Towards Fine-grained Dynamic Tuning of HPC Applications on Modern Multi-Core Architectures

Mohammed Sourouri
Norwegian University of Science and Technology (NTNU)
mohammed.sourouri@ntnu.no

Espen Birger Raknes
Aker BP
espen.birger.raknes@akerbp.com

Nico Reissmann
Norwegian University of Science and Technology (NTNU)
reissman@idi.ntnu.no

Johannes Langguth
Simula Research Laboraty
langguth@simula.no

Daniel Hackenberg
Technische Universität Dresden
daniel.hackenberg@tu-dresden.de

Robert Schöne
Technische Universität Dresden
robert.schoene@tu-dresden.de

Per Gunnar Kjeldsberg
Norwegian University of Science and Technology (NTNU)
pgk@ntnu.no

ABSTRACT

There is a consensus that exascale systems should operate within a power envelope of 20MW. Consequently, energy conservation is still considered as the most crucial constraint if such systems are to be realized.

So far, most research on this topic focused on strategies such as power capping and dynamic power management. Although these approaches can reduce power consumption, we believe that they might not be sufficient to reach the exascale energy-efficiency goals. Hence, we aim to adopt techniques from embedded systems, where energy-efficiency has always been the fundamental objective.

A successful energy-saving technique used in embedded systems is to integrate fine-grained autotuning with dynamic voltage and frequency scaling. In this paper, we apply a similar technique to a real-world HPC application. Our experimental results on a HPC cluster indicate that such an approach saves up to 20% of energy compared to the baseline configuration, with negligible performance loss.

KEYWORDS

Energy-efficiency; high performance computing; dynamic voltage and frequency scaling; autotuning; dynamic tuning

1 INTRODUCTION

In order to improve energy-efficiency, hardware vendors increasingly employ techniques from embedded systems. Features such as decoupled frequency domains, dynamic voltage and frequency scaling (DVFS), ultra-low power states, and the idea of fused CPUs and FPGAs, indicate a trend that HPC and embedded systems are converging. For example, the latest generations of Intel Xeon CPUs not only improve performance by adopting additional cores or wider registers, but also incorporate a series of user-controllable switches [16, 19, 28]. Access to such switches enables users to influence the energy characteristics of the CPU. In the Haswell-EP and subsequent architectures, programmers can manipulate both individual core frequencies and the *uncore* frequency, i.e. the frequency of the ring connecting the cores to the memory controller [16]. These user-controllable hardware switches are an effective source for conserving energy in embedded systems, and their increasing influx into HPC begs the question of how they can be used to improve energy-efficiency.

To answer this question, we look to embedded systems where fine-grained autotuning at code-level, tightly integrated with user-controllable switches has been extensively studied. One example are System-Scenarios [10], where highly tuned system configurations are created in order to map the underlying hardware architecture to the application behavior. Other examples include self-aware compute systems [18] and systems based on the observe-decide-act principle [29] where a system-monitor picks the best hardware and software configuration for a given behavior.

Regardless of the approach, a common denominator is the task of balancing multiple constraints such as performance and energy. While some of these approaches proved to be successful in embedded systems or other domains, very few were applied to typical HPC applications. Hence, there is no guarantee that such an approach works in a performance-first environment, as the fundamental objectives are different. Embedded systems typically work under *deadline* constraints required by real-time processing, while in HPC the *time-to-solution* has to be minimized.

In this paper, we combine fine-grained autotuning with user-controllable hardware switches and threads, and apply this technique to a real-world HPC application on a single compute node. The application solves the elastic wave equation repeatedly in order to create a seismic image of the subsurface of the earth. It is predominantly memory-bound, which makes it a good candidate for energy savings on CPUs with decoupled frequency domains, where the core frequency can be scaled down to conserve energy.

To evaluate our approach, we tune our application for the following three objectives: energy, energy delay product (EDP) and energy delay product squared (ED2P). Our overall results show energy savings of up to 20%, with an increase of only 3% in runtime compared to the reference implementation.

Thus, our primary contributions in this paper are:

- We implement and test an energy-conserving methodology inspired by embedded systems, which combines the strengths of fine-grained autotuning with user-controllable hardware switches in a real-world HPC application on a single compute node.
- We present experimental results and compare a reference implementation with statically and dynamically tuned versions.
- We evaluate the viability of our approach with respect to energy-performance trade-offs.
- We present energy results of a test application with and without AVX2 vectorization enabled.

The work described in this paper is conducted in the context of the READEX project, which aims at exploiting dynamic application behavior at runtime for energy-efficient exascale computing.

The remainder of the paper is organized as follows: Section 2 provides background on autotuners and user-controllable switches, while related work is surveyed in Section 3. We present our target application in Section 4, followed by an explanation of our dynamic tuning approach in Section 5. The details of our experimental setup are presented in Section 6, while results are discussed in Section 7. The paper concludes with Section 8.

2 BACKGROUND

This section gives a brief overview of state-of-the-art autotuners, user-controllable hardware switches found on some modern x86 CPUs, and the different energy-efficiency metrics.

