without significantly impacting their execution times. We first list three *key* intuitions that form the basis of our solution: (i) Though it is hard to statically measure the work-load of the tasks for ITP programs, it is more feasible to estimate the same efficiently, using a mixed static + runtime approach. For example, for a normalized (Muchnick 1997) for-loop with an input dependent loop-bound, if we can compute the work-load of the loop-body statically, then we can compute the work-load of the loop at runtime, by multiplying the loop-bound (known at runtime) with the statically computed work-load of the loop-body. (ii) The load-imbalance among the threads can vary during the execution of their assigned tasks; thus, it may be beneficial to perform frequency-scaling multiple times during the life time of the tasks. (iii) For MSMC systems, segregating the threads into different sockets (such that the load-imbalance among the sockets is maximized), can help in creating more opportunities for frequency-scaling. Importantly, all these points must be handled in a way such that the runtime overheads are minimal.

Achieving energy reduction with minimal increase in execution time becomes more challenging in the presence of *atomic-blocks*. An atomic-block in languages like X10 (or an isolation-block in HJ) provides mutual-exclusion. Reducing the frequency of the non-critical-sockets slows down the atomic-blocks executed by the corresponding threads. This in turn may increase the waiting time of the critical-thread at those atomic-blocks. Consequently, even though the critical-socket is running at MAXFREQ, the execution time of the critical-thread may increase. Thus, it is important to model the impact of atomic-blocks on the execution time of the critical-thread.

In this article, we present X10Ergy, a mixed compile-time + runtime technique to reduce the energy consumption of ITP programs, for task-parallel languages like X10. To the best of our knowledge, this is the first work that reduces the energy consumption of ITP programs, on MSMC systems, with minimal increase in execution time, even in the presence of atomic-blocks; Figure 3 summarizes some of the important prior works in the context of the challenges discussed above. An important point to note is that we can apply our proposed optimizations to derive additional energy gains, after the completion of the pass to load-balance the threads by traditional approaches (such as loop-chunking). For example, in contrast to the chunked version of the DS kernel, the X10Ergy version takes 21.2s (1% more) and consumes 1171 Joules (20% less). Though our proposed solution is presented for X10, it can be extended to other task-parallel languages that support parallel-loops and atomic-blocks (for example, HJ (Cavé et al. 2011; OpenMP 2013; Chapel 2005), and so on).

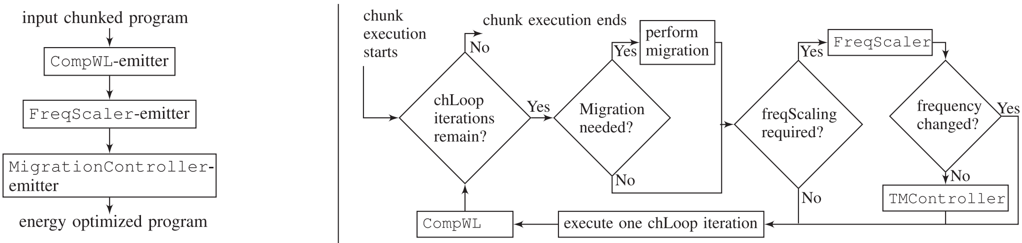
Our contributions:

- We propose a technique to estimate the work-load of the tasks by identifying two different parts of the program: code with input independent work-load and code with input dependent work-load. The work-load of input independent parts is calculated through static analysis, and combined with the work-load of input dependent parts, computed at runtime to find the work-load of each task.
- Considering the continuously decreasing work-loads of the tasks (all not at the same rate), we propose an algorithm to compute the remaining work-loads and perform frequency-scaling at regular intervals based on these remaining work-loads (instead of the initial estimated work-load (Ozturk et al. 2013) or the work-done so far (Rakvic et al. 2010; Cai et al. 2008; Chen et al. 2014)).
- Considering the special nature of the MSMC systems, especially in the context of ITP programs, we propose an algorithm that performs thread-migration at different intervals to maximize the load-imbalance among the sockets, which in turn improves the opportunities for frequency-scaling.
- To handle atomic-blocks, we propose a technique that takes into consideration the increase in waiting time of the critical-thread (due to frequency-scaling of the non-critical-sockets). Our technique uses a user-specified parameter ($wtTmP$) to fine-tune the frequencies of the non-critical-sockets, so the percentage increase in the execution time of the critical-thread is bounded by $wtTmP$.
- We have implemented X10Ergy in the x10-v-2.4 compiler and evaluated it against the Baseline versions (with -NO_CHECKS flag) and two prior works: the meeting-points based optimization (MP-OPT) of Rakvic et al. (2010) and the work-load-aware optimization (EEWA-OPT) of Chen et al. (2014). We show that on average, on IMSuite benchmarks (i) compared to Baseline, X10Ergy reduces the energy consumption by 15% with 2% increase in execution time. (ii) Though X10Ergy consumes approximately equal amount of energy than MP-OPT, it does not incur the execution time overhead (17%) incurred by MP-OPT. (iii) Compared to EEWA-OPT, X10Ergy reduces the energy consumption by 4% and does not incur the execution time overhead (9%) incurred by EEWA-OPT.

2 BACKGROUND

We give a brief background of some of the topics related to this manuscript.

- *Normalized Loop* is a for-loop in which the loop index variable starts at 0, is incremented by 1 after each iteration, and is bound by a loop-invariant expression (the *loop-bound* expression). For example, in the loop `‘‘for(i=0;i<V(k).f();i++) S’’`, `V(k).f()` is the loop-bound expression.
- In X10, each task is executed by a single thread—from start to end. The initial count of threads can be set using the environment variable X10_NTHREADS (typically set to the number of available cores). Starting from 0, the X10 runtime assigns a thread-id to each thread. In the rest of the article, we use “thread-id” of a thread, to refer to the X10 runtime assigned thread-id.
- Loop chunking (Nandivada et al. 2013) is one of the most effective ways to extract useful parallelism from the programmer specified ideal parallelism. We use the following terms to denote different parts of the chunked loop. (i) The *ACLoop* (Async creator loop) creates



(a) Compile-time interaction.

(b) Runtime interaction of the X10Ergy components with a chunk.

Fig. 4. Block diagrams.

nThreads number of asyncs and waits for them to terminate (Example ACLoop: Lines 2–14, Figure 1(b)). (ii) The *chunk* is the async block inside the ACLoop (example chunk: Lines 3–13, Figure 1(b)). (iii) The *chLoop* is the loop (contained in the chunk) that executes a block of asyncs of the unchunked loop serially (example chLoop: Lines 4–12, Figure 1(b)). (iv) The *chunk-size* of a chLoop is the maximum number of iterations executed by the loop.

We use a five-tuple ⟨n,cS,Spre,S,Spost⟩ to denote an ACLoop, where n is the loop bound of the ACLoop, cS is the chunk-size, Spre is the code in the chunk present before the chLoop, S is the body of chLoop, and Spost is the code in the chunk after chLoop. In this article, we restrict ourselves to the class of X10 programs where there is no nesting of ACLoops.

- *equi-acls*: Two ACLoops F_1 and F_2 are termed as equi-acls, if loop-bound(F_1) = loop-bound(F_2) and loop-bound(chLoop(F_1)) = loop-bound(chLoop(F_2)).
- *A post-dom$_F$ B*: In a CFG, A post-dom$_F B$, if each path from B to F in the CFG must pass through A. This is a generalization of the traditional post-dominance (post-dom) relation (Muchnick 1997).
- *Energy reduction via MP-OPT*: Rakvic et al. (2010) propose a scheme (called meeting-point optimization) for energy reduction, in which for each thread they maintain a counter representing the number of parallel-for-loop iterations completed by that thread. The critical-thread is the thread with the least counter value. After every 10 iterations, each thread sets its frequency = MAXFREQ× the ratio of the counters of the critical-thread and itself. We call this scheme MP-OPT.
- *Energy reduction via EEWA-OPT*: For each parallel-loop L, the feedback-based scheme proposed by Chen et al. (2014) first records the execution times of each task (created in L), by running all the cores at MAXFREQ. If L is embedded inside a serial loop, then in the jth instance of L, the ith task is executed at a frequency = MAXFREQ×(recorded-execution-time(i) at iteration ($j-1$)/critical-task-execution-time at iteration 0). We call this scheme EEWA-OPT.

3 X10ERGY

We now present a new scheme (X10Ergy) to optimize loop-chunked programs for energy, with minimal impact on the execution time. X10Ergy uses a mixed (compile-time + runtime) approach. At compile-time, for each ACLoop (see Section 2), it emits instrumentation code that at runtime, (i) estimates the work-load of each chunk created inside the ACLoop, and (ii) based on the estimated work-loads, uses frequency-scaling and thread migration to reduce the energy consumption.

Figure 4(a) shows the block diagram of the interaction of the compile-time components of X10Ergy; it consists of three components: (1) CompWL-emitter: emits code to estimate the initial and

Abbreviation	Definition
ACLoop	Async creator loop (creates as many asyncs as there are number of hardware threads available).
chunk	Async block inside the ACLoop.
chLoop	Loop contained in the chunk that executes a block of asyncs of the unchunked loop serially.

Fig. 5. Abbreviations used in Section 3.1 and further.

remaining work-load for each chunk. (2) `FreqScaler`-emitter: emits code to tune the frequency of the sockets based on the remaining work-load of all the chunks. (3) `MigrationController`-emitter: emits code to (i) identify the threads for migration (based on the remaining work-load of all the chunks) by invoking a function `TMController`, and (ii) perform the migration of the threads.

The codes emitted by the different emitters are invoked during the execution of the chunk. Figure 4(b) shows the block diagram of the runtime-interaction of the different components generated by X10Ergy (`CompWL`, `FreqScaler`, and `MigrationController`) with any chunk of the ACLoop. Before executing any ACLoop, `CompWL` estimates the initial work-load of all its chunks (not shown in the figure). In each chunk, before executing an iteration of the chLoop, a part of the code emitted by `MigrationController` is invoked to migrate the current thread, if it is marked for migration by `TMController`. `CompWL` is invoked after each iteration of the chLoop to update the remaining work-load of the chunk. In contrast, `FreqScaler` and `TMController` are invoked at variable intervals depending on the load imbalance among the threads.

We now give details regarding each of the components of X10Ergy shown in Figure 4(a). For the ease of presentation, we assume that the input code does not contain any atomic-blocks. Section 4 discusses how our proposed techniques are extended to handle atomic-blocks. For easy reference, we list the abbreviations and variables used in the upcoming sections in Figures 5, 11, and 13.

3.1 `CompWL`-emitter

The `CompWL`-emitter performs three main tasks: (1) generates code to estimate the initial work-loads of all the chunks; (2) identifies the appropriate program-points and emits the code generated by Task 1; and (3) emits the code to update the remaining work-load of all the chunks at regular intervals during chunk execution. In this article, we use the terms cost and work-load interchangeably.

Task 1: To generate code to estimate the initial cost of all the chunks, we first generate an expression denoting an upper bound on the cost of each iteration of their chLoops (abbreviations introduced in this section are listed in Figure 5). Note that since each chunk (in an ACLoop) executes the same piece of code, every chLoop in any such chunk will have the same syntactic cost-expression. For example, consider the ACLoop at Line 10 in Figure 6(a) (a synthetic code depicting a typical pattern found in the IMSuite kernels). For the chLoop (spanning Lines 12–17) the body of each iteration spans the Lines 13–16. The cost-expression for each such iteration is `V(k).list2.size()*`$L_{13} + C_{16}$. We use L_i as the cost of executing one iteration of the loop at Line i (including the cost of loop-body, predicate evaluation and loop-index variable increment), and C_j as the cost of executing Line j (see Section 5 for a discussion on computing the static costs). Figure 7 presents the pseudo code, of the function `getWL`, used by `CompWL`-emitter to compute the cost-expression of the chunk.

