Cuttlefish: Library for Achieving Energy Efficiency in Multicore Parallel Programs

Sunil Kumar
IIIT-Delhi
India

Akshat Gupta
IIIT-Delhi
India

Vivek Kumar
IIIT-Delhi
India

Sridutt Bhalachandra
Lawrence Berkeley
National Laboratory
USA

ABSTRACT

A low-cap power budget is challenging for exascale computing. Dynamic Voltage and Frequency Scaling (DVFS) and Uncore Frequency Scaling (UFS) are the two widely used techniques for limiting the HPC application's energy footprint. However, existing approaches fail to provide a unified solution that can work with different types of parallel programming models and applications.

This paper proposes *Cuttlefish*, a programming model oblivious C/C++ library for achieving energy efficiency in multicore parallel programs running over Intel processors. An online profiler periodically profiles model-specific registers to discover a running application's memory access pattern. Using a combination of DVFS and UFS, Cuttlefish then dynamically *adapts* the processor's core and uncore frequencies, thereby improving its energy efficiency. The evaluation on a 20-core Intel Xeon processor using a set of widely used OpenMP benchmarks, consisting of several irregular-tasking and work-sharing pragmas, achieves geometric mean energy savings of 19.4% with a 3.6% slowdown.

CCS CONCEPTS

• **Software and its engineering → Power management**.

KEYWORDS

Multicore parallelism, DVFS, UFS, energy efficiency

ACM Reference Format:
Sunil Kumar, Akshat Gupta, Vivek Kumar, and Sridutt Bhalachandra. 2021. Cuttlefish: Library for Achieving Energy Efficiency in Multicore Parallel Programs. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476163

1 INTRODUCTION

The current generation of supercomputers is composed of highly multicore processors. The ever-increasing core counts are likely to continue in the present and post-Exascale (10^{18} flops/sec) era. The Top500 list for November 2020 shows 50% of supercomputers have 20-24 cores per socket, and the fastest supercomputer, Fugaku, has 48 cores [2]. Limiting power consumption is one of the significant challenges today. For an exascale system to be delivered within a power budget of 20–30 MW [33], the hardware and software components need to be highly energy-efficient.

Cache-coherent shared memory processors that dominate modern high-performance computing (HPC) systems contain multiple levels of caches in each socket. Most processor architectures are a combination of *core* and *uncore* elements. The uncore includes all chip components outside the CPU core [21], such as shared caches, memory controllers, and interconnects (QPI on Intel platforms). The processor power consumption can be regulated using multiple knobs for such as Dynamic Voltage and Frequency Scaling (DVFS) [28] and Dynamic Duty-Cycle Modulation (DDCM) [24] for CPU cores, and Uncore Frequency Scaling (UFS) [21] for the uncore. DVFS allows scaling down the core's voltage and frequency, thereby reducing power consumption as lowering the voltage has a squared effect on active power consumption. The static losses have become significant with transistor shrinkage, undermining the previously achievable power savings with DVFS [15]. Nevertheless, DVFS is still the most effective and widely used power control.

Most energy-efficient HPC research has revolved around reducing the processor/core frequency by using DVFS or DDCM with minimum performance impact (see Section 6). These implementations adapt the core frequencies as follows: a) based on trace data collected from the application's offline profiling, b) whenever parallel programs encounter slack time due to unbalanced computation or inter-node communication delays, c) gathering workload characteristics by profiling iterative and loop-based applications, and d) programming model and runtime-dependent application profiling. A standard limitation of these studies is that they are specific to a particular application or programming model. Recently, UFS has been explored for achieving energy efficiency based on machine learning models on offline generated application data [3] and by dynamically profiling the DRAM power consumption [18]. However, these studies lack an integrated approach for core and uncore frequency scaling.

This paper explores a one-stop solution for achieving energy efficiency on multicore processors called Cuttlefish that dynamically adapts both the core and uncore frequencies on Intel processors without requiring training runs. Cuttlefish is oblivious to the parallel programming model and the concurrency decomposition techniques used in an application. In their application, programmers only need to use two API functions, `cuttlefish::start()` and `cuttlefish::stop()`, to define the scope requiring an energy-efficient execution. At runtime, Cuttlefish then creates a daemon thread that periodically profiles the application's Memory Access

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476163

Benchmark Name	Brief Description	Configuration	Parallelism Style	OpenMP Time (sec)	TIPI Range	Total TIPI Slabs	
						Distinct	Frequent
UTS	Unbalanced Tree Search [36]	TIXXL	Irregular Tasks	69.9	0-0.004	1	1
SOR-irt	Successive Over-Relaxation (SOR) [7]	32Kx32K (200)	Irregular Tasks	69.1	0.024-0.028	1	1
SOR-rt	——————— " ———————	——— " ———	Regular Tasks	69.4	0.024-0.028	1	1
SOR-ws	——————— " ———————	——— " ———	Work-sharing	68.7	0.012-0.028	3	1
Heat-irt	Heat diffusion (Jacobi-type iteration) [35]	32Kx32K (200)	Irregular Tasks	76.6	0.056-0.076	4	1
Heat-rt	——————— " ———————	——— " ———	Regular Tasks	75.5	0.056-0.072	3	2
Heat-ws	——————— " ———————	——— " ———	Work-sharing	70.9	0.012-0.068	11	1
MiniFE	Finite Element Mini-Application [1, 11]	256x512x512 (200)	Work-sharing	78.5	0.068-0.152	16	1
HPCCG	High Performance Computing Conjugate Gradients [1, 11]	256x256x1024 (149)	Work-sharing	60	0.060-0.148	17	1
AMG	Algebraic Multigrid solver [32]	256x256x1024 (22)	Work-sharing	63.7	0.060-0.332	60	2

Table 1: Description of the benchmarks used in this paper for the evaluation of Cuttlefish

Pattern (MAP) by reading the Model-Specific Registers (MSR) available on all Intel platforms. Each MAP is uniquely identified as the ratio of uncore event TOR_INSERT [25] and instructions retired. Cuttlefish daemon then uses DVFS to determine the core frequency that would provide maximum energy savings with minimal impact on execution time. After finding and setting this optimal core frequency, Cuttlefish daemon then uses the same exploration-based technique to determine the optimal uncore frequency. To reduce the loss in performance, it uses several runtime optimizations to complete the frequency explorations quickly. Once both optimal core and uncore frequencies are found for a given MAP, the rest of the program would execute at these frequencies. Cuttlefish would repeat this frequency exploration for core and uncore every time a new MAP is discovered. We chose a set of ten widely-used OpenMP benchmarks, consisting of several irregular-tasking and work-sharing pragmas to evaluate Cuttlefish on a 20-core Intel processor. We show that Cuttlefish significantly improves the energy-efficiency with a negligible impact on performance. We also evaluated Cuttlefish by implementing a subset of these benchmarks using **async–finish** task parallelism [30] to demonstrate that Cuttlefish is oblivious to the parallel programming model.

In summary, this paper makes the following contributions:

- Cuttlefish, a parallel programming model oblivious C/C++ library for achieving energy efficiency in multicore parallel programs running on Intel processors.
- A novel light-weight runtime for Cuttlefish that periodically monitors the memory access pattern of a running application and then dynamically adapts the core and uncore frequencies using DVFS and UFS, respectively.
- Evaluation of Cuttlefish on a 20-core Intel Xeon Haswell E5-2650 processor by using multiple HPC benchmarks and mini-applications implemented in OpenMP and **async–finish** programming model. Our evaluation shows that Cuttlefish can significantly improve energy efficiency with negligible impact on the execution time for several irregular-tasking and work-sharing pragmas.

2 EXPERIMENTAL METHODOLOGY

Before presenting the motivating analysis for Cuttlefish, we first describe our experimental methodology. To cover a broad spectrum of parallel applications, we have chosen benchmarks based

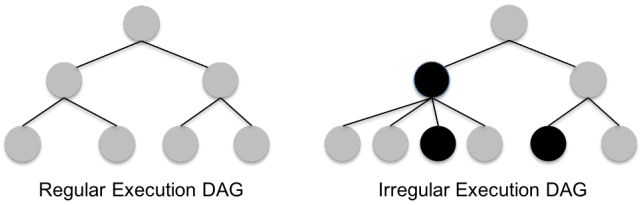


Figure 1: Heat-irt and SOR-irt use the irregular execution DAG, whereas Heat-rt and SOR-rt use the regular execution DAG. Non-root and non-leaf grey and black nodes have degrees three and five, respectively. Each node is parallel task

on the following attributes: a) memory access patterns, b) micro-kernels and real-world mini-applications, c) execution DAG, and d) concurrency decomposition techniques. We target six widely-used benchmarks for our experimental evaluation. These benchmarks mainly use OpenMP pragmas, but a few were ported to **async–finish** programming model supported by the HClib work-stealing library [19, 30] to evaluate Cuttlefish using two different parallel programming models. We created three variants (both for OpenMP and HClib) that differ in execution DAG and concurrency decomposition technique for two of these benchmarks, Heat and SOR. Two of these variants use dynamic task parallelism and vary in terms of irregular (*irt*) and regular (*rt*) execution DAGs. The third variant is a non-recursive implementation that uses work-sharing (*ws*) based static loop partitioning. The technique described in Chen et al. [8] was used for converting the loop-level parallelism in Heat and SOR into regular and irregular execution DAG, as shown in Figure 1. In total, we have ten benchmarks/mini-applications. Table 1 describes them and their respective configurations.

We ran all experiments on an Intel Xeon Haswell E5-2650 v3 20-core processor with a total of 94GB of RAM. The operating system (OS) was Ubuntu 16.04.7 LTS. This processor supports core and uncore frequencies between 1.2GHz–2.3GHz and 1.2GHz–3.0GHz, respectively, both in steps 0.1GHz. We used MSR-SAFE [34] for saving and restoring Model-Specific-Register (MSR) values. The turboboost feature on the processor was disabled, and each benchmark used interleaved memory allocation policies supported by the numactl library. The HClib implementation from the official Github repository with the commit id ab310a0 is used. Both OpenMP and HClib

versions of the benchmarks used the Clang compiler version 3.8.0 with the -O3 flag. All 20 threads used in the experiments were bound to their respective physical CPUs. Although we use an Intel Haswell processor in our evaluation, more recent Intel processors can use Cuttlefish by updating the MSRs specific to them. While the latest AMD processors do support per-core DVFS similar to Intel, based on public documentation available, it is not clear if support for UFS or TIPI measurement exists.