2.1 State-of-the-art Autotuning

The primary goal of autotuners is to optimize applications by finding the best combination of tuning parameters for a given system. Typical examples of tuning parameters are compiler flags, environmental settings, and application specific parameters. In the context of autotuning, the life cycle of an application is generally split in design-time and runtime. The actual tuning process either takes place at design-time, i.e. before the application is executed, or at runtime, i.e. during its execution. This is referred to *offline* and *online* tuning, respectively. Some autotuners support both approaches.

Furthermore, autotuners can also offer support for *static* or *dynamic tuning*. In the former, the best combination of tuning parameters is determined for an entire application, while the latter permits tuning parameter changes at runtime. Typically, this requires to decompose an application into smaller regions, which enables fine-grained tuning.

A key assumption of autotuners is that the time and energy spent on tuning is amortized by the production runs. Thus, long-running scientific applications, such as weather simulations or seismic modeling, are ideal target applications for autotuners. A survey of autotuners is presented in Section 3.

	Core frequency (GHz)	
# cores	**Non-AVX2**	**AVX2**
Cores 1-2	3.3	3.1
Cores 3	3.1	2.9
Cores 4	3.0	2.8
Cores 5+	2.9	2.8
Base frequency	2.5	2.1
Frequency stepping [MHz]	100	100

Table 1: Frequency distribution of individual cores for the Intel Xeon E5 2680v3 CPU. Core frequencies listed under the non-AVX2 column indicate frequencies for workloads not optimized for AVX2, while frequencies listed under the AVX2 column indicate frequencies for AVX2 optimized workloads. Numbers from [20].

2.2 User-controllable Hardware Switches

Dynamic Voltage and Frequency Scaling (DVFS) is a commonly used approach to conserve energy. In DVFS, energy savings are realized by lowering the clock frequency, which automatically results in a lowering of the supply voltage.

Recent CPUs based on the Intel's Haswell, Broadwell, and Skylake architecture provide user-access to uncore frequency scaling (UFS) in addition to conventional core frequency scaling. Moreover, these CPUs permit to change the frequency for individual cores, providing a fine level of granularity. For Haswell CPUs, the transition latency for each core is 20μs. However, the actual switching time can be up to 500μs in practice, due to the presence of a switching window. Our measurements show that the core transition latency is uniformly distributed between 100-600μs, confirming the findings from [16]. We found the transition latency of UFS to be close to 20μs, which is identical to the numbers presented in [12].

The CPU may automatically increase the core and uncore frequencies beyond its base frequency by monitoring heuristics based on temperature, workload, and the number of idle cores. This is referred to as *turbo boost* [17], and the frequencies can be increased up to the *max turbo frequency*. However, AVX2 vectorized code uses the AVX2 units that consume more power, which would exceed the Thermal Design Power (TDP) if the CPU were running at its base frequency. In such a scenario, requests for higher frequencies than what can be guaranteed are ignored by the CPU. Table 1 shows the different turbo frequencies with and without AVX2 for the CPU used in our experiments. For the sake of reproducibility and correspondence between user-selected core and uncore frequencies, we only operate within the boundaries of guaranteed frequencies.

2.3 Reporting Energy-efficiency

The appropriate metric for reporting energy efficiency in HPC is a much-debated topic. Some researchers prefer to report energy (Joules), while others prefer power (Watts). However, both are static metrics that do not capture the energy-performance tradeoff. A widely used metric to assess both power and performance, is the Energy Delay Product (ED^n) [11]. This metric relates the energy product, E, with circuit delay, D, using the positive integer, n. In simple terms, a lower EDP value indicates that power is more efficiently converted into performance. The factor n indicates the relative importance of performance. It is commonly set to either 1, 2 or 3. A higher value of n places more importance on performance.

3 RELATED WORK

Energy-efficiency is a diverse field of research, ranging from power-reducing techniques for large data centers to analytical energy models for scratchpad memory. We limit the discussion on autotuners or DVFS tuning in HPC systems, since our work focuses on the energy-performance tradeoff applied to real-world HPC applications.

Autotuners. Autotuning is applied extensively in compute intensive applications and libraries [40], but is also used in other application domains [3]. We present the most recent and notable autotuning frameworks and libraries.

Active Harmony [37] is an autotuning framework that provides a domain-specific language (DSL) for tuning applications. It performs both offline and online tuning [39], but requires modifications to the source code. It is not clear whether DVFS-tuning is supported.

The Periscope Tuning Framework (PTF) [34] is a static autotuner that supports user plugins to perform efficient tuning. It provides a plugin to perform DVFS, and dynamic tuning is planned to be introduced in the near future. The MATE [26] and ELASTIC [24] autotuners are the only surveyed frameworks that perform dynamic tuning, albeit limited to MPI processes. This is a more coarse grained approach compared to what we propose.