The function `getWL` maintains the cost-expression of the chunk in two parts: an input dependent part (in a string variable *dCost*), and an input independent part (in an integer variable *iCost*). The final expression denoting the cost of the chunk is obtained by concatenating *dCost* and *iCost*; we use the || operator to concatenate two strings. The function `getWL` iterates over all the statements of the chunk (Line 4). If the statement is a *loop*, then `getWL` generates a string denoting

```
 1 while(some condition){
 2  finish for(i=0;i<n;i+=chunkSz){
 3   async{
 4    for(k=i;k<min(i+chunkSz,n);k++){
 5     for(j=0;j<V(k).list1.size();j++){
 6      S1/*calculates some vertex x*/
 7      V(k).list2.add(x);
 8     } }
 9   } /* async */ }/* finish */
10  finish for(i=0;i<n;i+=chunkSz){
11   async{
12    for(k=i;k<min(i+chunkSz,n);k++){
13     for(j=0;j<V(k).list2.size();j++){
14      S2
15     }
16     S3
17    }
18   } /* async */ } /* finish */
19   ...more finish blocks
20 } /* while */
```

(a) A typical pattern found in many IMSuite kernels.

```
 1 finish for(i=0;i<n;i+=chunkSz){
 2  async{
 3   ch=i/chunkSz;sum=0;
 4   for(k=i;k<min(i+chunkSz,n);k++){
 5    sum=sum+L_5*V(k).list1.size();
 6   }
 7   oriWL_ACL1(ch)=WL_ACL1(ch)=sum;
 8  } /* async */ } /* finish */
 9 while(some condition){
10  finish for(i=0;i<n;i+=chunkSz){
11   async{
12    ch=i/chunkSz;sum=0;
13    for(k=i;k<min(i+chunkSz,n);k++){
14     for(j=0;j<V(k).list1.size();j++){
15      S1/*calculates some vertex x*/
16      V(k).list2.add(x);
17     }
18     sum+=L_13*V(k).list2.size()+C_16;
19     WL_ACL1(ch)-=L_5*V(k).list1.size();
20    }
21    WL_ACL2(ch)=sum;
22   } /* async */ } /* finish */
23  finish for(i=0;i<n;i+=chunkSz){
24   async{
25    ch=i/chunkSz;sum=0;
26    for(k=i;k<min(i+chunkSz,n);k++){
27     for(j=0;j<V(k).list2.size();j++){ S2 }
28     S3
29     WL_ACL2(ch)-=L_13*V(k).list2.size()+C_16;
30    }
31    WL_ACL1(ch)=oriWL_ACL1(ch);//restore
32   } /* aysnc */ } /* finish */
33   ...more finish blocks
34 } /* while */
```

(b) Transformed code

Fig. 6. Code transformation by CompWL. S1, S2, S3 represent blocks of code. The changes are shown in **bold**.

an expression that multiplies the loop-bound with the cost of the *loop* body (Lines 5–9), including the loop-predicate cost and appends this string to *dCost*. This multiplication ensures that at runtime, two chunks executing the same syntactic piece of code with different values for the loop-bound expressions will have different work-loads. Note that the function getLoopBound returns the loop-bound expression in terms of the visible fields (of the current class or globals), and if the desired loop-bound expression cannot be computed, then it conservatively returns a constant C_L (typically, set to a large value).

If the statement is a conditional (Lines 10–14), then getWL generates an expression to compute the conservative worst-case cost of the statement (including the cost of evaluating the predicate). The macro MAX($B1, B2$), expands to ‘‘(’’ $\|$ $B1$ $\|$ ‘‘$\geq$ ’’$\|B2$ $\|$ ‘‘)?’’$\|B1\|$ ‘‘:’’$B2$. The translation of a switch statement or a simple if-then statement is handled similarly.

If the current statement is a *simple-statement* (neither a loop, nor a conditional), then getWL increments *iCost* by the static cost of the current statement; see Section 5 for a discussion on computing the static cost. Note that if the statement is a *loop* over simple-statements and its loop-bound expression can be evaluated statically, then it is handled like a simple statement.

The function getWL returns a string denoting the work-load of one iteration of the chLoop, which is accumulated at runtime by adding the contributions of every iteration to get the total estimate of the work-load of the chunk. At runtime, we store this chunk's workload-sum (say

```
Function String getWL($Node\ B$)
    // Returns an expression (as a string) to estimate
    the work-load of one iteration of chLoop, whose
    body given by $B$.
    int $iCost$ := 0; // input independent
    String $dCost$:="0";// input dependent
    foreach $S \in B.statements$ do
        case $S$ is a loop : do
            String $lBound$ := getLoopBound($S$);
            String $pCost$ := predEvalCost;
            String $lCost$ := getWL($S$.body);
            $dCost$ := $dCost$ || "+" || $lBound$ || "*("
                        || $pCost$ || "+" || $lCost$ || ")";
        case $S$ is an if-then-else statement : do
            String $pCost$ := predEvalCost;
            String $tCost$ := getWL($S.thenBody$);
            String $eCost$ := getWL($S.elseBody$);
            $dCost$ := $dCost$ || "+" || $pCost$ || "+"
                        || MAX($tCost$, $eCost$);
        case Otherwise /*a simple-statement*/ : do
            $iCost$ := $iCost$ + $S.iCost$
    return $dCost$ || "+" || Integer.toStr($iCost$);
```

Fig. 7. Get the workload of the chLoop.

```
1  Function wLEmitter(chLoop $A$, ACLoop $F$)
2      $E$ := getWL($A$);// computed by Fig.7
3      $Defs$ := Set of defs used in $E$, reaching $A$;
4      $S$ := $IDF(Defs) \cup Defs$;
5      Find $p \in S$ such that $\forall l \in S$, $p$ post-dom$_F$ $l$;
6      if $p$ is inside ACLoop $N$ AND equi-acls($N, F$)
           AND $N \neq F$ then
7          EmitInACLoop($N$,$F$,$E$)
8      else
9          if equi-acl-defs($F, Defs$) then
10             foreach $st \in Defs$ do
11                 EmitInACLoop(ACLoop($st$), $F$, $E$);
12         else EmitAt($p$, $F$, $E$);
13     if $\exists$ a path $P$ from $F$ to $F$ in the mCFG AND the
           vars used in $E$ are not modified in between
       then
14         if $\exists$ ACLoop $R$: $R \in P$ AND equi-acls($R$,$F$)
               AND ($R$ post-dom $F$ OR $R$ dom $F$)
                 AND $R \neq F$ then
15             Emit " $WL_F$(ch) = $oriWL_F$(ch);" as
                 the last statement in chunk($R$).
16         else
17             Emit "for(i=0;i<$nThreads$;i++)
                 $WL_F$(i)=$oriWL_F$(i);" before $F$
```

Fig. 8. Algorithm to identify the appropriate program points and emit work-load estimation code.

indexed by i), in each ACLoop L, in a data structure $WL_L(\texttt{i})$. The code for summing up the work-load and storing the value in $WL_L(\texttt{i})$ is emitted by Task 2.

Task 2: The program-points where the cost expressions computed by Task 1 are emitted impact the overhead resulting from the emitted code. Figure 8 shows the steps followed by CompWL to emit the work-load estimation code at appropriate program-points to minimize the resulting overheads.

Note that, for any ACLoop F, we want to compute the work-load of the chunks present in F, before F starts executing, and these computations can be done in parallel for each chunk. Naively emitting even a parallel-loop to compute the work-load just before F can be quite inefficient: for instance, say F is inside a serial loop, then the task creation and termination overheads for the work-load computing loop can be high. Instead, we want to emit the work-load computing code such that these additional overheads can be minimized. We use Iterated Dominance Frontier (IDF) (Cytron et al. 1991) to achieve this goal and to minimize the program points where work-load estimation code needs to be emitted.

In Figure 8, we first calculate the IDF of the definitions ($Defs$) of the variables that are used in the cost-expression E returned by Task 1. We then select a node p from S ($= Defs \cup IDF(Defs)$) such that for each $l \in S$, p post-dom$_F$ l. This condition ensures that p is the closest reachable node to F among the nodes in S and every node in $Defs$ that reaches E (in A) must pass through p. Emitting the work-load computing code at p ensures that redundant work-load re-computation is avoided. Once p is identified, if p is contained in an ACLoop N, and N and F are equi-acls (see Section 2) then the code to estimate the work-load for each chunk in F is emitted in N using the function EmitInACLoop (Line 6). The function EmitInACLoop (Figure 9) modifies N in such a way that in each chunk ch, it additionally sums up the work-load of each chLoop-iteration of the corresponding chunk in F; the final sum is stored in the corresponding element of that chunk

```
Function void EmitInACLoop (ACLoop N,
                            ACLoop F, String E)
    Say N = ⟨n,ii,cS,Spre,S,Spost⟩;
    Replace N by ⟨n,ii,cS,S₁,S′,S₂⟩, where
     S₁ := Spre‖"int ch=ii/cS,sum=0;"
     S′ := S ‖ " sum = sum + " ‖ E ‖ ";"
     S₂:=Spost‖"WL_F(ch)=oriWL_F(ch)=sum;"
```

Fig. 9. Function EmitInACLoop: in ACLoop N emits code to compute WL_F.

```
Function void EmitAt(program-point p,
                     ACLoop F, String E)
    Say F := ⟨n,ii,cS,Spre,S,Spost⟩;
    After p, emit ⟨n,ii,cS,S₁,S′,S₂⟩, where
     S₁:="int ch=ii/cS,sum=0;"
     S′ := "sum = sum + " ‖ E ‖ ";"
     S₂:="WL_F(ch)=oriWL_F(ch)=sum;"
```

Fig. 10. Function EmitAt: after p create a new ACLoop to compute WL_F.

(WL_F(ch)). Note that the function EmitInACLoop does not create new tasks for computing the work-load and hence avoids the task creation and termination overheads.

If the predicate at Line 6 (Figure 8) fails, then the algorithm checks if all the $Defs$ are contained in ACLoops that are equi-acl to F—we call it the *equi-acl-defs* condition (Line 9, Figure 8). If so, then the function EmitInACLoop is invoked, which emits the work-load estimation-code inside the ACLoops (thereby avoiding additional task creation and termination operations), where the defs are present; Lines 10 and 11, Figure 8. Otherwise, a new ACLoop is emitted after p (Line 12, Figure 8) to calculate the work-load of A; we do so by invoking the function EmitAt (Figure 10).

After the execution of the chunks contained in ACLoop F is finished, the remaining work-loads of all those chunks (stored in WL_F) become zero. If F can be executed again (say, because it is enclosed in a loop), then we may need to reinitialize all the elements of WL_F before F starts executing. For a given ACLoop F, the algorithm (at Line 13) checks if the variables present in the string returned by getWL are not modified anywhere along the path from F to F in the mCFG (a modified CFG, where each ACLoop is treated as a single node). If such a path exists, then it checks whether there exists any ACLoop R such that R and F are equi-acls, and in the mCFG, either R post-dom F, or R dom F. If this is the case, then the code to re-initialize the work-load array (with the array *oriWL*) is emitted as the last statement of the chunk contained in R (Line 15). Otherwise, a serial for-loop that initializes the work-load array is emitted immediately before F (Line 17).

Task 3: The remaining work-load of the chunk (stored in the *WL* array) should be updated continuously, as its value decreases after the execution of each statement in the chunk. Considering the possibly significant overheads resulting from frequent updating of the *WL* array, we only update the *WL* array at the end of each iteration of the corresponding chLoop—this helps us maintain an approximate value of remaining work-load. For a given ACLoop N (say the chunk is indexed by ch, if getWL returns E, then Task 3 emits the following statement after the last statement of the chLoop: ‘‘WL_N(ch)=WL_N(ch)-E;’’.