We evaluated three variants of Cuttlefish: a) Cuttlefish-Core that only adapts the CPU frequency, b) Cuttlefish-Uncore that only adapts the Uncore frequency, and c) Cuttlefish that adapts both the CPU and Uncore frequencies. We compared these implementations against each benchmark's `Default` execution by setting the Intel power governor to `performance` policy. The performance power governor fixes the CPU frequency to the maximum. We chose `performance` power governor for `Default` execution as this same setting is used by several supercomputers in production [10, 20]. For Cuttlefish-based executions, the power governor is set to userspace to allow changes to the CPU frequency through the library. We changed the uncore frequency scaling option to *Auto* in the BIOS, allowing the Intel firmware to modulate the uncore frequency during the `Default` execution. The algorithm used by Intel is highly sensitive to memory requests. During Cuttlefish executions, the runtime controls the uncore frequency dynamically inside the library by writing the desired frequency value in the UFS MSR (`0x620`). We executed each implementation ten times and reported the mean value along with a 95% confidence interval.

3 MOTIVATING ANALYSIS

An application's memory access pattern (MAP) can be primarily classified as memory-bound or compute-bound. Memory-bound applications have a high number of memory accesses as compared to compute-bound applications. This section presents two analyses to support that MAP of an application can be accurately identified using MSRs (Section 3.1), and changing the core and uncore frequencies based on MAP can lead to energy savings (Section 3.2).

3.1 TOR Inserts per Instruction (TIPI)

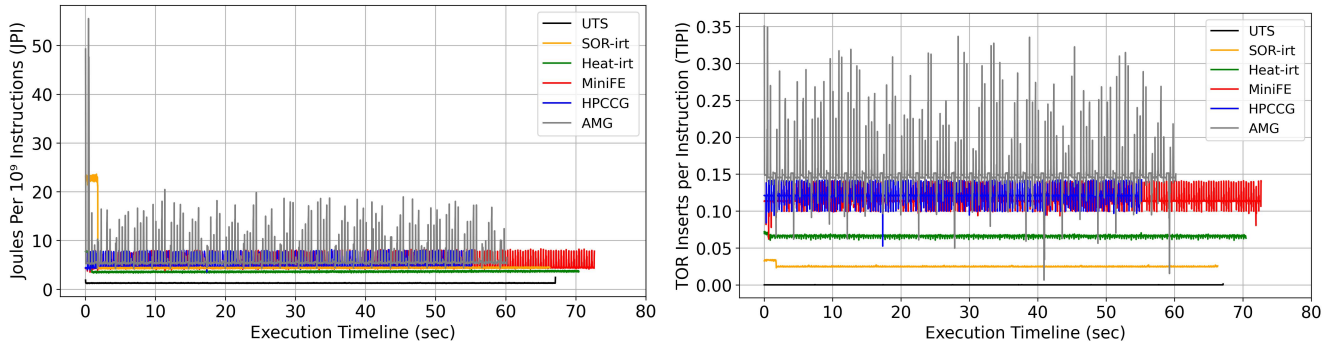
To accurately identify the MAP of an application, we propose the metric TOR Inserts per Instruction (*TIPI*), which is calculated as the ratio of total TOR Inserts (`TOR_INSERT.MISS_LOCAL + TOR_INSERT.MISS_REMOTE`) and total instructions retired (`INST_RETIRED.ANY`). Any requests coming to the LLC from the processor cores are place in TOR (Table of Request). `TOR_INSERT` MSR [25] is available on all Intel processors from Haswell generation and onwards. This MSR counts the number of memory requests that come to the socket-specific Last-Level Cache (LLC) from each core. It supports various Unit Masks (umask) to select the type of memory requests to be counted. `MISS_LOCAL` counts the misses to the local caches and memory, and `MISS_REMOTE` counts the misses to the remote caches and memory as per the Intel documentation [25]. We consider both these umasks as our experimental machine is a two-socket NUMA machine (Section 2). The other metric used is Joules per Instructions (*JPI*). JPI is calculated as the ratio of the total energy spent by the processor and the total instructions retired. The package energy is measured using the Intel RAPL MSRs.

We found that compute-bound and memory-bound applications have low and high TIPI values, respectively. It is also observed that TIPI has a strong correlation with JPI as an increase in TIPI also increases the JPI. These observations are reported in Figure 2. The core and uncore frequencies are initially set to the maximum, and periodically the TIPI and JPI for an application is recorded at every T_{inv} intervals. The T_{inv} is fixed at 20 milliseconds and is empirically derived for all experiments reported in this paper. The x-axis in Figure 2 represents the execution timeline of each benchmark in seconds. The y-axis in Figure 2(b) and Figure 2(a) shows the TIPI and JPI, respectively, during the benchmark's execution. The result of only `Heat-irt` and `SOR-irt` are reported as their variants have similar behaviours. As JPI is highly sensitive to TIPI, JPI is used to measure energy efficiency for a given TIPI. In Figure 2(a), `SOR-irt` has a higher JPI than `Heat-irt`, although `SOR-irt`'s TIPI is less than that of `Heat-irt` (Figure 2(b)). For `MiniFE`, `HPCCG`, and `AMG`, it is observed that TIPI varies throughout the execution. However, an increase in TIPI shows an increase in JPI. This implies that the TIPI v/s JPI trend holds only within an application, and the same TIPI value could have different JPIs for different applications.

3.2 Analysis of DVFS and UFS with TIPI

Cuttlefish has an online profiler that explores optimal core and uncore frequencies for every unique TIPIs discovered during a benchmark execution. To limit the search space, we divide each unique TIPIs in fixed slabs of 0.004 (derived empirically), i.e., TIPI values 0.004, 0.005, and 0.007 would be reported under the TIPI range 0.004-0.008. Hereafter, a TIPI value is reported in terms of its range instead of the actual value. Figure 3 shows the relation between TIPI and JPI for benchmarks at different core and uncore frequencies. In Figure 3(a), the Uncore Frequency (UF) is set to max (3.0 GHz) and each benchmark is executed with three different Core Frequencies (CF) – min (1.2 GHz), mid (1.8 GHz) and max (2.3 GHz). In Figure 3(b), the CF is set to max (2.3 GHz) and each benchmark is executed with three different UFs – min (1.2 GHz), mid (2.1 GHz) and max (3.0 GHz). In each execution of a benchmark, we record the TIPI and JPI at every T_{inv} and then calculate the average JPI for the frequently occurring TIPIs. TIPIs found in more than 10% of total T_{inv} samplings are mentioned as frequently occurring TIPIs during an execution.

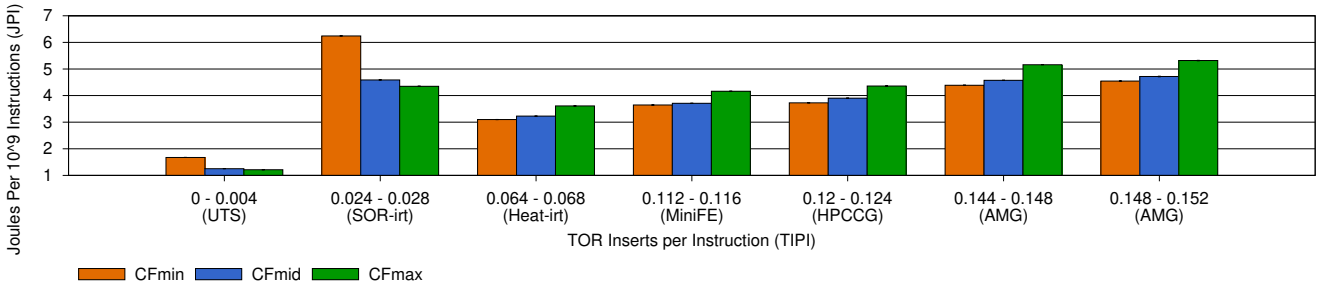
We can observe that `UTS` and `SOR-irt` have a low TIPI range and are compute-bound as their JPI decreases with increasing CF and JPI increases with increasing UF. `Heat-irt`, `MiniFE`, `HPCCG`, and `AMG` have a high TIPI range and are memory-bound applications as they behave precisely opposite to `UTS` and `SOR-irt`. JPI of these four benchmarks increases with increasing the CF, and their JPI decreases with increasing the UF. Although `Heat-irt`, `MiniFE`, `HPCCG`, and `AMG` are memory-bound, max uncore frequency is not apt for their TIPI range. This analysis implies that when TIPI is low, CPU cores should run at a higher frequency, and uncore should run at a lower frequency. In contrast, when TIPI is high, uncore should run at a higher frequency, and CPU cores should run at a lower frequency as the latter would frequently halt due to the memory access latency.



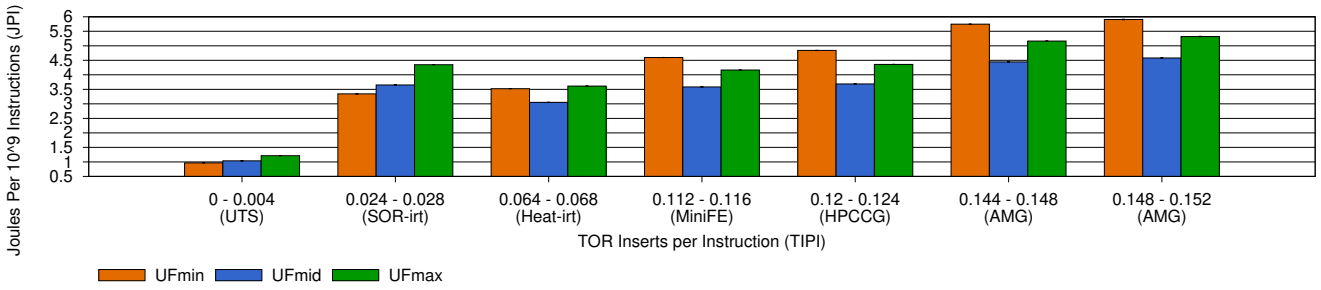
(a) Variation in JPI during the course of execution

(b) Variation in TIPI during the course of execution

Figure 2: Relation between TIPI and JPI. For each benchmark, JPI increases with the increase in TIPI



(a) Effect of fixing Uncore and changing Core frequency



(b) Effect of fixing Core and changing Uncore frequency

Figure 3: Motivational analysis to understand the effect of core and uncore frequencies on the JPI at each TIPI

4 DESIGN AND IMPLEMENTATION

The previous section highlighted that TIPI could be used to distinguish the MAP of an application and for estimating its optimal core and uncore frequencies. Hence, to achieve energy efficiency in an application with minimal loss in performance, it is vital to use a low-overhead online profiler to discover the TIPI ranges dynamically and appropriately set the core and uncore frequencies. We approach the problem using a lightweight daemon thread, Cuttlefish, that runs in tandem with the application to monitor the TIPI and JPI by activating itself after every T_{inv} durations to minimize the time-sharing of the CPU with the application. In their application, programmers only need to use two API functions, `cuttlefish::start()` and `cuttlefish::stop()`, to define the scope requiring an energy-efficient execution. Whenever Cuttlefish discovers a new TIPI range, it uses DVFS to explore the optimal core frequency, followed by UFS to explore the optimal uncore frequency for this TIPI range. An optimal frequency is the one that has the lowest JPI. Cuttlefish start its execution without any prior information on TIPI ranges and optimal frequencies. The insight is to minimize the application slowdown due to frequency exploration by compacting the exploration range on the fly based

on the optimal core and uncore frequencies of other TIPI range discovered since the start of this application.

4.1 Cuttlefish daemon loop

Algorithm 1: Cuttlefish daemon thread method

```
1  CF_prev ← CF_max; UF_prev ← UF_max
2  set_freq(CF_prev, UF_prev)
3  sleep(warmup duration)
4  CF_next ← UF_next ← -1
5  N_curr← N_prev← NULL        // TIPI LinkedList nodes
6  while shutdown not initiated from stop API do
7      Read TIPI and JPI values from MSR
8      if TIPI NOT found in LinkedList then
9          N_curr← insert_in_sortedLinkedList(TIPI)
10         CF_LB ← CF_min; CF_UB ← CF_max        // for N_curr
11         UF_LB ← UF_min; UF_UB ← UF_max        // for N_curr
           // Section 4.4
12         Reduce CF_LB/CF_RB if LinkedList_size>1
13         CF_next ← find(CF, JPI, CF_prev, N_prev, N_curr)
14         UF_next ← UF_max
15     else
16         N_curr← fetch_from_sortedLinkedList(TIPI)
17         if N_curr.CF_opt & N_curr.UF_opt = -1 then
18             CF_next ← find(CF, JPI, CF_prev, N_prev, N_curr)
19             UF_next ← UF_max
20             if N_curr.CF_next = N_curr.CF_opt then
21                 Estimate UF_LB & UF_RB (Algorithm 3)
                   // Section 4.4
22                 Reduce UF_LB/UF_RB if LinkedList_size>1
23                 UF_next ← N_curr.UF_RB
24             end
25         else if N_curr.UF_opt = -1 then
26             CF_next ← N_curr.CF_opt
27             UF_next←find(UF, JPI, UF_prev, N_prev, N_curr)
28         else
29             CF_next ← N_curr.CF_opt
30             UF_next ← N_curr.UF_opt
31         end
32     end
33     set_freq(CF_next , UF_next )
34     N_prev← N_curr
35     CF_prev ← CF_next; UF_prev ← UF_next;
36     sleep ( T_inv)
37 end
```

Cuttlefish daemon thread is spawned by the API `cuttlefish::start()` and is pinned to a fixed core. Algorithm 1 lists the pseudocode implementation of this daemon thread. It sets the core and uncore frequencies to the maximum (Line 2). It runs in a loop where it first calculates the TIPI and JPI for the whole processor (Line 7), followed by execution of the Cuttlefish runtime policy (Line 8–Line 33), and finally goes back to sleep for T_{inv} duration (Line 36). The implementation for measuring the TIPI and JPI in Cuttlefish is inspired by the RCRtool [38]. It continues the execution of this loop until `cuttlefish::stop()` is called inside the user application. From Figure 2(b) and Figure 2(a), we can observe that TIPI and JPI fluctuate heavily at the beginning of the execution timeline. This fluctuation is prominent across all three variants of Heat and SOR and `AMG` (seven out of ten benchmarks), whereas it's minuscule in the remaining three benchmarks. This instability is due to cold caches at the start of execution, but it becomes stable after a while. Hence, to avoid recording unstable values of TIPI and JPI, the Cuttlefish daemon loop activates only after a warmup duration of two seconds (Line 3).

4.2 Cuttlefish runtime policy

After every T_{inv}, the Cuttlefish policy (Line 8–Line 33) is invoked where it uses DVFS and UFS to determine Optimal Core Frequency (CF_{opt}) and Optimal Uncore Frequency (UF_{opt}) for a given TIPI range. Cuttlefish maintains a sorted doubly linked list of unique TIPI ranges discovered during an execution. This list is empty at the beginning (Line 8). Each node in this linked list has the following fields: TIPI range, arrays to store JPI for each core and uncore frequencies, latest exploration range for core and uncore frequencies, CF_{opt}, and UF_{opt}. Figure 4(a) shows one such node in this list for a hypothetical processor having seven frequency levels, `A`–`G`, for both core and uncore. `A` and `G` are the lowest and highest frequencies, respectively, in this processor. To explain the Cuttlefish runtime policy's working, we use this same hypothetical processor in all our discussions hereafter. The pseudocode implementation of core and uncore frequency exploration is shown in Algorithm 2.

4.3 Frequency exploration in single TIPI-range

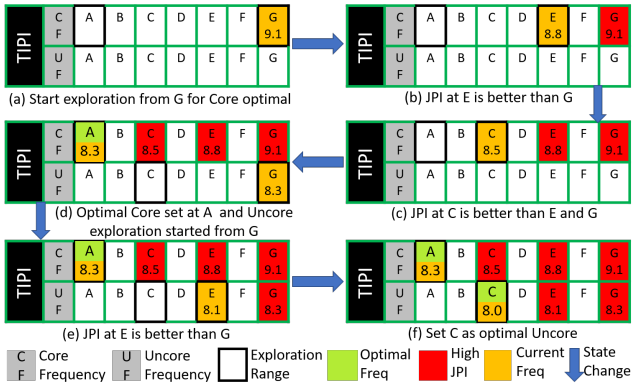


Figure 4: Frequency exploration for a single TIPI

Here, we explain the steps followed by the Cuttlefish in exploring the CF_{opt} and UF_{opt} for benchmarks having a single TIPI range (Table 1 lists the total number of distinct TIPI slabs in each benchmark). Cuttlefish starts the exploration first for the CF_{opt} within the default exploration range with the left bound core and uncore frequencies (CF_{LB} and UF_{LB}, respectively) set to minimum, and the right bound core and uncore frequencies (CF_{RB} and UF_{RB}, respectively) set to maximum (Algorithm 1, Line 10–Line 11). For the

Algorithm 2: Method `find` for CF / UF exploration

Input: type, JPI_{curr}, FQ_{curr}, N_{prev}, N_{curr}
Output: FQ_{next}

```
1  ptr = N_curr.FQ_table[type] // type is CF/UF
2  if ptr.FQ_LB & ptr.FQ_RB are adjacent then
3  |   Choose FQ_opt from FQ_LB & FQ_RB // Figure 5
4  |   return ptr.FQ_opt
5  end
   // Discard JPI in TIPI transition phase
6  if N_prev = N_curr then
7  |   N_curr.JPI_table[type][FQ_curr] ← average(JPI_curr)
8  end
   /* JPI_avg at any FQ is average of 10 readings.
      Hence, Line 9 / Line 11 equates to true during
      TIPI transition due to incomplete JPI_avg    */
9  if JPI_avg NOT exists for ptr.FQ_RB then
10 |   return ptr.FQ_RB
11 else if JPI_avg NOT exists for ptr.FQ_RB-2 then
12 |   return ptr.FQ_RB-2
13 end
14 if JPI_avg at ptr.FQ_RB-2 is less than ptr.FQ_RB then
15 |   ptr.FQ_RB ← ptr.FQ_RB-2
16 |   FQ_next = (ptr.FQ_RB - ptr.FQ_LB > 2) ? ptr.FQ_RB-2 : ptr.FQ_LB
17 else
18 |   FQ_next ← ptr.FQ_LB ← ptr.FQ_RB-1
19 end
20 if ptr.FQ_LB & ptr.FQ_RB are same then
21 |   FQ_next ← ptr.FQ_opt ← ptr.FQ_RB
22 end
   // Section 4.5
23 Update FQ_LB or FQ_RB of other TIPIs if LinkedList_size>1
24 return FQ_next
```

hypothetical processor, this exploration is shown in Figure 4, where CF_{LB} =CF_A and CF_{RB} =CF_G. As execution starts with the frequencies set to CF_G and UF_G, after the first T_{inv}, the JPI value for CF_G is recorded. To ensure stability in the JPI value, Cuttlefish computes the average JPI at CF_G across ten T_{inv}. Henceforth, the JPI value reported for any CF and UF in Cuttlefish is an average of ten readings (Algorithm 2, Line 7). Frequency exploration cannot continue until an average of ten readings of JPI is available at a given frequency. Once the JPI is finalized for CF_G (Figure 4(a)), Cuttlefish will use DVFS to set the core frequency at CF_E without changing the uncore (Algorithm 1, Line 33). Frequencies are always explored at steps of two to minimize the exploration steps. Cuttlefish will then compare the JPIs at CF_G and CF_E (Figure 4(b)). This comparison is performed at Line 14–Line 16 in Algorithm 2. As CF_E has lower JPI than CF_G, Cuttlefish will update CF_{RB} =CF_E and then set the frequency at CF_C. As it turns out that CF_C has lower JPI than CF_E, now CF_{RB} =CF_C (Figure 4(c)), and the frequency would be set to CF_A. As even CF_A has lower JPI than CF_C, CF_{LB} =CF_{RB} =CF_A (Algorithm 2, Line 21), thereby making it as CF_{opt} for this TIPI (Figure 4(d)).