Example: Consider the snippet in Figure 6(a). For the list objects, we assume that list.add() is a write operation and list.size() is a read operation. Task 1: For the chLoop at Line 4, getWL returns L_5*V(k).list1.size(); and for the chLoop at Line 12, getWL returns $L13$*V(k).list2.size()+C_{16}. Task 2: The algorithm in Figure 8 transforms the running example to the code shown in Figure 6(b). In Figure 6(a), for the chLoop starting at Line 4, V(k).list1 is defined only during the initialization (before the while-loop, not shown in the figure). Thus, the initial work-load of each chunk here remains unchanged across the multiple iterations of the while-loop (Line 1). In such cases, it is enough to evaluate the cost-expression for each such chunk exactly once to avoid redundant computations. The code generated by Task 2 achieves this by emitting the code shown in Lines 1–8 (Figure 6(b)). For the same chunks, Task 2 also emits the code at Line 31 to re-initialize the work-loads of these chunks. Consider the second chLoop of Figure 6(a) at Line 12: V(k).list2 is defined at Line 7. So the estimated initial work-load of the second chunk keeps changing in the iterations of the while-loop. CompWL emits the code at Lines 18

Name	Definition
$curLdr$	stores the chunk-id of the leader (type: shared, atomic).
$done$	per chunk, stores a flag indicating if the chunk has finished execution (type: shared, atomic).
backoff	stores the interval after which `FreqScaler` is called by the current chunk (thread-local).
$STEP$	stores the initial value of the *backoff* interval (user specified parameter).
socketRWL	per socket, stores the max remaining work-load among the chunks running on that socket.
$maxWL_s$	stores the work-load of the critical-chunk.

Fig. 11. Variables used in Section 3.2 and further.

and 21. Task 3: This task emits the code at Lines 19 and 29 in Figure 6(b) to update the appropriate *WL* arrays.

3.2 `FreqScaler`-emitter

`FreqScaler`-emitter emits the code to update the frequency of the sockets (at specific intervals) based on the remaining work-load of the chunks assigned to the threads. Recall (Section 2) that the number of chunks created is equal to the number of threads, which is set equal to the total number of cores on the system. Hereafter, we use the phrase "*chunks running on a socket*" to refer to the chunks assigned to the threads running on the cores that are connected to that socket. We call the chunk executed by the critical-thread, a *critical-chunk* and the rest as *non-critical-chunks*. The main part of the code emitted by `FreqScaler`-emitter deals with the function `FreqScaler`. For the ease of reading, we list the variables introduced in this section in Figure 11.

The function `FreqScaler` is invoked in each chunk where it checks if it is the *leader*, and if so updates the frequencies of all the sockets (by calling the function `updateFreq` discussed later). A leader should satisfy one of the following two properties:

Property 1—The leader is a critical-chunk. Note: if two chunks with chunk-ids i and j have the same remaining work-load and $i > j$, then i is considered the chunk with larger remaining work-load.

Property 2—The leader is not the critical-chunk, but it was marked as the future leader by the previous leader. We use the variable *curLdr* (an atomic variable) to store the chunk-id of the leader. The *curLdr* can also store two special values: (i) INIT-LDR: indicates the absence of a leader at the start of an ACLoop. (ii) NO-LDR: indicates the leader has executed all its chLoop iterations.

The code to invoke `FreqScaler` is inserted before the first statement of the body of the chLoop under consideration. This call is guarded by a predicate that ensures that `FreqScaler` is invoked at the beginning of the chLoop, and after every *backoff* number of iterations. The variable *backoff* is initialized to *STEP* (a system-parameter, see Section 5 for a discussion on the value of *STEP*). For example, in Figure 6(b) we insert the following piece of code, after Lines 13 and 26:

```
if (((k-ii)%backoff) == 0) backoff = FreqScaler(ch, backoff, WL);
```

The function `FreqScaler` takes three arguments: (i) the current chunk-id `ch`, (ii) a thread local variable *backoff* that is used to determine the interval at which `FreqScaler` is invoked in that chunk, and (iii) the *WL* array of the corresponding ACLoop in execution.

In addition to the code emitted to invoke `FreqScaler`, the following piece of code is emitted as the last statement of the chunk (in Figure 6(b), after Lines 21 and 31):

```
done(ch) = true ; chunk ch has finished execution.
curLdr.CAS(ch ,NO-LDR); if ch is same as the curLdr, set curLdr to NO-LDR.
```

To avoid race condition, a key requirement of frequency updating (by calling the function `updateFreq`) is that it must happen inside a critical section (hereafter, referred to as CS). In

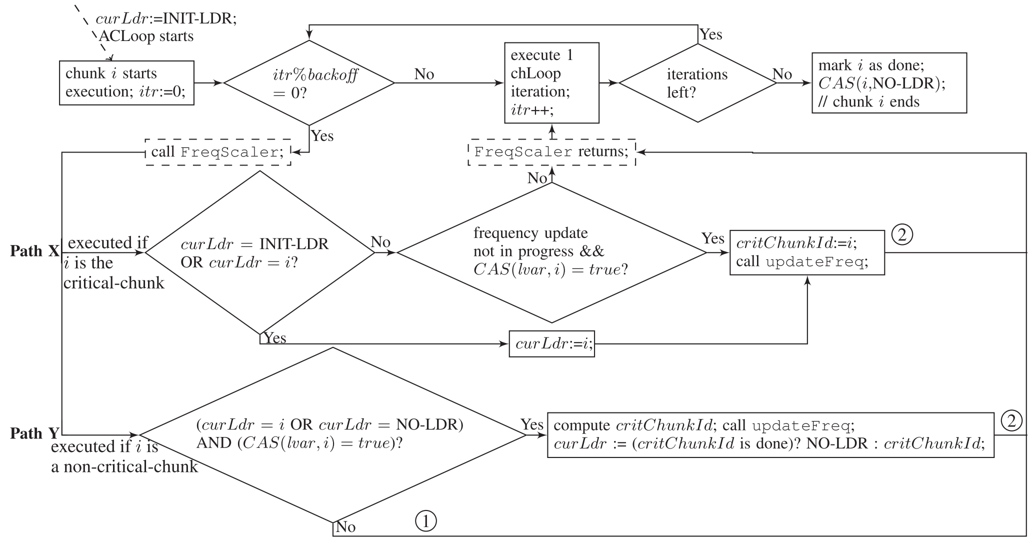

curLdr:=INIT-LDR;
ACLoop starts
chunk i starts execution; itr:=0;
itr%backoff = 0?
No
Yes
execute 1 chLoop iteration; itr++;
iterations left?
Yes
No
mark i as done; CAS(i,NO-LDR); // chunk i ends
call FreqScaler;
FreqScaler returns;
No
Path X
executed if i is the critical-chunk
curLdr = INIT-LDR OR curLdr = i?
No
frequency update not in progress && CAS(lvar, i) = true?
Yes
critChunkId:=i; call updateFreq;
2
Yes
curLdr:=i;
Path Y
executed if i is a non-critical-chunk
(curLdr = i OR curLdr = NO-LDR) AND (CAS(lvar, i) = true)?
Yes
compute critChunkId; call updateFreq; curLdr := (critChunkId is done)? NO-LDR : critChunkId;
2
No
1

Fig. 12. Flow chart of the execution of chunk *i* in the presence of the code emitted by FreqScaler. We use the abbreviation $CAS(x, y)$ to indicate $curLdr.CAS(x, y)$. Except *curLdr* and the array *done*, the rest of the variables are thread-local variables. Note: *lvar* is initialized to *curLdr*, before each use.

addition, we also want to avoid performing excessive CAS operations to minimize the associated overheads. We now describe FreqScaler that ensures that both of these requirements are met.

Algorithm intuition: After every *backoff* number of iterations, each chunk first checks if it is the critical-chunk. If so, then it marks itself as the current leader, updates the frequencies of the sockets provided no other chunk is doing the same, and then resumes the execution of its chLoop iterations. Each non-critical-chunk checks if it is the current leader (the first case), or no leader is selected yet (the second case). If the check succeeds, then it atomically marks itself as the current leader and updates the frequencies of the sockets, otherwise it resumes the execution of its chLoop iterations. Note that the atomic operation here resolves the possible contention either between the leader non-critical-chunk and the non-leader critical-chunk (for the first case), or among multiple non-critical-chunks (for the second case).

Figure 12 shows the flow chart of the chunk execution (with chunk id = *i*), in the presence of the code emitted by FreqScaler-emitter. Before the execution of any chunk, *curLdr* is set to INIT-LDR. Each chunk has a local variable *itr* that holds the number of chLoop iterations executed by the chunk. Starting from *itr*=0, and after every *backoff* iterations, the function FreqScaler is invoked. Once all the iterations of the chunk *i* have finished execution, a flag is set to indicate the completion of chunk *i*. Further, if *curLdr* is set to *i* then it is (atomically) set to NO-LDR. This helps choose a new leader (among the remaining chunks), if the current leader finishes early.

The function FreqScaler first checks if *i* is the critical-chunk (in Figure 12, critical-chunk takes the **path X**—Property 1 holds). If so, then it invokes the function updateFreq, provided one of the following conditions hold: (i) *curLdr* = INIT-LDR, (ii) *curLdr* = *i* (the critical-chunk), or (iii) if no other chunk is updating the frequency and chunk *i* succeeds in setting (atomically) *curLdr* to *i*. Note that if the condition (i) or (ii) is true, then there is no need of CAS to update *curLdr*, because it is guaranteed that there will be at most one chunk, which will find its remaining work-load to be the largest and satisfy (i) or (ii). However, if the condition (iii) is true, we need to update *curLdr* atomically, because one or more chunks (not having the largest remaining work-load) may also try to update *curLdr* (before calling updateFreq) under some conditions (see below).

If i is a non-critical-chunk (in Figure 12, non-critical-chunk takes the **path Y**), then it invokes the function `updateFreq`, if $curLdr = i$, or NO-LDR and chunk i succeeds in setting (atomically) $curLdr$ to i; these conditions enforce Property 2. Here, since i is a non-critical-chunk, we find the critical-chunk-id and set $curLdr$ to this id. We add an additional check to see if $curLdr$ has finished executing all the iterations of its chLoop in which case we set $curLdr$ to NO-LDR. This additional step ensures that we do not choose a leader that has finished execution. We skip the detailed `FreqScaler` algorithm for space and instead discuss some of the salient points of `FreqScaler`.

- We have observed that if chosen as the leader, the critical-chunk typically remains as the leader across multiple invocations of `FreqScaler`. To reduce the amount of overheads of `FreqScaler` in this common case, we first check if the critical-chunk is the current leader. If so, then we avoid performing the CAS operation and simply execute the CS. Note that no CAS operation is required in this case, because once a chunk i finds that it is the critical-chunk and it was earlier selected to be the leader, no other chunk can start executing CS until chunk i decreases its remaining work-load (done only after completing the execution of the corresponding instance of `FreqScaler` and one chLoop iteration).
- *Property 1* alone is not sufficient to ensure that the leader will always be selected. It may happen that every chunk created inside a particular ACLoop finds its remaining work-load to be lesser than that of some other chunk in that ACLoop, in which case the leader may not get selected at all. To avoid such a scenario, we also give a chance to the non-critical-chunks to become the leader. Note that there could be multiple such chunks, and we use CAS operations to select the leader. To reduce the number of CAS operations, we give an opportunity to a chunk i to be the leader, if either it is the current leader ($curLdr = i$) or there is no leader ($curLdr$=NO-LDR).
- *backoff* update: The value of the thread-local variable *backoff* indicates the number of chLoop iterations after which the function `FreqScaler` is re-invoked by a chunk. Calling `FreqScaler` too often (low *backoff* value) increases the overheads of executing the function; it becomes worse, if most of the invocations return, without changing the frequency. Similarly, calling `FreqScaler` very infrequently (high *backoff* value), may reduce the chances of exploiting the load-imbalance efficiently. As a trade-off, we increase the *backoff* value for the chunks that are less likely to call `updateFreq`. We use the following three intuitions, which are based on the temporal nature of the non-leaders and load-imbalance: (i) Non-critical and non-leader chunks continue to remain as non-critical and non-leader in the near future (double the *backoff* value, on the edge labeled ①, in Figure 12). (ii) If `updateFreq` does not change the frequency of any socket, then the load is balanced and will be in such a state in the near future (in such a case, double the *backoff* value, on the edge labeled ②, in Figure 12). (iii) If the frequencies have changed, then the load is not balanced and it may change again (in such a case, halve the *backoff* value, on the edge labeled ②, in Figure 12).
- The proposed design of `FreqScaler` induces a data race: while a particular chunk is reading $WL(j)$, the jth chunk may be updating the same element, in parallel. If the critical-chunk is chosen as the leader (common case), then the race has no impact on the frequency of the critical-socket, while the other sockets may be set to a sub-optimal frequency. Consequently, the race does not lead to any increase in execution time, though it may reduce some energy gains. If the non-critical-chunk is chosen as the leader (less-common case), then the race may lead the critical-socket to run at a sub-optimal frequency (till the critical-chunk is again chosen as the leader), thereby possibly increasing the execution time during that period.