Cuttlefish explores both core and uncore frequencies linearly at the steps of two instead of using binary search for reducing the total number of explorations and performance degradation. Except when the optimal frequency lies at the boundary of the exploration range, Cuttlefish cannot use the naive binary search algorithm. When the optimal frequency lies between the exploration range, JPI will increase when moving left/right from the optimal frequency. For example, assume that JPI_A<JPI_G and E is the optimal frequency for some TIPI (Figure 4). Applying a binary search to find E would require measuring JPIs at `mid`, `mid+1`, and `mid-1` at each split. This would require more explorations as compared to the linear search (steps of two). For the worst-case scenario (optimal at default minimum), the total number of explorations for CF_{opt} on our Intel Haswell processor using linear search would be six (`total_frequencies/2`) compared to eight by using the modified binary search. Also, as the JPI at each frequency is an average of ten values, exploring `mid`, `mid+1`, and `mid-1` frequencies in binary search exploration would cause more performance degradation than linear search (highest to lowest).

After exploring CF_{opt} =CF_A, Cuttlefish will start the exploration for UF_{opt} by fixing the core frequency to CF_A (Algorithm 1, Line 20). However, for UF_{opt}, Cuttlefish would *not* explore the default exploration range of UF_{LB} =UF_A to UF_{RB} =UF_G. Instead, it uses Algorithm 3 to estimate the uncore exploration range (Algorithm 1, Line 21). Algorithm 3 is based on our observation in Section 3.2 that a *high* core frequency as optimal implies a *low* uncore frequency as optimal, and vice-versa. Hence, CF_{opt} =CF_{max} implies that UF_{opt} =UF_{min}, and CF_{opt} =CF_{min} implies UF_{opt} =UF_{max}. Now, for estimating the optimal uncore (UF_{opt_est}) based on the CF_{opt}, we use the curve-fitting technique, where we map the coordinates (CF_{max}, UF_{min}) and (CF_{min}, UF_{max}) on a straight line. It is shown in Lines 2-3 of Algorithm 3. We also saw the latest Intel processor's trend that the number of uncore frequencies and core frequencies is roughly similar. Hence, to prepare a short exploration range for UF, we calculate the ratio of the number of uncore frequencies and core frequencies. To get a modest exploration space, we are multiplying the ratio by a constant of 4. This ratio calculation is shown in Line 1 of Algorithm 3. Line 4–Line 11 of Algorithm 3 shows the UF_{LB} and UF_{RB} calculation that uses the range and ratio described above. By using this Algorithm, Cuttlefish uses the exploration range of UF_{LB} =UF_C and UF_{RB} =UF_G, and continues the UF_{opt} exploration similar to the CF_{opt} exploration (Figure 4(e) and Figure 4(f)).

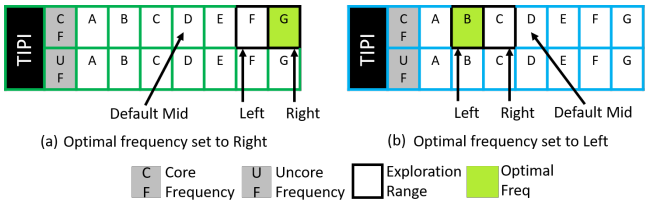


Figure 5: Determining CF_{opt} when CF_{RB}-CF_{LB} =1

Figure 4 depicted the situation where reducing the frequency was reducing the JPI. However, there will be scenarios where reducing the frequency increases the JPI. These situations are depicted in Figure 5 when low bound and high bound frequencies are adjacents

Algorithm 3: Algorithm to find UF exploration range

Input: CF_{opt}
Output: UF_{LB}, UF_{RB}
1 Range ← 4 * (UF_{max} - UF_{min} + 1)/(CF_{max} - CF_{min} + 1)
2 α ← (UF_{max} - UF_{min})/(CF_{max} - CF_{min})
3 UF_{opt_est} ← UF_{max} - (α * (CF_{opt} - CF_{min}))
4 UF_{LB} ← max(UF_{min} , UF_{opt_est} - Range/2)
5 UF_{RB} ← min(UF_{max} , UF_{opt_est} + Range/2)
6 **if** UF_{max} - UF_{opt_est} ≤ Range/2 **then**
7 | UF_{LB} ← UF_{LB} - (UF_{opt_est} + Range/2 - UF_{max})
8 **end**
9 **if** UF_{opt_est} - UF_{min} ≤ Range/2 **then**
10 | UF_{RB} ← UF_{RB} + (UF_{min} - (UF_{opt_est} - Range/2))
11 **end**

(Algorithm 2, Line 2–Line 5). In Figure 5(a), JPI at CF_E was higher than that at CF_G. In this case, CF_{LB} =CF_F. However, as the CF_{LB} and CF_{RB} (CF_G) are consecutive frequencies, CF_{opt} will be set to CF_G to minimize the loss in performance as high CF indicates a compute-bound MAP. Figure 5(b) depicts a situation where JPI at CF_A was higher than that at CF_C. In this case, CF_{LB} =CF_B. Again, left bound CF_{LB} and CF_{RB} (CF_C) are consecutive frequencies, but this time CF_{opt} will be set to CF_B to maximize the energy efficiency as low CF indicates a memory-bound MAP. These two scenarios depicted for CF can also occur for UF.

4.4 Frequency exploration for subsequent TIPIs

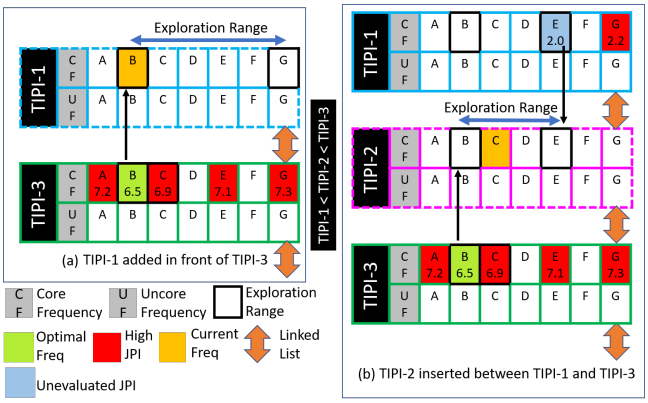


Figure 6: Insertion of a node for a newly found TIPI in the doubly linked list, and determining the CF exploration range for this new TIPI

An application could have a variety of MAP, e.g., in AMG, we found 60 distinct MAPs (Table 1). Using the default CF exploration range CF_{LB} =CF_A and CF_{RB} =CF_G for every newly discovered TIPIs can easily degrade the application performance, especially on a processor supporting a wide range of frequencies. Hence, Cuttlefish uses the default CF_{LB} and CF_{RB} only for the first TIPI (Algorithm 1, Line 10–Line 11). For subsequent TIPIs, it sets CF_{LB}, and CF_{RB} based on the CF_{opt} and the latest values of CF_{LB} and CF_{RB} in previously discovered TIPI ranges. Here, the insight is to use a sorted doubly linked list of TIPI ranges, where moving from left to right in the linked list signifies a shift from compute-bound MAP to memory-bound MAP. Hence, when a new TIPI node is inserted in the linked list, Cuttlefish can look up the CF_{opt}, CF_{LB} and CF_{RB} of its adjacent nodes (left and right) to dynamically decide the CF_{LB} and CF_{RB} for this new TIPI node. Compared to CF, the exploration range of UF is already smaller (Algorithm 3). Still, Cuttlefish attempts to reduce this exploration range even further for subsequent TIPIs by following the same insight mentioned above for CF. This optimization is shown in Figure 6 for CF exploration (Algorithm 1, Line 12) and in Figure 7 for UF exploration (Algorithm 1, Line 22).

Figure 6(a) shows an execution phase N (time elapsed since Cuttlefish started is N×T_{inv}), at which MAP was pointing to TIPI-3, but a new TIPI range, TIPI-1, was discovered by the Cuttlefish. Comparing TIPI-1 with other TIPIs in the linked list, it is found that TIPI-1 should be added in the front. The position of the TIPI-1 signifies that it is compute-bound *relative* to the TIPI-3, i.e., CF_{opt} for TIPI-1 will be the same or greater than the CF_{opt} for TIPI-3. Hence, CF_{LB} =CF_B and CF_{RB} =CF_G for TIPI-1. As there was a TIPI transition in the last T_{inv} phase, JPI calculated for CF_B is not recorded in the CF table for TIPI-1 (Algorithm 2, Line 6–Line 8). It will be recorded after the next T_{inv} phase. After a few execution duration with the MAP still pointing to TIPI-1, another new TIPI, TIPI-2, was discovered by the Cuttlefish. This is shown in Figure 6(b). At this time, Cuttlefish calculated the JPI for CF_C to compare it with the JPI at CF_E in TIPI-1. However, it won't be able to do so because the new TIPI-2 was discovered in the last T_{inv}. By comparing TIPI-2 with other TIPIs, it found that the TIPI-2 node should be inserted between the TIPI-1 and TIPI-3 nodes in the linked list. This signifies that TIPI-2 is memory-bound relative to the TIPI-1 but compute-bound relative to the TIPI-3. Hence, going by the same algorithm as in TIPI-1, TIPI-2's CF_{LB} and CF_{RB} should be the CF_{opt} for TIPI-3 and TIPI-1, respectively. However, as the CF_{opt} for TIPI-1 is still not found, TIPI-2's CF_{RB} will be set to CF_{RB} of TIPI-1 (CF_E). Due to the TIPI transition in the last phase, the JPI (CF_C) of the last T_{inv} will not be recorded in TIPI-2.