Abbreviation	Definition	Abbreviation	Definition
`CpS`	number of cores per socket.	execTime	execution time of the critical-chunk.

Fig. 13. Abbreviations used in Section 3.3.

However, our experiments have shown that such scenarios were very infrequent and the impact of the `FreqScaler` on the execution time was minimal.

Note that the data races have undefined behavior in some languages (for example, Java). To use our technique in such a language without impacting the execution time negatively (note—the transformation does not alter the input program semantics), we need a guarantee from the hardware that the integer writes are atomic in nature and the obtained (read) value of the workload (an integer value) is either an old value or the current value (no "out-of-thin-air" values); such guarantees are typical of the ones provided by popular architectures like those from Intel and AMD.

- If-else-statement: `CompWL` adds the maximum of the work-loads of the if-block (if-path) and the else-block (else-path) to the total work-load of the chunk (see Section 3.1). We call the path whose work-load is maximum as the worst-case path. At runtime, the chunk execution may deviate from the worst-case path, which in turn causes the average remaining work-load to be lesser than the worst-case remaining work-load. For example, on an average, across all kernels discussed in Section 6, the value of average-case remaining work-load is less than worst-case remaining work-load by 14% (not too high) in each chLoop iteration. An important point to note is that when the deviation of average remaining work-load from the worst-case remaining work-load is large for a particular chunk, its remaining work-load decreases by a larger amount. Consequently, the socket running such a chunk is tuned at a lesser frequency compared to other sockets, which is logical.

Function `updateFreq`: This function is invoked by the leader from the function `FreqScaler`. The function `updateFreq` takes two arguments: *critChunkId* (the chunk-id of the critical-chunk) and *WL* (discussed above). As discussed in Section 1, the frequency of a MSMC system can only be changed at a per-socket granularity. The leader first finds the chunk with the maximum remaining work-load from each socket (we call it the socketRWL of that socket). After this, the frequency of each socket is set such that the time taken by any of its chunks does not exceed that of the critical-chunk. We set the new frequency F_i for the socket i as the nearest system-frequency $\geq$ $(socketRWL_i \div maxWL_s) * \text{MAXFREQ}$, where $maxWL_s$ is the remaining work-load of the critical-chunk. To avoid redundant frequency updates for a socket, we update the frequency of any socket, only if its current frequency does not match the new frequency. Note: (i) The critical-socket always runs at MAXFREQ. (ii) We select the largest remaining work-load of the chunks running on each socket as the socketRWL of that socket, because it ensures that the calculated frequency of the socket is not under-estimated (otherwise, it may increase the execution time). We skip the detailed algorithm for space.

3.3 `MigrationController`-**emitter**

The scheme proposed in Section 3.2 exploits the imbalance among the socketRWLs of the available sockets. For a given system with n sockets, the total %load imbalance can be computed as $(\Sigma_{i=0}^{n}(maxWL_s - socketRWL_i) \div execTime) \times 100$; Figure 13 lists the variables introduced in this section. The imbalance can be maximized by sorting the threads based on the execution times of the chunks assigned to them and mapping a thread at index i in the sorted order to the socket $\lfloor i \div \texttt{CpS} \rfloor$. More imbalance in general can lead to more energy saving. Figure 16 (columns 8 and

```
Function TMController(int $backoff$, int $WL[]$)
    // $backoff$ the frequency update interval
        is passed by reference.
    $sortedThLst$ := sort $thData$ by the workloads
        of their chunks;
    $counter := 0$;
    foreach $i \in sortedThLst$ do
        $curSockNum := i.coreId / \texttt{CpS}$; /* integer division */
        $newSockNum := counter / \texttt{CpS}$; /* integer division */
        if $(curSockNum \neq newSockNum)$ then
            $sInfo(curSockNum).freeC.add(i.coreId)$;
            $sInfo(newSockNum).newT.add(i)$;
        $counter := counter + 1$;
    boolean $migrationNeeded := false$
    foreach $k \in sInfo$ do
        while $\neg k.newT.empty()$ do
            $i := k.newT.remove()$;
            $migrationNeeded := true$;
            $i.newCoreId := k.freeC.remove()$;
            $i.migrate := true$;
    if $migrationNeeded = false$ then $backoff := backoff \times 2$ ;
    else $backoff := backoff \div 2$;
```

```
// Global arrays used by Figure 14
sockInfo sInfo[]; // for each socket
TData thData[];   // for each thread
```

Fig. 14. The function TMController.

9) lists the imbalance and the maximum possible imbalance for the IMSuite kernels ran on a two socket Intel system. As it is visible, there is significant gap between the two columns in many of the kernels (for example, DS, DST, and MIS). Naturally, thread-migration can be used to increase the load-imbalance.

MigrationController-emitter emits the code to perform thread migration that may increase the chances to scale down the frequencies of the sockets. This code consists of three parts.

Part 1: A call to the function TMController (algorithm shown in Figure 14) is emitted in the function FreqScaler after the call to function updateFreq. TMController is called only when updateFreq does not modify the frequency of any socket. The function TMController (executed, if at all, only by the leader) uses some heuristics (listed below) to mark the threads that need to be migrated. The actual migration is performed by the individual marked threads.

To ensure that there is no increase in execution time, the frequency of any socket should be set such that the execution times of the chunks running on the socket do not exceed that of the critical-chunk—we call it the *critical* condition. Since the critical-socket is set to MAXFREQ, the maximum energy gains can be obtained by setting the non-critical-sockets' frequencies to the lowest frequency satisfying the critical condition. We first sort all the threads by the remaining work-loads of their assigned chunks and then migrate the threads to sockets using a block distribution: thread with index *i* in the sorted array is migrated to a core in the socket $\lfloor i \div \texttt{CpS} \rfloor$. To avoid redundant migrations, a thread is marked for migration only if its current socket is different from the newly identified socket.

To assist in the process of thread migration, we maintain two arrays *sInfo* (of type sockInfo) and *thData* (of type TData). The class sockInfo includes two fields: *freeC* and *newT* both of type ArrayList. The list *freeC* contains the list of cores of the socket that have no threads mapped to them. The list *newT* contains the list of threads that need to be migrated to one of the cores of the socket. Similarly, we maintain an object of class TData, for each thread, with fields *coreId* (gives the id of the core to which the thread is mapped), *migrate* (indicates if the thread needs to be migrated), *chunkId* (gives the id of the chunk executed by the thread), and *newCoreId* (gives the id of the core to which the thread is to be migrated).

TMController (Figure 14) performs the following four tasks (i) Line 3: It sorts the threads according to the remaining work-load of the chunks executed by them. (ii) Lines 5–11: It iterates over *sortedThLst* and for each thread i in this list, it checks if the current socket number (*curSockNum*) does not match the proposed socket number (*newSockNum*). If so, then it adds i to the *newT* list of the socket *newSockNum* and adds the id of the core in which the thread i is currently running (*i.coreId*) to the *freeC* list of the socket *curSockNum*. (iii) Lines 13–18: For each socket k, it selects the new thread i and a core-id from *freeC* and assigns it to *i.newCoreId*. It also sets the *migrate* flag for i (iv) Lines 19–20: If no thread is marked for migration, then it increases *backoff* (based on the heuristic assumption that the remaining work-load is balanced among the threads, since there is no migration). Otherwise, it decreases *backoff*.

Note: The order of sorting (increasing or decreasing) impacts the number of threads marked for migration. Hence, TMController chooses a sorting order that leads to lesser number of threads being marked for migration (code not shown explicitly in Figure 14).

Part 2: To ensure that the updateFreq (as discussed in the previous Section) sees a consistent picture of the thread-to-core mapping, we emit a guard code that ensures that the leader calls updateFreq only after all the threads marked for migration have been migrated (or have terminated); this is done by iterating over the *migrate* flag of all the threads (details omitted for brevity).

Part 3: The following piece of code is emitted as the first statement of each chLoop body (for example, after Lines 13 and 26 in Figure 6(b)). This code is executed by each thread to migrate itself, if it is marked for migration:

```
if(thData(threadId).migrate == true){ /* marked for migration */
   thData(threadId).coreId = thData(threadId).newCoreId;
   // migrate threadId to thData(threadId).coreId -- code not shown
   thData(threadId).migrate = false; }
```

4 HANDLING ATOMICS

The X10 atomic construct brings in new challenges to the frequency-scaling schemes proposed in Section 3. As discussed in Section 1, in each ACLoop containing atomic-blocks, scaling down the frequency of a non-critical-socket can slow down the critical-thread (even though the critical-socket is running at MAXFREQ). This is because running the non-critical-sockets at lesser frequencies than MAXFREQ causes the atomic-blocks executed on these sockets to run slow. If the critical-thread is waiting to execute any such atomic-blocks, then it has to wait longer, thereby increasing the execution time of the critical-thread, which in turn increases the execution time of the program. We now propose a scheme that tries to limit this increase in execution time. Our intuition is to first calculate the frequencies of the non-critical-sockets using the techniques discussed in Section 3 and then use a user-specified parameter ($wtTmP$) to further fine-tune the frequencies of the non-critical-sockets so the percentage increase in the execution time of the critical-thread is bounded by $wtTmP$, while maximizing the energy gains.

Let $maxWL_s$ be the remaining work-load of the critical-thread, in a stand-alone manner (ignoring the effects of inter-thread atomic-blocks); Figure 15 lists the variables used in this section and their brief descriptions. The remaining execution time of the critical-thread (at MAXFREQ) is given by

$$execTm_{s+mf} = maxWL_s \div \text{MAXFREQ}. \tag{1}$$

Let $\mathbb{T}$ be the set of threads (equal to the number of chunks created inside the ACLoop, see Section 2). For each chunk i, aWL_i be the sum of remaining work-loads of the atomic-blocks

Variable	Brief description
$maxWL_s$	Remaining work-load of the (standalone) critical-thread
$execTm_{mf}$	Remaining execution time of the critical-thread at MAXFREQ.
$execTm_{s+mf}$	Remaining execution time of the (standalone) critical-thread at MAXFREQ
aWL_i	Sum of the remaining work-loads of the atomic-blocks in chunk i.
$atWtTm_{mf}$	Worst-case waiting time (at atomic-blocks) of the critical-thread when all the sockets are running at MAXFREQ
$atWtTm_{varfreq}$	Worst-case waiting time (at atomic-blocks) of the critical-thread when not all non-critical-sockets are running at MAXFREQ
$diffP$	Percentage degradation in execution times due to frequency-scaling.
$wtTmP$	Max permitted % degradation in waiting time of the critical-thread (user specified parameter).