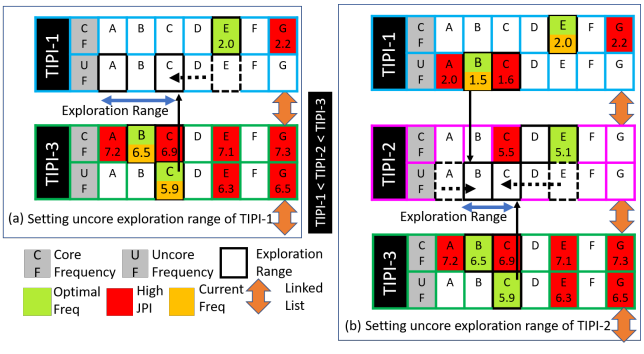


Figure 7: Determining the UF exploration range based the UF_{opt} of adjacent TIPI nodes in doubly linked list

Figure 7 demonstrates the same optimization discussed above, but only for uncore exploration. Figure 7(a) shows an execution

phase when Cuttlefish has found the CF_{opt} for TIPI-1, and now it has to start the exploration of UF_{opt} for TIPI-1. By using Algorithm 3, UF_{LB} =UF_A and UF_{RB} =UF_E for TIPI-1. As TIPI-1 is compute-bound relative to the TIPI-3, its UF_{opt} will be the same or lower than the UF_{opt} (UF_C) for TIPI-3. Hence, UF_{RB} =UF_C for TIPI-1. This is precisely opposite to what was done in the case of CF. Figure 7(b) shows an execution phase when Cuttlefish has found the CF_{opt} for TIPI-2, and now it has to start exploring UF_{opt} for TIPI-2. It sets the UF_{opt} of TIPI-3 as UF_{RB} for TIPI-2 and UF_{opt} of TIPI-1 as UF_{LB} of TIPI-2. TIPI-2 is memory-bound relative to TIPI-1. Hence its UF_{opt} will be the same or higher than the UF_{opt} of TIPI-1.

4.5 Revalidating frequency exploration range

Previous sections described the optimizations carried out in Cuttlefish to reduce the frequency exploration range when the exploration is about to start. This section describes the third and final optimization in Cuttlefish to shrink the exploration range even further after the exploration has begun (Algorithm 2, Line 23).

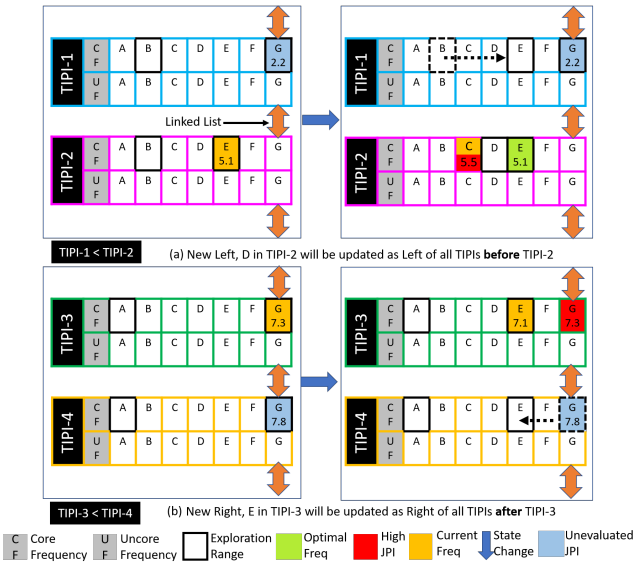


Figure 8: Revalidation of CF exploration range

Figure 8(a) shows a phase change from left to right after some T_{inv}. There are two TIPIs, TIPI-1 and TIPI-2, in the linked list, where TIPI-1 is compute-bound relative to TIPI-2. Before the phase change, MAP was set to TIPI-2 with its CF_{LB} =CF_B and CF_{RB} =CF_E. JPI for CF_E has been evaluated for TIPI-2. Now Cuttlefish has to calculate the JPI for CF_C for TIPI-2. TIPI-1's CF_{LB} =CF_B and CF_{RB} =CF_G. After some T_{inv}, Cuttlefish evaluates the JPI for CF_C and found it more than the JPI for CF_E. Hence, for TIPI-2 (Figure 8(a)), it would update the CF_{LB} =CF_D and set CF_{opt} =CF_E as CF_{LB} and CF_{RB} are consecutive frequencies (Section 4.3). However, as TIPI-1 is compute-bound relative to TIPI-2, Cuttlefish will also update the CF_{LB} =CF_E (CF_{opt} for TIPI-2). In Figure 8(b), after the phase change, Cuttlefish calculates that JPI at CF_E is better than that at CF_G for TIPI-3. Hence, it updates the CF_{RB} =CF_E for TIPI-3. However, as TIPI-4 is memory-bound relative to the TIPI-3, Cuttlefish will also update the CF_{RB} of TIPI-4 to that of TIPI-3.

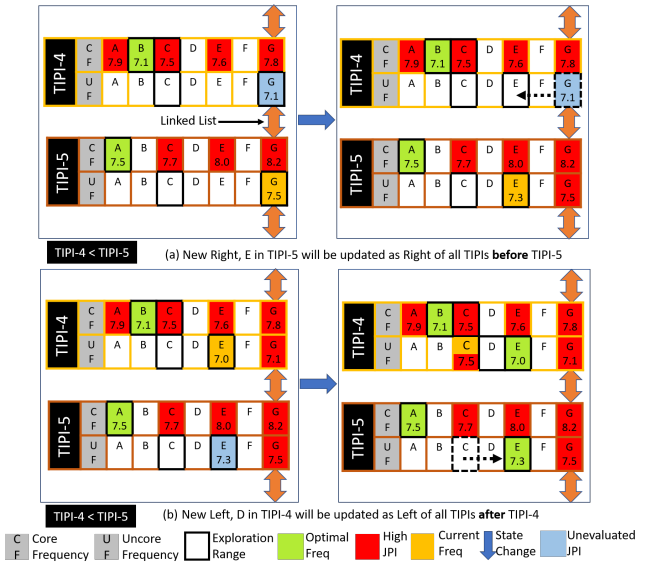


Figure 9: Revalidation of UF exploration range

Figure 9 demonstrates the same optimization discussed above, but only for revalidating the uncore exploration range. In Figure 9(a), before phase change from left to right, MAP was set to TIPI-5 with the JPI available for UF_{RB} =UF_G. Uncore was then set to UF_E, and after some T_{inv}, JPI at UF_E was found to be less than that at UF_G for TIPI-5. This changed the UF_{RB} from UF_G to UF_E at TIPI-5. However, as TIPI-4 is compute-bound relative to the TIPI-5, its UF_{RB} will also shift from UF_G to UF_E.

In Figure 9(b), before phase change from left to right, MAP was set to TIPI-4 with the JPI available for UF_{RB} =UF_E. Uncore was then set to UF_C, and after some T_{inv}, JPI at UF_C was found to be more than that at UF_E for TIPI-4. This changes the UF_{LB} from UF_C to UF_D at TIPI-4. Now, as the UF_{LB} and UF_{RB} at TIPI-4 are consecutive frequencies, UF_{opt} for TIPI-4 will be set to UF_E. As TIPI-5 is memory-bound relative to the TIPI-4, its UF_{LB} will also shift from UF_C to UF_E (TIPI-4's UF_{opt}). However, now as both UF_{LB} and UF_{RB} in TIPI-5 points to UF_E, its UF_{opt} will also be set to UF_E.

4.6 Cuttlefish in distributed computing

Cuttlefish is currently suitable for profiling a single multicore parallel program. Hence, one can also use it in MPI+X style distributed computing programs, where a single process is executed at each node for inter-node communications (MPI, UPC++, etc.), and a multithreaded library (e.g., OpenMP) is used for parallelizing intra-node computations. However, Cuttlefish cannot regulate the processor frequencies to mitigate the workload imbalance between the process [4]. Thereby, its scope is limited to the node level parallel regions (e.g., OpenMP) in regular MPI+X parallel programs that do not exhibit any load-imbalance due to overlapping computation and communication. We aim to extend Cuttlefish for achieving energy efficiency in MPI+X style hybrid parallel programs as future work.

(a) Energy savings relative to `Default`

(b) Execution time relative to `Default`

(c) EDP relative to `Default`

Figure 10: Experimental evaluation using OpenMP

5 EXPERIMENTAL EVALUATION

We provide build-time flags for the three different policies, Cuttlefish, Cuttlefish-Core, and Cuttlefish-Uncore, in the Cuttlefish library. Section 4 described the design and implementation of Cuttlefish that dynamically changes both the core and uncore frequencies. Cuttlefish-Core and Cuttlefish-Uncore are the subsets of the Cuttlefish implementation. Cuttlefish-Core only adapts the core frequencies by fixing the uncore frequency at its maximum (3.0GHz). Cuttlefish-Uncore only adjusts the uncore frequencies by setting the cores frequency at its maximum (2.3GHz). For the first TIPI-range, both Cuttlefish-Core and Cuttlefish-Uncore start the frequency exploration described in Section 4.3 in their respective default exploration ranges (1.2GHz-2.3GHz for core and 1.2GHz-3.0GHz for uncore). For subsequent TIPI ranges, they use all the runtime optimizations described in Section 4.4 and Section 4.5 but restrict these optimizations to core-only in Cuttlefish-Core and uncore-only in Cuttlefish-Uncore.

We begin our evaluation of Cuttlefish policies by measuring energy savings and slowdown for both OpenMP and HClib benchmarks. We then compare the CF_{opt} and UF_{opt} calculated by the Cuttlefish to `Default` for all frequent TIPI ranges.

5.1 Evaluation of OpenMP benchmarks

Figures 10(a), 10(b), and 10(c) compare the energy savings, execution time, and EDP, respectively, of the OpenMP benchmarks while using Cuttlefish's policies to that of the `Default`. Recall from Section 2, `Default` was executed by setting the `performance` power governor that runs each core at the highest frequency (2.3GHz). During the `Default` execution, uncore frequency is controlled by the processor based on the memory access pattern. Geomean energy-savings with Cuttlefish, Cuttlefish-Core, and Cuttlefish-Uncore are 19.6%,

3.1%, and 9.9%, respectively. Geomean loss in performance with Cuttlefish, Cuttlefish-Core, and Cuttlefish-Uncore are 3.6%, 2.5%, and 3%, respectively. Compared to the `Default`, Cuttlefish-Core required more energy in `UTS`, `SOR-irt`, `SOR-rt`, and `SOR-ws`. As these benchmarks are purely compute-bound (see TIPI-range in Table 1), Cuttlefish-Core would fix the CF_{opt} for these benchmarks at the highest frequency (2.3GHz). However, unlike Cuttlefish-Core, the `Default` optimizes the uncore frequency, thereby requiring less energy than Cuttlefish-Core. Geomean EDP savings by Cuttlefish, Cuttlefish-Core, and Cuttlefish-Uncore are 16.5%, 0.7%, and 7.2%, respectively, over the `Default`.