Fig. 15. Variables used in Section 4 along with brief descriptions. The variables referring to the metrics of the standalone threads are subscripted with s. The variables referring to the time metrics of the threads at MAXFREQ are subscripted with mf.

contained in the chunk; the remaining work-load of the atomic-blocks are obtained in a way similar to that of a chunk (Figure 7). In the worst case, the critical-thread has to wait for all the other threads to exit their atomic-blocks before it starts executing its own. Hence, if all the threads are running at MAXFREQ, the worst-case waiting time of the critical-thread (with id *critT*) due to the atomic-blocks (hereafter, referred to as atWtTm) is given by

$$atWtTm_{mf} = (\Sigma_{i=0,\, i \neq critT}^{|\mathbb{T}|} aWL_i) \div \text{MAXFREQ}. \tag{2}$$

The worst-case execution time of the critical-thread would be the sum of its waiting time at the atomic-blocks and the time to execute its remaining work-load. Hence, when all the sockets are running at MAXFREQ, the worst-case execution time of the critical-thread is given by

$$execTm_{mf} = execTm_{s+mf} + atWtTm_{mf}. \tag{3}$$

Recall that the procedure discussed in Section 3 changes the frequencies of the sockets during the execution of the ACLoop. For a thread $i \in \mathbb{T}$, say F_i is the computed "new" frequency of the socket containing the core to which i is mapped. The atWtTm because of the frequency change is given by

$$atWtTm_{varfreq} = \Sigma_{i=0,\, i \neq critT}^{|\mathbb{T}|} (aWL_i \div F_i). \tag{4}$$

Note that since $F_i < \text{MAXFREQ}$, $atWtTm_{varfreq} > atWtTm_{mf}$. This increase in the waiting time of the critical-thread after frequency-scaling causes increase in the execution time of the program. The percentage difference in the execution time due to the frequency-scaling is given by

$$diffP = 100 \times (atWtTm_{varfreq} - atWtTm_{mf}) \div execTm_{mf}. \tag{5}$$

As it can be seen, when the frequency of any non-critical-socket is reduced, the value of *diffP* increases. Rewriting Equation (5):

$$atWtTm_{varfreq} = atWtTm_{mf} + (diffP \times execTm_{mf}) \div 100. \tag{6}$$

Using Equations (4) and (6):

$$\Sigma_{i=0,\, i \neq critT}^{|\mathbb{T}|} (aWL_i \div F_i) = atWtTm_{mf} + (diffP \times execTm_{mf}) \div 100. \tag{7}$$

Here, $atWtTm_{mf}$, $aWL_i(\forall i)$, and $execTm_{mf}$ are runtime constants, for each invocation of `FreqScaler`. Thus, varying values of F_i, varies *diffP*, and vice versa.

We now present an overview of our scheme that uses Equation (7) to bound the possible increase in execution time of the critical thread in the presence of atomic-blocks. We first use the scheme in Section 3.2 to compute a set of new frequencies; say, the computed frequency of the socket containing core to which thread i is mapped is given by F_i. We check if $\mathit{diffP} \leq wtTmP$ (a user specified parameter). If so, then the calculated increase in waiting time of the critical-thread is already bounded by $wtTmP$ and we use the computed frequencies without any further change. Otherwise, we obtain an updated frequency $\overline{F}_i$, while ensuring that $\overline{F}_i \geq F_i$ and $\mathit{diffP} \leq wtTmP$. Given the set of F_i values, we can potentially have many different sets of values of $\overline{F}_i$ that satisfy the above constraint. For simplicity, we compute (and use) a single constant $scale(\geq 1)$, across all the non-critical-sockets, and use the conservative solution $\forall i$, $\overline{F}_i = scale \times F_i$. Let us assume that the function $critSock(x)$ returns true if the socket containing the core in which thread x is mapped is critical. Rewriting Equation (7), we get

$$\frac{\Sigma_{i=0,\, i \neq critT,\, critSock(i)}^{|\mathbb{T}|} aWL_i}{\text{MAXFREQ}} + \frac{\Sigma_{i=0,\, \neg critSock(i)}^{|\mathbb{T}|} aWL_i}{(scale \times F_i)} \leq atWtTm_{mf} + \frac{wtTmP \times execTm_{mf}}{100} \tag{8}$$

$$\Rightarrow scale \geq \frac{\Sigma_{i=0,\, \neg critSock(i)}^{|\mathbb{T}|} (aWL_i \div F_i)}{atWtTm_{mf} + \frac{wtTmP \times execTm_{mf}}{100} - \Sigma_{i=0,\, i \neq critT,\, critSock(i)}^{|\mathbb{T}|} aWL_i \div \text{MAXFREQ}}. \tag{9}$$

We set $scale$ to the least value such that the percentage increase in execution time of the critical-thread does not increase beyond $wtTmP$:

$$scale = \frac{\Sigma_{i=0,\, \neg critSock(i)}^{|\mathbb{T}|} (aWL_i \div F_i)}{atWtTm_{mf} + \frac{wtTmP \times execTm_{mf}}{100} - \Sigma_{i=0,\, i \neq critT,\, critSock(i)}^{|\mathbb{T}|} aWL_i \div \text{MAXFREQ}}. \tag{10}$$

Note that if we use higher values for $scale$, then even though the execution time of the critical-thread does not increase beyond $wtTmP$%, the non-critical threads will run at higher frequencies (of course, limited by MAXFREQ), thereby reducing the energy gains. In our proposed approach, we use Equation (10) to compute the value of $scale$ and use it to re-calibrate the frequencies computed using the techniques described in Section 3.

We now explain the `atomicsHandler`-emitter (invoked after the `MigrationController`-emitter, see Figure 4(a)) to translate ACLoops containing atomic-blocks. It replaces the call to `FreqScaler` (inserted by `FreqScaler`-emitter) by a variant of `FreqScaler` that takes an additional argument aWL (see the definition earlier in this section). The modified `FreqScaler` follows the same flow-chart as shown in Figure 12, except that it invokes `updateFreqAtomics` (that takes an additional argument of aWL) instead of `updateFreq`.

For each socket k, the function `updateFreqAtomics` (detailed algorithm skipped for brevity) first stores the frequency (calculated using the schemes discussed in Section 3) in the array $freqs$. It then uses Equations (2) and (4) to compute $atWtTm_{mf}$ and $atWtTm_{varfreq}$. After this, it computes $execTm_{mf}$ and $scale$ using Equations (3) and (10), respectively. For each socket k, it multiplies $scale$ to $freqs(k)$ to get the scaled frequency of k. Similar to the function `updateFreq`, it then updates the frequencies of the sockets.

Similar to the code emitted by Task 3 of `CompWL`-emitter (Section 3.1), for each chunk i that contains atomic-blocks, we emit code to decrement the remaining work-load of atomic-blocks (stored in aWL) after the last statement of chLoop.

5 DISCUSSION

Computing the work-load: Wilhelm et al. (2008) present a survey of various works that statically compute the worst-case execution time of programs. All these works rely on some form of user-annotations (or static bounds) to get the loop-bounds. In case of ITP programs, the loop-bounds are input-dependent in most of the cases, thereby making it hard to provide them as user-annotations.

To overcome such a limitation, our scheme emits code that (at runtime) calculates the loop-bounds and consequently the work-load of the chunk. To handle codes (blocks) with input-independent (statically known) work-loads, we use a heuristic similar to that of Thoman et al. (2012): we count the number of assembly instructions corresponding to those blocks of codes to obtain a fair estimate of the work-load of the block. Conservatively assuming that each instruction takes one cycle to finish, we can divide the computed work-load by the current operating frequency of the core to get an estimate of the execution time. While more precise ways of estimating the work-loads of the input-independent blocks can be used (for example, schemes that consider the individual instruction latencies, and so on), our evaluation shows that our simple heuristic gives encouraging results. Further, our cost calculations also take into consideration the cost of the instrumentation code. In Section 6, we show that overheads due to the instrumentation code is minimal.

Limiting the overheads of X10Ergy: If a given ACLoop has no load-imbalance among its chunks, then invoking X10Ergy does not lead to much additional gains. Similarly, in the presence of deep nesting of loops, since the work-load is estimated at execution time, the resultant work-load expression returned by `getWL` can become very expensive to execute. To limit such overheads, we invoke X10Ergy only on those ACLoops that satisfy the following conditions: (i) has for-loops (with input dependent loop-bounds) inside the chLoop, (ii) does not have nested loops (with nesting depth > 3) and whose loop-index variables are dependent on each other. Moreover, the execution of the program is also affected by (a) the total number of instrumented instructions that are added by X10Ergy, and (b) the total number of times the sorting of remaining work-loads is done to perform thread migration (see Section 3.3). We have experimented with the kernels discussed in Section 6, and found that on an average, across all kernels, (a) the number of additional instructions executed because of X10Ergy is $\approx 0.5\%$ of the total instructions executed in the program, which is minimal, and (b) we performed sorting after every 2919 chLoop iterations, which is not too frequent.

Value of the parameters *STEP* and *wtTmP*: Experimentally (details in Section 6), we found that setting $STEP = 16$ (the initial value of *backoff*, see Section 3.2) and $wtTmP = 3$ (giving a percentage bound on the waiting time of the critical thread) gave the best results (max energy gains for the least increase in execution time).

Cache modeling: Although we do not model cache explicitly, our proposed algorithm takes care of it implicitly by maintaining the remaining work-load of each chunk. Assuming all other factors remain the same, if two chunks mainly differ in their cache-hits (say, T1 has more cache-hits than T2), then T1 will decrement its remaining work-load faster than that of T2. Consequently, the corresponding socket of T1 will be made to run at lower frequencies, in the future iterations of chLoop. Or in other words, the computation of the remaining work-loads of different chunks at different points of execution takes into consideration (indirectly) the impact of cache memory.

Loop Scheduling: We show our optimizations in the presence of block-chunking, although it can also be applied in the context of many other scheduling policies (Kennedy and Allen 2002). However, X10Ergy optimizations are not suitable for dynamic scheduling policies (for example, dynamic-scheduling (Kruskal and Weiss 1985), guided scheduling (Polychronopoulos and Kuck 1987)), where the iterations assigned to chunks are not decided before the beginning of the ACLoop. We believe it to be an acceptable limitation, because as claimed by Zhu et al. (2006), Cai et al. (2008), and Rakvic et al. (2010), such dynamic techniques are not preferred (due to the incurred runtime overheads), especially in the context of ITP programs.

Runtime scheduling: Using the typical OS schedulers, the OS enters the power-saving states only when the hardware is completely unused by applications (Kambadur and Kim 2016), and hence the resulting energy savings are limited. In general, in the absence of any compiler analysis, a runtime scheduler cannot take advantage of the underlying behavior of the program (such as remaining work-load). As a result, the decisions taken by such a scheduler can at best be based on the past behavior of the program—can be ineffective, especially in the context of ITP programs.

X10Ergy vs work-stealing and task-resizing: In general, work stealing reduces the energy consumption as a side-effect of the decreased execution time. However, work-stealing in itself is not sufficient for the kernels we have under consideration, because for large inputs, most of them terminate with errors like “heap overflow.” Further, for chunked-loops, work-stealing cannot decrease the execution time nor energy any further, as the number of tasks created in a particular Async Creator Loop matches the number of cores (no tasks to steal). Similarly, in loop-chunked ITP programs traditional task-resizing techniques (Zahran and Franklin 2003) do not get much scope to resize and balance the work-load as #tasks = #cores. However, X10Ergy can be used to derive additional energy gains, after the completion of the load-balancing passes by applying frequency-scaling and thread-migration techniques discussed in this manuscript.

Interference of X10Ergy with other thread-migration techniques: It is natural to expect that if different migration techniques (for example, X10Ergy—to improve energy gains, the schemes of Brown et al. (2011) and Shim et al. (2014)—to improve cache-locality, the scheme of Salami et al. (2014)—to reduce chip temperature, and so on) are deployed together, the overall gains from any migration technique may get impacted (negatively) because of the interference from other migration techniques. It would be interesting to study the impact of such interfering migration techniques and devise new methodologies to reduce the negative impact thereof. Considering the complexities involved in such a study, we leave it as a future work.

Energy measurement: The Running Average Power Limit (RAPL) interface supported in the Intel E5 systems has a provision to get notifications on the energy consumption of the socket. We use RAPL to obtain the energy consumed by a socket by reading a register (called the MSR register), specific to that socket. We get the energy consumed by a program by computing the sum of the differences between the MSR values of each socket before and after the program execution.

Though the energy consumed by a socket increases monotonically, the width of the MSR register is fixed (32 bits). Once the value of the MSR reaches the highest supported limit, it gets wrapped around and again starts increasing from 0; in such a case the computed difference between the MSR values can be negative. In our evaluation, we identified and ignored such runs.