For memory-bound benchmarks (Heat variants, `MiniFE`, `HPCCG`, and `AMG`), energy-savings from Cuttlefish-Core and Cuttlefish-Uncore are almost similar (the difference is less than 5%). Although there is some degradation in time, the max is 6.3% (`Heat-ws`). This demonstrates that using Cuttlefish even in core-only or uncore-only mode is also quite effective for memory-bound benchmarks. As Cuttlefish adapts both core and uncore frequencies, it has the potential to save energy in both compute and memory-bound benchmarks. Energy savings are higher in the memory-bound benchmarks because Cuttlefish can adapt both the uncore and core frequencies. However, as it has to explore both these cases, the performance degradation is slightly higher (max 8.1% in `MiniFE`) than the compute-bound benchmarks (max 1.6% in `UTS`).

5.2 Evaluation of HClib benchmarks

To support our hypothesis that Cuttlefish is a programming model oblivious, we now present the evaluation of Cuttlefish policies using HClib implementations of SOR and Heat variants. We omit `MiniFE`, `HPCCG`, and `AMG` due to porting challenges. We also discarded `UTS` as this benchmark has an inbuilt work-stealing implementation, and modifying it to use HClib would alter its algorithm. HClib supports **`async-finish`** task parallelism and internally uses a work-stealing runtime for dynamic load-balancing these asynchronous tasks. The results of this experiment are shown in Figure 11. Figure 11(a), Figure 11(b), and Figure 11(c) compare the energy savings, time degradation, and EDP, respectively, for Cuttlefish policies to that of the `Default`. Comparing these results with Figure 10 shows that Cuttlefish delivers similar results in HClib and OpenMP benchmarks.

5.3 Frequency settings using Cuttlefish

Table 2 shows the percentage of distinct TIPI ranges (see Table 1) for which Cuttlefish was able to find the CF_{opt} and UF_{opt} in OpenMP benchmarks. This table also lists the CF_{opt} and UF_{opt} set by the Cuttlefish for frequently found TIPI ranges and then compare them with the `Default` settings. `AMG` was having the highest number of distinct TIPI ranges (total 60), due to which it experiences a wide variation in MAP during the execution timeline (Figure 2(b)). Cuttlefish discovered CF_{opt} and UF_{opt} in 68% and 3% of the distinct TIPIs, respectively, even for such an unstable execution. We found that the major exploration was carried out only for the frequently found TIPI ranges (total two). Still, optimal frequencies for the rest were mostly set using the runtime optimizations described in Section 4.4 and Section 4.5. As 58 of the 60 distinct TIPI ranges appear in less than 10% of the total T_{inv} durations, Cuttlefish doesn't get too many opportunities to explore UF_{opt} in `AMG` for these infrequent TIPI ranges (CF_{opt} is explored before UF_{opt}).

For the frequently found TIPI ranges in each benchmark, Cuttlefish accurately set both the CF_{opt} and UF_{opt}. These frequencies match the trend observed during motivational analysis in Section 3.2. `Heat-rt` has only two distinct TIPI ranges. Although TIPI 0.060-0.064 appears during 15% of T_{inv} durations in `Heat-rt`, their occurrences are widespread across the execution timeline. Hence, in `Heat-rt`, Cuttlefish could not set CF_{opt} and UF_{opt} for the TIPI range 0.060-0.064. Due to the performance power governor in `Default`, the `CF` was set to 2.3GHz (CF_{max}). The processor's uncore settings in `Default` were 2.2GHz and 3.0GHz (UF_{max}) for compute-bound and memory-bound benchmarks, respectively.

Benchmark	TIPI ranges (%) having CF_{opt} and UF_{opt}		Frequent TIPI Ranges	Cuttlefish		Default	
	CF_{opt}	UF_{opt}		CF_{opt}	UF_{opt}	CF	UF_{opt}
UTS	100%	100%	0.000- 0.004 (100%)	2.3 (±0%)	1.3 (±9%)	2.3	2.2
SOR-irt	100%	100%	0.024 -0.028 (100%)	2.3 (±0%)	1.2 (±0%)	2.3	2.2
SOR-rt	100%	100%	0.024 -0.028 (100%)	2.3 (±0%)	1.2 (±5%)	2.3	2.2
SOR-ws	100%	100%	0.024 -0.028 (93%)	2.3 (±0%)	1.2 (±0%)	2.3	2.2
Heat-irt	50%	25%	0.064-0.068 (88%)	1.2 (±0%)	2.2 (±0%)	2.3	3.0
Heat-rt	33%	33%	0.060-0.064 (15%)	-	-	2.3	3.0
			0.064-0.068 (84%)	1.2 (±0%)	2.2 (±0%)	2.3	3.0
Heat-ws	18%	9%	0.056-0.060 (88%)	1.3 (±9%)	2.2 (±1%)	2.3	3.0
MiniFE	44%	6%	0.112-0.116 (76%)	1.3 (±9%)	2.2 (±1%)	2.3	3.0
HPCCG	35%	6%	0.120-0.124 (76%)	1.3 (±9%)	2.2 (±1%)	2.3	3.0
AMG	68%	3%	0.144-0.148 (56%)	1.3 (±11%)	2.2 (±1%)	2.3	3.0
			0.148-0.152 (25%)	1.2 (±0%)	2.2 (±0%)	2.3	3.0

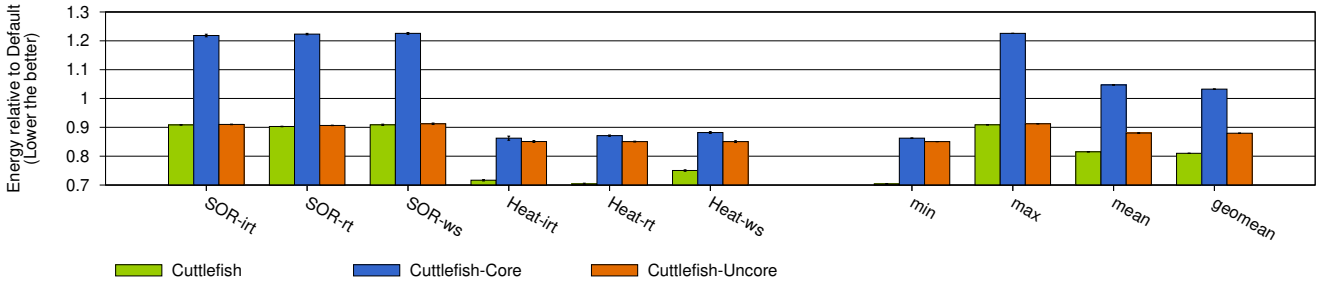
Table 2: CF_{opt} and UF_{opt} set by Cuttlefish in OpenMP benchmarks and its comparison with the `Default` settings

T_inv	Energy Savings	Slowdown
10ms	19.5%	4.1%
20ms	**19.4%**	**3.6%**
40ms	18.8%	2.9%
60ms	17.8%	2.9%

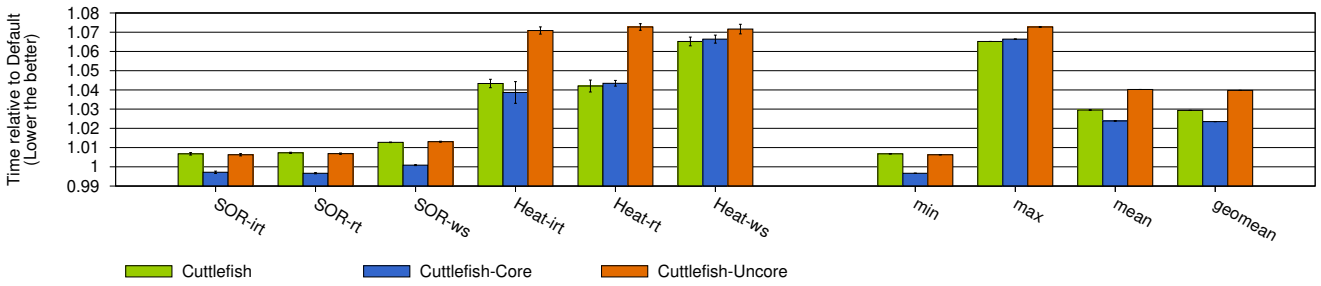
Table 3: Geomean energy savings and slowdown in OpenMP benchmarks using different values of T_{inv} in Cuttlefish

5.4 Sensitivity to T_{inv}

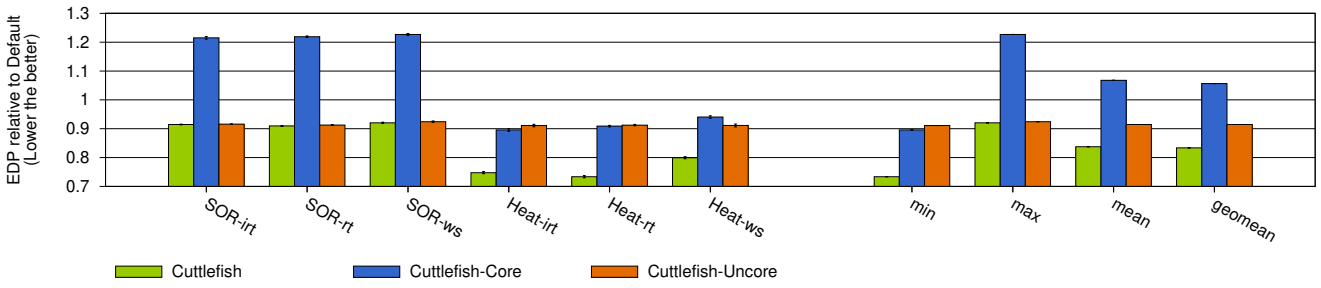
This section discusses the impact of T_{inv} on the overall energy savings and slowdown from using Cuttlefish. Table 3 compares the geomean energy savings and slowdown of OpenMP benchmarks relative to `Default` by using Cuttlefish at different values of T_{inv}. RAPL MSRs are updated every 1 ms on Intel Haswell [24]. Hence, we chose T_{inv} as 10×, 20×, 40×, and 60× of this processor default. We can observe there is a marginal decrease in energy savings and slowdown with increasing T_{inv}. Recall from Section 5.1, Cuttlefish saved more energy in memory-bound benchmarks (22%–29%) than compute-bound (8%–10.1%), but with some performance degradation (3.6%–8.1%). This is because optimal frequencies were lower for both core (1.2GHz/1.3GHz) and uncore (2.2GHz) in memory-bound benchmarks (Table 2). As Cuttlefish performs frequency exploration linearly, starting from the highest to lowest frequencies, higher T_{inv} increases the exploration time, thereby allowing benchmarks to



(a) Energy savings relative to `Default`



(b) Execution time relative to `Default`



(c) EDP relative to `Default`

Figure 11: Experimental evaluation using HClib

run longer at higher frequencies, causing a slight reduction in energy savings and slowdown. We chose T_{inv} =20ms as the default configuration of Cuttlefish, as it achieved similar energy savings as with T_{inv} =10ms, but with lesser slowdown.