Generality of the proposed scheme: Though our proposed solution is presented for X10 programs, it can be extended to other task-parallel languages that support parallel-loops and atomic-blocks (for example, HJ (Cavé et al. 2011; OpenMP 2013; Chapel 2005), and so on). Our proposed solution is applicable to those hardwares that provide user-level interfaces to change the frequencies of the cores, or sockets (for example, Intel machines like Nehalem (Intel 2008), Sandy Bridge (Rotem et al. 2012), Ivy Bridge (Intel 2017), and AMD machines like Opteron (AMD 2016), and so on). Note that the current AMD machines do not provide any register that stores the energy consumption of the program (thus, we cannot directly measure the energy gains here), though we can apply our techniques to reduce the energy consumption. For these machines, we can use various energy profilers (Manousakis et al. 2015; Alonso et al. 2012) to measure the energy consumption.

6 IMPLEMENTATION AND EVALUATION

We have implemented our proposed scheme, along with those of Rakvic et al. (2010) and Chen et al. (2014), in the X10 compiler version 2.4. Our choice of these prior works for comparison is based on the fact that similar to X10Ergy both of these techniques use compile-time analysis to emit the code that at runtime decides on the quantum of frequency-scaling (based on prior execution histories in terms of number of iterations executed or the execution times). For each kernel, the execution time and the energy readings are reported as an average over 30 runs. Each kernel was compiled with the -NO_CHECKS flags of the X10 compiler that omits many typical checks such

kernel	Brief description	i/p size	SLC	DLC	SAC	DAC	LI	MLI
BF	Breadth-first-tree construction (BellmanFord)	65,536	2	8	1	2,097,129	12	12
BY	Byzantine consensus	1,024	3	5,289	1	204,203,388	32	44
DP	Leader election in a general network	65,536	4	116	1	3,221,193	77	80
DR	Dijkstra routing	1,024	1	1	0	0	0	5
DS	Dominating set	32,768	9	27	2	65,528	22	66
DST	Breadth-first-tree construction (Dijkstra)	32,768	7	84	3	182,413	40	96
HS	Leader election in a ring network (bidirectional)	16,384	5	98,313	0	0	8	19
KC	K-Committee construction	8,192	10	4,501	2	123,446,870	14	22
LCR	Leader election in a ring network (unidirectional)	16,384	3	32,769	0	0	6	17
MIS	Maximal independent set	32,768	5	5	3	153,692,873	11	31
MST	Minimum spanning tree	32,768	15	28	4	65,535	0	3
VC	Vertex Coloring	65,536	5	17	0	0	58	60

Fig. 16. Characteristics of Input kernels. We use the following abbreviations: **SLC**, static ACLoop count; **DLC**, dynamic ACLoop count; **SAC**, static atomic-block count; **DAC**, dynamic atomic-block count; **LI**, %load imbalance over the execution time; **MLI**, %maximum load imbalance over the execution time.

as NullPointerCheck, ArrayOutOfBoundsCheck, and so on; we call this the *Baseline* version. To invoke our proposed optimizations, and those of Rakvic et al. (2010) and Chen et al. (2014), an additional flag -X10Ergy, -MP-OPT, and -EEWA-OPT is passed to the X10 compiler, respectively. The experiments were conducted on a dual socket (8 cores per socket) Intel E5-2670 system. Each socket can be set to a frequency ranging from 1.2GHz to 2.6GHz; the operating frequencies are 1.2GHz, 1.3GHz, . . . , 2.6GHz.

We evaluate our proposed optimizations using the kernels from IMSuite (Gupta and Nandivada 2015). A short description of these kernels along with the chosen input sizes, static and dynamic counts of the ACLoops, and the static and dynamic counts of the atomic-blocks are given in Figure 16. The input is chosen as the maximum sized (in powers of 2) graph such that (i) the overall execution time of the complete program (including the time to read the input from the files, initialization and the actual computation) does not exceed two hours, and, (ii) across multiple invocations, for at most 10% of the runs of the actual computation code, the difference between the values of the MSR register (before and after the computation code) is negative. This ensures that we do not have to ignore too many runs, because of the MSR register overflow (see Section 5).

Of the 12 kernels in IMSuite, 8 have visible load-imbalance (max-load-imbalance (MLI) ≈ 20% or more): BY, DP, DS, DST, HS, KC, MIS, and VC. In Section 6.1, we discuss the gains resulting from X10Ergy for kernels that exhibit load-imbalance. Later (in Section 6.2), we show that X10Ergy does not lead to much overheads even in kernels that do not have much load-imbalance.

6.1 Impact of X10Ergy on Kernels with Load-imbalance

X10Ergy vs. Baseline: Figure 17 shows the normalized execution time and the energy consumption of the kernels optimized using X10Ergy, with respect to those resulting from the Baseline optimizations. As it can be seen, the X10Ergy versions save a significant amount of energy (on average 15% less) compared to the Baseline versions, while incurring negligible performance degradation (on average 2%). This is because our proposed techniques exploit the load imbalance in the application and appropriately scale down the frequencies of the non-critical-sockets (leads to energy gains), while ensuring that the critical-socket run at MAXFREQ (limits the increase in execution time).

For the HS kernel, the load-imbalance present in the input program is low (column 8, Figure 16), and even after thread-migration, the maximum possible load imbalance is not too high (column 9, Figure 16). Consequently, the gains are also low (7%). For DS, KC, and MIS, the higher load-imbalance translates to improved energy gains (20%, 19%, and 11%, respectively). In case of BY, though the load imbalance is not too low, the energy gains are moderate. This is because, the load

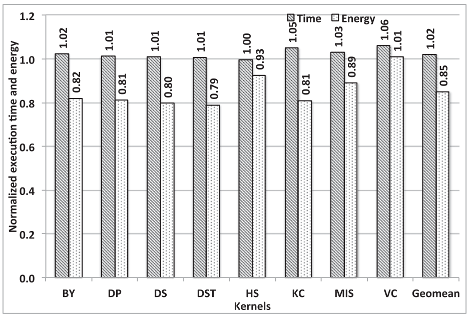


Fig. 17. Normalized execution time and energy consumption of X10Ergy versions against the Baseline versions at MAXFREQ.

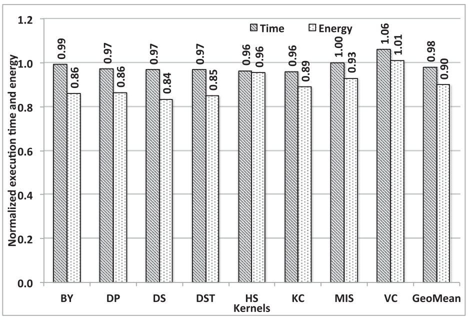


Fig. 18. Normalized execution time and energy consumption of X10Ergy versions against the Baseline versions at the threshold-frequency.

imbalance is spread across a large number of ACLoop instances, thereby reducing the scope for exploiting the load imbalance. Further, many of the ACLoops have no load-imbalance and thus give no scope to reduce energy. This in-turn reduces the value of the energy-gains metric that is computed by normalizing the energy-reduction against the total energy consumption across all the ACLoops.

DST shows an interesting case where the energy gains (21%) though are better than that of BY (note: the load-imbalance in the Baseline versions of DST is more than that of BY), the energy gains in DST are not quite matching the large maximum load-imbalance (96%) that can be realized by thread-migration (column 9, Figure 16). The main reason for such a behavior is that here, after thread-migration, the execution time difference among the critical-threads and the non-critical-threads is quite high. Consequently, the difference of energy consumption between the critical-socket with that of the non-critical-socket (after thread-migration) is also quite high. As a result even though there is a significant % reduction in energy in the non-critical-socket, we achieve low overall gains (computed as energy-gains/(total energy consumption across both critical and non-critical-sockets)).

In case of DP, X10Ergy leads to 19% gains. Here, there are some ACLoops with large load imbalance (similar to DS)—leading to good energy gains. These gains get slightly overshadowed because of the presence of many ACLoops where we don't get much energy gains (either because they have no imbalance or very large imbalance).

In case of VC, there is an increase in both the execution time (6%) and energy consumption (1%). This is because the execution time of the ACLoops is very low (≈10 ms) and the overhead of adding the instrumentation code outweighed the energy gains.

Energy gains: The overall energy gains depend on the amount of load-imbalance. Since the overall gains is calculated as 100×(energy-gains)/(total-energy-consumed), the presence of parts of code in the input program that do not lead to much energy gains (either because of low or very high load-imbalance), reduce the overall-gains. *Possible deterioration*: Our proposed techniques are more suited for the cases where the execution time of the ACLoop is significant (order of ≈100 ms), so the overhead of executing the instrumentation code is amortized by the ACLoop execution time.

X10Ergy vs. Baseline at threshold-frequency: As shown in Figure 17, X10Ergy results in significant energy gains while incurring marginal execution time increase. To invalidate the claim that such energy gains (without increasing execution time) could also be achieved by simply running the sockets at a lower frequency, we ran each of the Baseline kernels at a frequency (< MAXFREQ), such that its execution time is just more than that of its X10Ergy counterpart; we

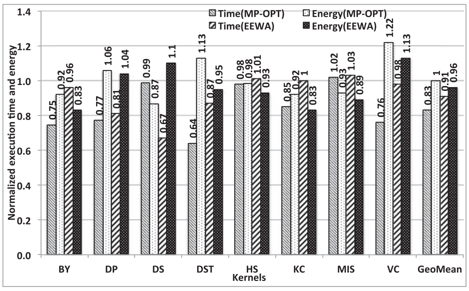


Fig. 19. Normalized execution time and energy consumption of X10Ergy versions against the MP-OPT and EEWA-OPT versions.

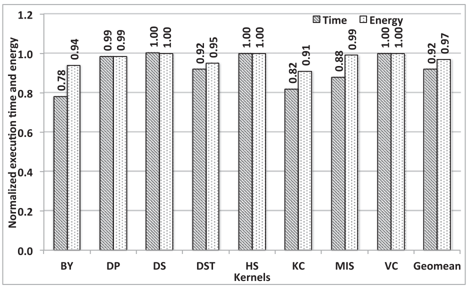


Fig. 20. Normalized execution time and energy consumption of X10Ergy versions against the X10Ergy-NA versions.

call it the threshold-frequency. Figure 18 shows the normalized execution time and the energy consumption of the X10Ergy versions, with respect to the Baseline versions executing at their respective threshold-frequencies. As it can be seen, though the Baseline versions running at the threshold-frequencies take more time than the X10Ergy versions, the average energy consumption is still higher than that of X10Ergy (except for VC, see above for a discussion about the behaviour of VC). This indicates that the energy gains realized by the X10Ergy cannot be achieved by the Baseline versions, by simply reducing the frequency of the sockets (without significantly impacting the execution time).

X10Ergy vs. MP-OPT and EEWA-OPT: Figure 19 shows the execution times and the energy consumption of X10Ergy normalized against the MP-OPT and EEWA-OPT. As it can be seen, on average, X10Ergy optimized codes consume almost as much energy as MP-OPT, but at a much lower execution time (≈17% less). For kernels DS, HS, and MIS, since MP-OPT performs no thread migration, the opportunities for energy savings diminish. In contrast, X10Ergy is able to exploit the load-imbalance to further reduce the energy consumption (≈13%, 2%, and 7%, respectively). For kernels BY and KC , MP-OPT optimized code takes more time to execute (compared to X10Ergy ≈25% and 15%) as it is oblivious to the presence of atomic-blocks in ACLoops. On the other hand, X10Ergy reduces the energy consumption with a minimal increase in execution time.

For kernels DP, DST, and VC, we see that X10Ergy optimized code consumes more energy (compared to MP-OPT ≈ 6%, 13%, and 22%, respectively), but takes significantly less time (compared to MP-OPT ≈ 23%, 36%, and 24%, respectively). This is because in MP-OPT, a thread does not change the frequency of the other threads. Hence, if some non-critical-thread t is later identified as critical then t may still run at a lower frequency until it executes a fixed number of iterations (leading to execution time increase). On the other hand, in X10Ergy, the leader changes the frequency of all the sockets ensuring that the critical-thread starts running at MAXFREQ as soon as it is identified.