5.5 Summary

The encouraging results for Cuttlefish demonstrate its efficacy for achieving energy efficiency using a combination of DVFS and UFS on modern multicore processors in a wide range of applications with minimal impact on execution time. Cuttlefish, therefore, promises a one-stop solution as it uses both these dynamically available fine-grain power controls on Intel processors to regulate power. Its two variants, Cuttlefish-Core and Cuttlefish-Uncore, provide alternatives to using both DVFS and UFS together. Overall, Cuttlefish-Uncore achieved better EDP than Cuttlefish-Core. While Cuttlefish-Uncore is suitable for both compute and memory-bound applications, Cuttlefish-Core is not ideal for compute-bound applications as it sets the uncore to the maximum frequency that is otherwise adaptable in the default settings. However, on memory-bound applications, Cuttlefish-Core was able to deliver EDP close to that of Cuttlefish-Uncore.

6 RELATED WORK

Most energy-efficient HPC research in software has focused on making efficient use of micro-architectural power control support available. DVFS is supported by both AMD and Intel processors. It has been the primary choice to control processor frequency in several studies [4, 5, 16, 17, 23, 27, 29, 39, 41, 47]. In the past, processors provided only chip-level DVFS where a frequency change affected all the cores. Most modern processors today support core-specific implementations of DVFS, allowing each core to operate at different frequencies. Intel has provided core-specific support

for DVFS from its Haswell generation processors. DDCM has also been used to control processor frequency at a core level in several studies [6, 46, 50].

The target programming model of prior work has dictated much of their design and implementation decisions. The approaches that target the Message Passing Interface (MPI) applications mainly involve mitigation of workload imbalance between the process (slack) [4, 16, 29, 41]. Other MPI-centric solutions address cases where the processor cores wait on the memory or network [5, 22, 26, 43, 48, 49]. Concurrency throttling has been widely used by adapting the thread count in OpenMP programs that are memory-constrained to reduce power consumption [12, 13, 31, 37]. Methods [52] that involve assigning optimal frequencies to OpenMP loops based on their memory access patterns have been proposed. Static analysis of loop-based parallel programs has been used to compute each loop chunk's workload and then tune the chip frequency dynamically by calculating the remaining workload [42]. Iterative parallel programs provide an opportunity to dynamically tune the frequencies of the cores according to the workload information of the tasks collected with the online profiling of the first few iterations [9]. The impact of power capping on compiler transformations has been studied by utilizing the memory access density information of OpenMP loops [51]. Fine-tuning the core frequencies based on online profiling of thief-victim relationships and the size of deques in a work-stealing runtime is another approach for programming model-specific solution [40]. There have been attempts to standardize the interfaces controlling power at different levels of the HPC system hierarchy. GEOPM [14] is one such implementation that leverages feedback from the application to identify which nodes are on the critical path and then adjusts processor power cap settings to accelerate the critical path and improve the application's time-to-solution. The current effort may easily fit as a part of the third-party software components that GEOPM and such standardized interfaces support.

UFS is a recent research topic for achieving energy efficiency, although it has been available to the user since the Intel Haswell processor generation [21]. The high efficacy of UFS has been demonstrated in terms of its energy-saving potential [45]. UFS has been used to dynamically adapt the uncore frequency based on the DRAM power usage [18]. A machine learning-based UFS model has been proposed to gather performance metrics from offline executions and predict the optimal uncore frequency [3]. The work described in [44] aims to achieve energy efficiency by using both DVFS and DRAM scaling. However, unlike Cuttlefish, this implementation needs prior data to learn some machine-dependent parameters using regression analysis. These parameters are used at runtime to calculate the optimal core and uncore frequency.

7 CONCLUSION

In this paper, a programming model oblivious C/C++ library is proposed for achieving energy efficiency that is not limited by language constraints and semantics. The model can detect the changes in the memory access patterns even for the same application across multiple language implementations to assign optimal frequencies. Using both DVFS and UFS allows additional energy savings than solutions that utilize a single power control. The effectiveness of the current approach is demonstrated for irregular-tasking as well as work-sharing pragmas. The use of both micro-kernels and real-world HPC mini-applications for evaluation helps better understand the library's efficacy for production use.

In the future, we would like to extend this framework to support hybrid MPI+X applications. With the emergence of scientific workflows in HPC systems, we want to explore the possibility of using Cuttlefish to control the power of co-running components of a workflow on a node. Graphics processing units (GPUs) are ever more prevalent in HPC today. The energy-efficiency potential in the context of GPU-offload support offered by many high-level languages is another area worth exploring.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers for their suggestions on improving the presentation of the paper. Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] Accessed 2021. The Mantevo Performance Co-design Project. https://mantevo.github.io/
[2] November 2020. TOP500. https://www.top500.org/statistics/list/
[3] Solomon Abera Bekele, M Balakrishnan, and Anshul Kumar. 2019. ML Guided Energy-Performance Trade-Off Estimation For Uncore Frequency Scaling. In *2019 Spring Simulation Conference (SpringSim)*. 1–12. https://doi.org/10.23919/SpringSim.2019.8732878
[4] Sridutt Bhalachandra, Allan Porterfield, Stephen L. Olivier, and Jan F. Prins. 2017. An Adaptive Core-Specific Runtime for Energy Efficiency. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 947–956. https://doi.org/10.1109/IPDPS.2017.114
[5] Sridutt Bhalachandra, Allan Porterfield, Stephen L. Olivier, Jan F. Prins, and Robert J. Fowler. 2017. Improving Energy Efficiency in Memory-Constrained Applications Using Core-Specific Power Control. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing (E2SC'17)*. ACM NY USA, Article 6, 8 pages. https://doi.org/10.1145/3149412.3149418
[6] Sridutt Bhalachandra, Allan Porterfield, and Jan F. Prins. 2015. Using Dynamic Duty Cycle Modulation to Improve Energy Efficiency in High Performance Computing. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 911–918. https://doi.org/10.1109/IPDPSW.2015.144
[7] J Mark Bull, Lorna A Smith, Martin D Westhead, David S Henty, and Robert A Davey. 2000. A benchmark suite for high performance Java. *Concurrency: Practice and Experience* 12, 6 (2000), 375–388. https://doi.org/10.1002/1096-9128(200005)12:6%3C375::AID-CPE480%3E3.0.CO;2-M
[8] Quan Chen, Minyi Guo, and Haibing Guan. 2014. LAWS: Locality-Aware Work-Stealing for Multi-Socket Multi-Core Architectures. In *Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14)*. ACM NY USA, 3–12. https://doi.org/10.1145/2597652.2597665
[9] Quan Chen, Long Zheng, Minyi Guo, and Zhiyi Huang. 2014. EEWA: Energy-Efficient Workload-Aware Task Scheduling in Multi-core Architectures. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*. 642–651. https://doi.org/10.1109/IPDPSW.2014.75
[10] Brandon Cook, Thorsten Kurth, Brian Austin, Samuel Williams, and Jack Deslippe. 2017. Performance variability on Xeon Phi. In *International Conference on High Performance Computing*. Springer, 419–429. https://doi.org/10.1007/978-3-319-67630-2_31
[11] Paul Stewart Crozier, Heidi K Thornquist, Robert W Numrich, Alan B Williams, Harold Carter Edwards, Eric Richard Keiter, Mahesh Rajan, James M Willenbring, Douglas W Doerfler, and Michael Allen Heroux. 2009. Improving performance via mini-applications. (2009). https://doi.org/10.2172/993908
[12] Matthew Curtis-Maury, Filip Blagojevic, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. 2008. Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes. *IEEE Transactions on Parallel and Distributed Systems* 19, 10 (2008), 1396–1410. https://doi.org/10.1109/TPDS.2007.70804

[13] Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. 2006. Online Power-Performance Adaptation of Multithreaded Programs Using Hardware Event-Based Prediction. In *Proceedings of the 20th Annual International Conference on Supercomputing* (Cairns, Queensland, Australia) *(ICS '06)*. ACM NY USA, New York, NY, USA, 157–166. https://doi.org/10.1145/1183401.1183426

[14] Jonathan Eastep, Steve Sylvester, Christopher Cantalupo, Brad Geltz, Federico Ardanaz, Asma Al-Rawi, Kelly Livingston, Fuat Keceli, Matthias Maiterth, and Siddhartha Jana. 2017. Global extensible open power manager: a vehicle for HPC community collaboration on co-designed energy management solutions. In *International Supercomputing Conference*. Springer, Cham, 394–412. https://doi.org/10.1007/978-3-319-58667-0_21

[15] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2012. Dark Silicon and the End of Multicore Scaling. *IEEE Micro* 32, 3, 122–134. https://doi.org/10.1109/MM.2012.17

[16] Vincent W. Freeh and David K. Lowenthal. 2005. Using Multiple Energy Gears in MPI Programs on a Power-Scalable Cluster. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. ACM NY USA, New York, NY, USA, 164–173. https://doi.org/10.1145/1065944.1065967

[17] R. Ge, Xizhou Feng, and K.W. Cameron. 2005. Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 34–34. https://doi.org/10.1109/SC.2005.57

[18] Neha Gholkar, Frank Mueller, and Barry Rountree. 2019. Uncore Power Scavenger: A Runtime for Uncore Power Conservation on HPC Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*. Article 27, 23 pages. https://doi.org/10.1145/3295500.3356150

[19] Max Grossman, Vivek Kumar, Nick Vrvilo, Zoran Budimlic, and Vivek Sarkar. 2017. A pluggable framework for composable HPC scheduling libraries. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 723–732. https://doi.org/10.1109/IPDPSW.2017.13

[20] Alastair Hart, Harvey Richardson, Jens Doleschal, Thomas Ilsche, Mario Bielert, and Matthew Kappel. 2014. User-level power monitoring and application performance on cray xc30 supercomputers. *Proceedings of the Cray User Group (CUG)* (2014).

[21] David L Hill, Derek Bachand, Selim Bilgin, Robert Greiner, Per Hammarlund, Thomas Huff, Steve Kulick, and Robert Safranek. 2010. The Uncore: A Modular Approach to Feeding the High Performance Cores. *Intel Technology Journal* 14, 3 (2010).

[22] Torsten Hoefler and Dmitry Moor. 2014. Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations. *Supercomput. Front. Innov.: Int. J.* 1, 2 (July 2014), 58–75. https://doi.org/10.14529/jsfi140204

[23] Chung hsing Hsu and Wu chun Feng. 2005. A Power-Aware Run-Time System for High-Performance Computing. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 1–1. https://doi.org/10.1109/SC.2005.3

[24] Intel. Accessed 2021. Intel 64 and IA-32 architectures software developer's manual. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

[25] Intel. Accessed 2021. Intel Xeon processor E5 v3 family uncore performance monitoring. https://www.intel.com/content/dam/www/public/us/en/zip/xeon-e5-v3-uncore-performance-monitoring.zip

[26] Krishna Kandalla, Emilio P. Mancini, Sayantan Sur, and Dhabaleswar K. Panda. 2010. Designing Power-Aware Collective Communication Algorithms for InfiniBand Clusters. In *2010 39th International Conference on Parallel Processing*. 218–227. https://doi.org/10.1109/ICPP.2010.78

[27] N. Kappiah, V.W. Freeh, and D.K. Lowenthal. 2005. Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 33–33. https://doi.org/10.1109/SC.2005.39

[28] Wonyoung Kim, Meeta S. Gupta, Gu-Yeon Wei, and David Brooks. 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 123–134. https://doi.org/10.1109/HPCA.2008.4658633

[29] Hideaki Kimura, Mitsuhisa Sato, Yoshihiko Hotta, Taisuke Boku, and Daisuke Takahashi. 2006. Emprical study on Reducing Energy of Parallel Programs using Slack Reclamation by DVFS in a Power-scalable High Performance Cluster. In *2006 IEEE International Conference on Cluster Computing*. 1–10. https://doi.org/10.1109/CLUSTR.2006.311839

[30] Vivek Kumar, Yili Zheng, Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. 2014. HabaneroUPC++: A Compiler-Free PGAS Library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14)*. ACM NY USA, Article 5, 10 pages. https://doi.org/10.1145/2676870.2676879

[31] Dong Li, Bronis R de Supinski, Martin Schulz, Kirk Cameron, and Dimitrios S. Nikolopoulos. 2010. Hybrid MPI/OpenMP power-aware computing. (2010), 1–12. https://doi.org/10.1109/IPDPS.2010.5470463

[32] LLNL. Accessed 2021. AMG. https://github.com/LLNL/AMG

[33] LLNL. Accessed 2021. Exascale Computing Project. https://exascale.llnl.gov/

[34] LLNL. Accessed 2021. MSR-SAFE. https://github.com/LLNL/msr-safe

[35] M.I.T. 2010. Cilk-5.4.6. http://supertech.csail.mit.edu/cilk/

[36] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. 2006. UTS: An Unbalanced Tree Search Benchmark. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing (LCPC'06)*. Springer Berlin Heidelberg, 235–250. https://doi.org/10.1007/978-3-540-72521-3_18

[37] Allan Porterfield, Rob Fowler, Sridutt Bhalachandra, and Wei Wang. 2013. OpenMP and MPI Application Energy Measurement Variation. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing (E2SC '13)*. ACM NY USA, Article 7, 8 pages. https://doi.org/10.1145/2536430.2536437

[38] Allan Porterfield, Rob Fowler, and Min Yeol Lim. 2010. *RCRTool: Design document version 0.1*. Technical Report.

[39] Karunakar Reddy Basireddy, Eduardo Weber Wachter, Bashir M. Al-Hashimi, and Geoff Merrett. 2018. Workload-Aware Runtime Energy Management for HPC Systems. In *2018 International Conference on High Performance Computing Simulation (HPCS)*. 292–299. https://doi.org/10.1109/HPCS.2018.00057

[40] Haris Ribic and Yu David Liu. 2014. Energy-Efficient Work-Stealing Language Runtimes. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM NY USA, 513–528. https://doi.org/10.1145/2541940.2541971

[41] Barry Rountree, David K. Lowenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. 2009. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM NY USA, 460–469. https://doi.org/10.1145/1542275.1542340

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

[43] Vaibhav Sundriyal and Masha Sosonkina. 2011. Per-call Energy Saving Strategies in All-to-All Communications. In *Recent Advances in the Message Passing Interface*, Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, 188–197. https://doi.org/10.1007/978-3-642-24449-0_22

[44] Vaibhav Sundriyal and Masha Sosonkina. 2016. Joint Frequency Scaling of Processor and DRAM. *The Journal of Supercomputing* 72, 4 (2016), 1549–1569. https://doi.org/10.1007/s11227-016-1680-4

[45] Vaibhav Sundriyal, Masha Sosonkina, Bryce M. Westheimer, and Mark Gordon. 2018. Comparisons of Core and Uncore Frequency Scaling Modes in Quantum Chemistry Application GAMESS. In *Proceedings of the High Performance Computing Symposium (HPC '18)*. Society for Computer Simulation International, Article 13, 11 pages.

[46] Vaibhav Sundriyal, Masha Sosonkina, and Zhao Zhang. 2014. Automatic runtime frequency-scaling system for energy savings in parallel applications. *The Journal of Supercomputing* 68, 2 (2014), 777–797. https://doi.org/10.1007/s11227-013-1062-0

[47] Ananta Tiwari, Michael Laurenzano, Joshua Peraza, Laura Carrington, and Allan Snavely. 2012. Green Queue: Customized Large-Scale Clock Frequency Scaling. In *2012 Second International Conference on Cloud and Green Computing*. 260–267. https://doi.org/10.1109/CGC.2012.62

[48] Akshay Venkatesh, Abhinav Vishnu, Khaled Hamidouche, Nathan Tallent, Dhabaleswar (DK) Panda, Darren Kerbyson, and Adolfy Hoisie. 2015. A Case for Application-Oblivious Energy-Efficient MPI Runtime. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM NY USA, Article 29, 12 pages. https://doi.org/10.1145/2807591.2807658

[49] Abhinav Vishnu, Shuaiwen Song, Andres Marquez, Kevin Barker, Darren Kerbyson, Kirk Cameron, and Pavan Balaji. 2010. Designing Energy Efficient Communication Runtime Systems for Data Centric Programming Models. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications Int'l Conference on Cyber, Physical and Social Computing*. 229–236. https://doi.org/10.1109/GreenCom-CPSCom.2010.133

[50] Bo Wang, Dirk Schmidl, and Matthias S. Müller. 2015. Evaluating the Energy Consumption of OpenMP Applications on Haswell Processors. In *OpenMP: Heterogenous Execution and Data Movements*, Christian Terboven, Bronis R. de Supinski, Pablo Reble, Barbara M. Chapman, and Matthias S. Müller (Eds.). Springer International Publishing, Cham, 233–246. https://doi.org/10.1007/978-3-319-24595-9_17

[51] Wei Wang. 2016. *Performance, Power, and Energy Tuning Using Hardware and Software Techniques For Modern Parallel Architectures*. Ph.D. Dissertation. University of Delaware.

[52] Wei Wang, Allan Porterfield, John Cavazos, and Sridutt Bhalachandra. 2015. Using Per-Loop CPU Clock Modulation for Energy Efficiency in OpenMP Applications. In *2015 44th International Conference on Parallel Processing*. 629–638. https://doi.org/10.1109/ICPP.2015.72

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We have targeted six widely-used benchmarks for our experimental evaluations. These benchmarks use OpenMP pragmas, but we also ported some of them to use the async–finish programming model supported by the HClib work-stealing library. For two of these benchmarks, Heat and SOR, we created three variants (both for OpenMP and HClib) that differ in execution DAG and concurrency decomposition technique. The other benchmarks include UTS, MiniFE, HPCCG and AMG. We ran these benchmarks on an Intel Xeon Haswell E5-2650 20-core processor with a total of 94GB of RAM. The operating system (OS) was Ubuntu 16.04.7 LTS. We used OpenMP supported by the Clang compiler version 3.8.0 and used the same compiler for compiling HClib benchmarks. We used -O3 flag with the compiler.

Author-Created or Modified Artifacts:

```
Persistent ID: https://doi.org/10.5281/zenodo.5167629
Artifact name: Cuttlefish: Library for Achieving
↪  Energy Efficiency in Multicore Parallel Programs
↪  (Artifact)
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: CPU: Intel Xeon Haswell E5-2650 20-core processor, Filesystem: Ext4

Operating systems and versions: Ubuntu 16.04.7 LTS running on linux kernel 4.4.0-200-generic x86_64 (64 bit)

Compilers and versions: Clang v3.8.0

Applications and versions: UTS (v1.1), SOR-ws (scimark 4), Heat-rt (Cilkbench commit ID ab94183), MiniFE (Mantevo commit ID c043cd1), HPCCG (Mantevo commit ID 80dd2f1), AMG (LLNL commit ID 3ada8a1)

Libraries and versions: msr-safe (commit ID e7f9423), numactl, likwid v5.0.1, mpich v3.3

URL to output from scripts that gathers execution environment information.

`https://doi.org/10.5281/zenodo.5167629`