Compared to EEWA-OPT, on average, X10Ergy is able to realize slightly more energy-gains (4%) at a much lower execution time (9% less). This is because EEWA-OPT assumes that the work-loads of the chunks present inside an ACLoop remain same across different invocations of that ACLoop. So, the frequency estimation of the sockets for the subsequent invocations of any ACLoop N is only based on the work-loads collected for the first invocation of N, and hence we see significant increase in execution time for the EEWA optimized kernels without much gains in energy (except VC where the work-loads are similar across different invocation of the ACLoop).

Compared to the Baseline (numbers not explicitly shown), on average MP-OPT and EEWA-OPT reduced the energy consumption by 14% and 11%, respectively, and in the process increased the execution time by 22% and 13%, respectively. In contrast, compared to the Baseline, X10Ergy

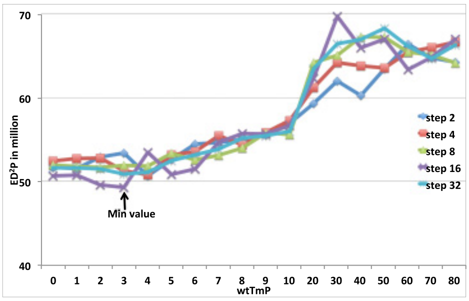


Fig. 21. ED^2P averaged over BY, DST, KC, and MIS. The plots are not continuous; they are connected for improving the readability.

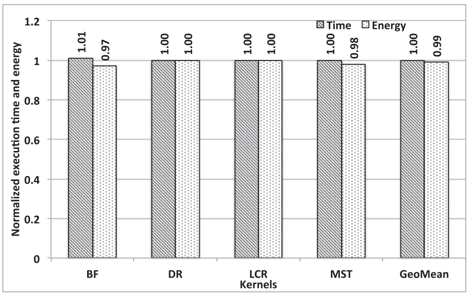


Fig. 22. Normalized execution time and energy consumption of X10Ergy versions against the Baseline balanced versions.

reduced the energy consumption by 15%, but at a minimal increase in the execution time (2%). This is in sync with our motivation to reduce the energy consumption with minimal impact in execution time.

Impact of handling atomic-blocks in X10Ergy: We compared X10Ergy with a scheme that is oblivious to the presence of atomic-blocks; we term it as X10Ergy-NA. Figure 20 compares the normalized execution time and energy consumption of the X10Ergy optimized kernels, with respect to the X10Ergy-NA optimized kernels. As it can be seen, compared to X10Ergy-NA, X10Ergy reduces the energy consumptuon (≈3% less), and decreases the execution time significantly (9%). This is because in the X10Ergy-NA versions, after scaling down the frequencies of the non-critical-sockets, the atWtTm (see Section 4) increases, thereby increasing the execution time of the ACLoops containing atomic-blocks. X10Ergy versions on the other hand, enforce an upper-bound (given by the parameter *wtTmP*) on the increase in atWtTm ensuring that the execution time of the ACLoops containing atomic-blocks is bounded.

Impact of values of *STEP* and *wtTmP*: We now present our experiment to tune the two parameters *STEP* and *wtTmP* so the energy reduction is maximized while minimizing the increase in execution time of the program. To study the impact of these parameters on the execution time and energy consumption of X10Ergy optimized kernels, we experimented with the four kernels of IMSuite (BY, DST, KC, and MIS) that contain a significant number of dynamic atomic-blocks and have significant load imbalance (≈20% or more). We varied *STEP* from 2 to 32 (in powers of 2) and *wtTmP* from 0, 1, …, 10, and 20, 30, …, 80. For each pair of *STEP* and *wtTmP*, we calculated the weighted Energy-Delay Product (ED^2P). For two ED^2P values a and b, if $a < b$, then a is said to have better tradeoff of energy consumption (lower importance) and execution time (higher importance) than b (Laros III et al. 2013). This metric is aligned with the goals of the article: energy reduction with minimal impact on the execution time. Figure 21 shows the geometric mean ED^2P values (over the four discussed kernels) for different pairs of *STEP* and *wtTmP*. We find that the best value of ED^2P is obtained for $STEP = 16$ and $wtTmP = 3$.

Overall, one can also observe that for lower values (0 to 5%) of *wtTmP* the non-critical-sockets are tuned at lower frequencies and the decrease in the energy consumption of non-critical-sockets is more than the increase in waiting time of the critical-thread (caused by tuning the non-critical-socket at lower frequency), thereby obtaining lower ED^2P values. For higher values of *wtTmP* (from 6% to around 30%) the overall ED^2P values increase. This is because energy gains obtained by further lowering the frequency of the non-critical-sockets were lower than the increase in waiting time of the critical-thread, thereby leading to higher ED^2P values. We have observed that for larger values (>30%) of $wtTmP$, ED^{2}P does not change much. This is because here the non-critical-sockets are tuned to the least possible frequency and no further decrease in the frequency is possible for them.

6.2 Impact of X10Ergy on Kernels with Balanced Load

Figure 22 shows the normalized execution time and energy consumption of the X10Ergy versions with respect to the Baseline versions, for the kernels that exhibit very little load-imbalance. As can be seen in the figure, while the energy gains are minimal in these kernels (on average 1%), the execution time overheads are also insignificant.

7 RELATED WORK

Du Bois et al. (2013) propose a hardware-based (runtime) approach to identify critical-threads (based on the running time of the threads, number of threads waiting at the barrier, and so on) and use frequency-scaling to accelerate the critical-thread (the other threads run at lower frequencies), which in turn leads to energy reduction. Bhattacharjee and Martonosi (2009) propose an approach where cache misses decides the thread-criticality with more weightage given to the last level cache misses. The critical-threads thus identified are run at MAXFREQ and the non-critical-threads are frequency-scaled. Cai et al. (2011) target multi-core systems with SMTs and propose a technique (thread-shuffling) where threads with similar criticality are mapped to the same SMT core after a fixed number of execution cycles and then non-critical-cores are frequency-scaled. Our approach is different from all the above works as (i) ours is a software-based approach that aims at reducing the energy consumption of multi-socket-multi-core (MSMC) systems, (ii) we identify the critical-threads based on the remaining work-load (not on the past behavior), (iii) we perform thread-migration across sockets, (iv) importantly, we handle codes with atomic-blocks such that reducing the frequency of non-critical-threads has a minimal impact on the execution time of the critical-thread.

In contrast to the MP-OPT scheme (see Section 2) of Rakvic et al. (2010) (i) we aim at reducing the energy consumption of MSMC systems (DVFS can be applied only at a socket level granularity), (ii) we identify the critical-threads based on the remaining work-load (more intuitive than the number of parallel tasks executed so far), (iii) we perform thread-migration across sockets, (iv) importantly, we handle codes with atomic-blocks in such a way that reducing the frequency of non-critical-threads does not significantly impact the execution time of the critical-thread. Interestingly, besides the scheme to apply DVFS on the non-critical-threads, Rakvic et al. (2010) propose another orthogonal scheme, where they clock-gate the entire processor (running the critical-thread) during the phases of inactivity leading to decrease in energy. An interesting future work would be to combine X10Ergy with their scheme, especially for MSMC systems.

Similar to the feedback based scheme of Chen et al. (2014) (see Section 2), Liu et al. (2005) measure the thread-criticality by measuring the amount of time each thread waits at the barriers. The authors use this information to scale the frequency of the non-critical-threads before the invocation of the future instances of those barriers. Both of these works assume that the work-load of each iteration of a parallel-for-loop do not vary across all threads in the subsequent iterations. However, this assumption may not hold in case of ITP programs because of the irregular nature of the workloads. Further, both these techniques are oblivious to the restrictions imposed by MSMC systems on performing DVFS (that is, they stick to the default thread-to-core mapping, thereby not able to fully exploit the load imbalance present in the application).

Salami et al. (2014) propose a scheme to dynamically balance the heat dissipation among the cores by performing task migration based on the temperature threshold limit set for each core.

Rangan et al. (2009) propose a scheme where the frequencies of the cores are fixed with different values and the threads are moved between high- and low-frequency cores depending on the cache misses incurred by the threads. Rauber and Rünger (2015) propose an analytical model to measure the energy and then minimize the energy consumption for the task-parallel programs having regular work-loads. Ribic and Liu (2014) propose an energy-efficient runtime system to apply DVFS based on the task-queues occupancy (not the remaining work-load) of the threads.

Barik et al. (2016) propose a runtime technique that combines the runtime behavior of the program with the power characterization of the processor to partition the work among the CPU and GPUs with the aim of reducing the energy consumption. Jibaja et al. (2016) propose a technique that dynamically assigns priorities and schedules the threads (holding the contended locks) to "big" cores to speed up the performance and reducing the energy (in heterogeneous systems). Our work is different from these works as we use a compiler + runtime scheme to obtain maximum energy gains using DVFS for symmetric multi-core processors, without increasing the execution time.

Noureddine and Rajan (2015) propose a compiler technique that performs energy optimization of the design patterns by reducing the calls to object instantiations, function calls, and memory operations. Kambadur and Kim (2016) provide application-level knobs to the programmers, which is used to trade-off the precision of the output with the energy consumption given the energy budgets of the program. Sampson et al. (2011) use type qualifier annotations for the data types such that the values of "approximate"-annotated data types can be approximately computed (using low-power operations) and stored on low-power memory to decrease the energy consumption. There also have been prior works (Rangasamy and Srikant 2011; Rangasamy et al. 2008) that use petri-net based performance models in the compiler to set the frequency of the cores. Hsu and Kremer (2003) discuss a technique for serial programs that identifies the program regions where the CPU can be slowed down with negligible performance loss. These program regions are the point where the CPU is idle due to memory stalls. Jimborean et al. (2014) and Koukos et al. (2016) propose compile-time techniques in which the program is transformed into access-execute programs where an access phase is run at low frequency to reduce the energy consumption, and execute phase is run at high frequency with minimal cache misses. Both of these works differ in their approaches to transform different types of applications; the former optimizes the multi-threaded scientific applications containing affine codes while the latter optimizes the sequential general purpose applications. Ozturk et al. (2013) propose a compile-time technique to estimate the work-load of each thread and map the threads to the sockets according to the sorted workloads at compile time. Their work necessitates that the loop-bounds are statically known with minimal if-else statements. Our technique is different from the above proposed compiler techniques, since our technique considers the more general case where the loop bounds and the branch targets are not known at compile time. Also our technique uses the remaining work-load of the threads to identify the critical-threads. Further, we also consider the effect of the atomic-blocks on the execution time of the critical-threads. We also take into consideration the MSMC systems and perform required thread migration before performing DVFS.

8 CONCLUSION

Energy-efficient compilation is a critical problem for the multi-socket-multi-core (MSMC) systems. We propose a novel energy reduction scheme (X10Ergy) for irregular programs with task-parallel loops (ITP programs) with minimal impact on the execution time. X10Ergy scales the frequency of the sockets based on the remaining work-loads of the chunks as against the prior works that either use the extremely inaccurate (especially for ITP programs) static estimations or the estimates based on past behavior. To the best of our knowledge, this is the first work that reduces the energy consumption of the ITP programs with atomic-blocks (on MSMC systems), while ensuring that the impact on the execution time is minimal. X10Ergy uses a mixed compile-time + runtime scheme that performs compile-time analysis to compute the input-independent work-loads of the chunks created inside the ACLoops and emits the instrumentation code to (i) calculate the input-dependent work-loads, (ii) combine the two to obtain the initial work-load of the chunks, (iii) update the remaining work-loads of the chunks at regular intervals, (iv) perform thread-migrations across sockets based on the remaining work-loads, and (v) scale the frequency of the sockets based on the remaining work-loads. We show that for IMSuite X10 kernels, with reasonable load-imbalance (≈20% or more among the program threads), X10Ergy leads to (average) 15% energy gains, with 2% increase in execution time. For kernels with low load-imbalance, the impact of X10Ergy is less in terms of energy reduction (natural), but importantly with not much increase in the execution time.

Extending our proposed work to handle SMT-enabled MSMC systems that add another level of abstraction for thread migration is an interesting future work. Another interesting future work would be to extend X10Ergy to be aware of the accesses to memory and multi-level caches. Similarly, extending X10Ergy to handle nested asyncs is an interesting future work, where we can use MHP analysis (Sankar et al. 2016) to improve the precision of the estimated work-loads.

REFERENCES

P. Alonso, R. M. Badia, J. Labarta, M. Barreda, M. F. Dolz, R. Mayo, E. S. Quintana-Ortï, and R. Reyes. 2012. Tools for power-energy modelling and analysis of parallel scientific applications. In *Proceedings of ICPP*. 420–429. DOI: http://dx.doi.org/10.1109/ICPP.2012.57

AMD. 2016. AMD Opteron Wiki Page. Retrieved from https://en.wikipedia.org/wiki/Opteron.

Rajkishore Barik, Naila Farooqui, Brian T. Lewis, Chunling Hu, and Tatiana Shpeisman. 2016. A black-box approach to energy-aware scheduling on integrated CPU-GPU systems. In *Proceedings of CGO*. ACM, New York, NY, 70–81. DOI: http://dx.doi.org/10.1145/2854038.2854052

Abhishek Bhattacharjee and Margaret Martonosi. 2009. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of ISCA*. ACM, New York, NY, 290–301. DOI: http://dx.doi.org/10.1145/1555754.1555792

Jeffery A. Brown, Leo Porter, and Dean M. Tullsen. 2011. Fast thread migration via cache working set prediction. In *Proceedings of HPCA*. IEEE Computer Society, Washington, DC, 193–204. Retrieved from http://dl.acm.org/citation.cfm?id=2014698.2014857.

Qiong Cai, José González, Ryan Rakvic, Grigorios Magklis, Pedro Chaparro, and Antonio González. 2008. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In *Proceedings of PACT*. ACM, New York, NY, 240–249. DOI: http://dx.doi.org/10.1145/1454115.1454149

Q. Cai, J. Gonzlez, G. Magklis, P. Chaparro, and A. Gonzlez. 2011. Thread shuffling: Combining DVFS and thread migration to reduce energy consumptions for multi-core systems. In *Proceedings of ISLPED*. 379–384. DOI: http://dx.doi.org/10.1109/ISLPED.2011.5993670

Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-java: The new adventures of old X10. In *Proceedings of PPPJ*. ACM, New York, NY, 51–61. DOI: http://dx.doi.org/10.1145/2093157.2093165

Chapel. 2005. The Chapel Language Specification Version 0.4. Retrieved from http://chapel.cray.com/.

Q. Chen, L. Zheng, M. Guo, and Z. Huang. 2014. EEWA: Energy-efficient workload-aware task scheduling in multi-core architectures. In *Proceedings of IEEE IPDPSW*. 642–651. DOI: http://dx.doi.org/10.1109/IPDPSW.2014.75

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. DOI: http://dx.doi.org/10.1145/115372.115320

Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout. 2013. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of ISCA*. ACM, New York, NY, 511–522. DOI: http://dx.doi.org/10.1145/2485922.2485966

T. Gautier, F. Lementec, V. Faucher, and B. Raffin. 2013. X-kaapi: A multi paradigm runtime for multicore architectures. In *Proceedings of ICPP*. 728–735. DOI: http://dx.doi.org/10.1109/ICPP.2013.86

Suyash Gupta and V. Krishna Nandivada. 2015. IMSuite: A benchmark suite for simulating distributed algorithms. *J. Parallel Distrib. Comput.* 75 (2015), 1–19. DOI: http://dx.doi.org/10.1016/j.jpdc.2014.10.010

Chung-Hsing Hsu and Ulrich Kremer. 2003. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of PLDI*. ACM, New York, NY, 38–48. DOI: http://dx.doi.org/10.1145/781131.781137

Intel. 2008. First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem). Retrieved from http://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf.

Intel. 2017. Intel Ivy bridge Wiki Page. Retrieved from https://en.wikipedia.org/wiki/Ivy_Bridge_(microarchitecture).

Ivan Jibaja, Ting Cao, Stephen M. Blackburn, and Kathryn S. McKinley. 2016. Portable performance on asymmetric multicore processors. In *Proceedings of CGO*. ACM, New York, NY, 24–35. DOI: http://dx.doi.org/10.1145/2854038.2854047

Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, and Stefanos Kaxiras. 2014. Fix the code. Don't tweak the hardware: A new compiler approach to voltage-frequency scaling. In *Proceedings of CGO*. ACM, NY, Article 262, 11 pages. DOI: http://dx.doi.org/10.1145/2544137.2544161

Melanie Kambadur and Martha A. Kim. 2016. NRG-loops: Adjusting power from within applications. In *Proceedings of CGO*. ACM, New York, NY, 206–215. DOI: http://dx.doi.org/10.1145/2854038.2854045

K. Kennedy and J. R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Jonathan G. Koomey, Christian Belady, Michael Patterson, Anthony Santos, and Klaus-Dieter Lange. 2009. Assessing trends over time in performance, costs, and energy use for servers. *Analytics Press* (Aug. 2009).

Konstantinos Koukos, Per Ekemark, Georgios Zacharopoulos, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. 2016. Multiversioned decoupled access-execute: The key to energy-efficient compilation of general-purpose programs. In *Proceedings of CC*. ACM, New York, NY, 121–131. DOI: http://dx.doi.org/10.1145/2892208.2892209

C. Kruskal and A. Weiss. 1985. Allocating independent subtasks on parallel processors. *IEEE Trans. Software Eng.* 11, 10 (October 1985), 1001–1016.

James H. Laros III, Kevin Pedretti, Suzanne M. Kelly, Wei Shu, Kurt Ferreira, John Vandyke, and Courtenay Vaughan. 2013. *Energy Delay Product*. Springer, London, 51–55. DOI: http://dx.doi.org/10.1007/978-1-4471-4492-2_8

Jian Li, Jose F. Martinez, and Michael C. Huang. 2004. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *Proceedings of HPCA*. IEEE Computer Society, Washington, DC, 14–23. DOI: http://dx.doi.org/10.1109/HPCA.2004.10018

Chun Liu, Anand Sivasubramaniam, M. Kandemir, and M. J. Irwin. 2005. Exploiting barriers to optimize power consumption of CMPs. In *Proceedings of IEEE IPDPS*. 5a. DOI: http://dx.doi.org/10.1109/IPDPS.2005.211

Ioannis Manousakis, Foivos S. Zakkak, Polyvios Pratikakis, and Dimitrios S. Nikolopoulos. 2015. TProf: An energy profiler for task-parallel programs. *Sustain. Comput.: Informat. Syst.* 5 (2015), 1–13. DOI: http://dx.doi.org/10.1016/j.suscom.2014.07.004

Developer Manual. 2016. Intel 64 and IA-32 Architectures Software Developers Manual. Retrieved from http://www.intel.in/content/www/in/en/processors/architectures-software-developer-manuals.html.

Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., CA.

V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. 2013. A transformation framework for optimizing task-parallel programs. *ACM Trans. Program. Lang. Syst.* 35, 1, Article 3 (April 2013), 48 pages. DOI: http://dx.doi.org/10.1145/2450136.2450138

Adel Noureddine and Ajitha Rajan. 2015. Optimising energy consumption of design patterns. In *Proceedings of ICSE*. IEEE Press, Piscataway, NJ, 623–626.

OpenMP. 2013. OpenMP Application Program Interface Version 4.0. Retrieved from http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

Ozcan Ozturk, Mahmut Kandemir, and Guangyu Chen. 2013. Compiler-directed energy reduction using dynamic voltage scaling and voltage islands for embedded systems. *IEEE Trans. Comput.* 62, 2 (2013), 268–278. DOI: http://dx.doi.org/10.1109/TC.2011.229

C. D. Polychronopoulos and D. J. Kuck. 1987. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.* 36, 12 (Dec. 1987), 1425–1439. DOI: http://dx.doi.org/10.1109/TC.1987.5009495

R. Rakvic, Q. Cai, J. González, G. Magklis, P. Chaparro, and A. González. 2010. Thread-management techniques to maximize efficiency in multicore and simultaneous multithreaded microprocessors. *ACM Trans. Archit. Code Optim.* 7, 2, Article 9 (Oct. 2010), 25 pages. DOI: http://dx.doi.org/10.1145/1839667.1839671

Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. 2009. Thread motion: Fine-grained power management for multi-core systems. In *Proceedings of ISCA*. ACM, New York, NY, 302–313. DOI: http://dx.doi.org/10.1145/1555754.1555793

Arun Rangasamy, Rahul Nagpal, and Y. N. Srikant. 2008. Compiler-directed frequency and voltage scaling for a multiple clock domain microarchitecture. In *Proceedings of ICF*. ACM, New York, NY, 209–218. DOI: http://dx.doi.org/10.1145/1366230.1366267

Arun Rangasamy and Y. N. Srikant. 2011. Evaluation of dynamic voltage and frequency scaling for stream programs. In *Proceedings of ICF*. ACM, New York, NY, Article 40, 10 pages. DOI: http://dx.doi.org/10.1145/2016604.2016654

Thomas Rauber and Gudula Rünger. 2015. Modeling and analyzing the energy consumption of fork-join-based task parallel programs. *Concurr. Comput.: Pract. Exp.* 27, 1 (2015), 211–236. DOI: http://dx.doi.org/10.1002/cpe.3219

Haris Ribic and Yu David Liu. 2014. Energy-efficient work-stealing language runtimes. In *Proceedings of ASPLOS*. ACM, 513–528. DOI: http://dx.doi.org/10.1145/2541940.2541971

E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. 2012. Power-management architecture of the intel microarchitecture code-named sandy bridge. In *Proceedings of MICRO*, Vol. 32. 20–27. DOI: http://dx.doi.org/10.1109/MM.2012.12

Bagher Salami, Mohammadreza Baharani, and Hamid Noori. 2014. Proactive task migration with a self-adjusting migration threshold for dynamic thermal management of multi-core processors. *J. Supercomput.* 68, 3 (2014), 1068–1087. DOI: http://dx.doi.org/10.1007/s11227-014-1140-y

Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate data types for safe and general low-power computation. In *Proceedings of PLDI*. ACM, New York, NY, 164–174. DOI: http://dx.doi.org/10.1145/1993498.1993518

Aravind Sankar, Soham Chakraborty, and V. Krishna Nandivada. 2016. Improved MHP analysis. In *Proceedings of CC*. 207–217.

V. Saraswat, B. Bard, P. Igor, O. Tardieu, and D. Grove. 2014. *X10 Language Specification Version 2.4*. Technical Report. IBM.

K. S. Shim, M. Lis, O. Khan, and S. Devadas. 2014. Thread migration prediction for distributed shared caches. *IEEE Comput. Arch. Lett.* 13, 1 (Jan 2014), 53–56. DOI: http://dx.doi.org/10.1109/L-CA.2012.30

Peter Thoman, Herbert Jordan, Simone Pellegrini, and Thomas Fahringer. 2012. Automatic OpenMP loop scheduling: A combined compiler and runtime approach. In *Proceedings of IWOMP*. Springer-Verlag, Berlin, 88–101. DOI: http://dx.doi.org/10.1007/978-3-642-30961-8_7

R. Wattenhofer. 2011. *Lecture notes on Principles of Distributed Computing*. Swiss Federal Inst. of Tech. Zurich.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem-overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 36 (May 2008), 53 pages. DOI: http://dx.doi.org/10.1145/1347375.1347389

Tomofumi Yuki and Sanjay Rajopadhye. 2013. Folklore confirmed: Compiling for speed = compiling for energy. In *Proceedings of LCPCW*. Springer International Publishing, Cham, Switzerland,169–184. DOI: http://dx.doi.org/10.1007/978-3-319-09967-5_10

M. Zahran and M. Franklin. 2003. Dynamic thread resizing for speculative multithreaded processors. In *Proceedings of ICCD*. 313–318. DOI: http://dx.doi.org/10.1109/ICCD.2003.1240912

Weirong Zhu, Juan Del Cuvillo, and Guang R. Gao. 2006. Performance characteristics of OpenMP language constructs on a many-core-on-a-chip architecture. In *Proceedings of IWOMP*. Springer-Verlag, Berlin, 230–241.

Received February 2017; revised August 2017; accepted August 2017