Several multi-objective autotuners were proposed [1, 14, 21]. In these systems, the user specifies multiple tuning objectives, such as performance, energy, resource utilization, or time-to-solution. To the best of our knowledge, none of the currently available autotuners supports multi-objective dynamic tuning using DVFS on modern HPC clusters.

DVFS in HPC. Numerous studies investigated the use of DVFS in an HPC context. Some focused on the different energy-performance metrics [7], while others provided an analytical framework for power-performance tradeoffs [4]. Many studies also explored the prospect of power/energy profiling [9] and measurement[31]. The following studies applied DVFS to one or more HPC applications on large clusters.

Freeh et al. [8] studied the energy-time tradeoff in HPC by using applications from the NAS parallel benchmark suite on ten nodes consisting of single-core CPUs. They found that an increase of 3% in runtime could reduce the CPU's energy consumption by 20% on a single node. However, it is not clear how this result would translate to modern multi-core CPUs.

Li et al. [22, 23] developed a runtime library to exploit regions where Dynamic Concurrency Throttling or DVFS may be beneficial. A selection of MPI+OpenMP applications from the NAS parallel benchmark suite as well as two AMG based proxy applications from the ASC Sequoia benchmark suite were chosen to evaluate their approach. Other runtime libraries, such as the one by Cicotti et al. [5], were also evaluated on proxy applications for energy efficiency. In contrast to these approaches, we use dynamic tuning of core and uncore frequencies, as well as threads to find the best energy-performance tradeoff in a real-world HPC application. In addition, our energy monitoring is conducted at a higher resolution. This provides more accurate results and gives further insight into the energy cost of HPC applications.

4 SEISMIC WAVE PROPAGATION

This section describes the computational building blocks governing our application as well as key implementation details.

4.1 The Elastodynamic Wave Equation

The elastodynamic wave equation (1) and its source-term, f_i, constitutes an important backbone in many computationally complex applications, such as advanced 3D imaging and inversion of the earth's subsurface [27]. If we consider an isotropic elastic medium (2), $\Omega \subset \mathbb{R}^3$, parameterized by the density, ρ, and the Lamé parameters λ and μ, then for any $(\mathbf{x}, t) \in (\Omega, T)$, where T is the time interval, an elastic wave propagating in Ω is governed by the following elastodynamic wave equation

$$\begin{cases} \rho \dot{v}_x = \partial_x \sigma_{xx} + \partial_y \sigma_{xy} + \partial_z \sigma_{xz} + f_x \\ \rho \dot{v}_y = \partial_x \sigma_{xy} + \partial_y \sigma_{yy} + \partial_z \sigma_{yz} + f_y \\ \rho \dot{v}_z = \partial_x \sigma_{xz} + \partial_y \sigma_{yz} + \partial_z \sigma_{zz} + f_z \end{cases} \tag{1}$$

$$\begin{cases} \dot{\sigma}_{xx} = (\lambda + 2\mu)\partial_x v_x + \lambda(\partial_y v_y + \partial_z v_z) \\ \dot{\sigma}_{yy} = (\lambda + 2\mu)\partial_y v_y + \lambda(\partial_x v_x + \partial_z v_z) \\ \dot{\sigma}_{zz} = (\lambda + 2\mu)\partial_z v_z + \lambda(\partial_x v_x + \partial_y v_y) \\ \dot{\sigma}_{yz} = \mu(\partial_y v_z + \partial_z v_y) \\ \dot{\sigma}_{xz} = \mu(\partial_x v_z + \partial_z v_x) \\ \dot{\sigma}_{xy} = \mu(\partial_x v_y + \partial_y v_x) \end{cases} \tag{2}$$

A much-used numerical framework used for computing elastic waves is the staggered-grid explicit finite-difference method on a 3D Cartesian grid. Unlike others [6, 13], we employ a standard eight-order 16-point stencil for spatial approximations and a second-order stencil for temporal approximations.

4.2 Multi-core Parallelization Using OpenMP

The main computational flow of the application can be divided into two parts, as shown in Listing 1. For brevity, loops responsible for initializing components, such as stress and velocity, are not shown. However, first touch [38] was employed in these loops to minimize performance problems associated with Non-Uniform Memory Access.

The differentiators, as shown in (1), must be computed before the computation of the velocity (v_i) and the stress field (σ_{ij}) as displayed in Listing 1. The kernels within a single iteration have data dependencies, as well as a loop-carried dependency between

```
for (int it = 0; it < Nt; it++) {
  for (const auto& coordinate: coordinates) {
    compute_velocity( v_i );
    compute_stress( σ_ij );
}}
```

Listing 1: Pseudo code displaying the main computational part of the application along the Cartesian coordinates.

```
#pragma omp parallel for num_threads(nthreads)
for (int k = 0; k < nz_ghost; k++) {
  for (int j = 0; j < ny_ghost; j++) {
    for (int i = 0; i < nx_ghost-1; i++) {
      vx[k][j][i] += dt * (2.0f / (rho[k][j][i]
             + rho[k][j][i+1])) * (del1[k][j][i]
             + del2[k][j][i] + del3[k][j][i]);
}}}
```

Listing 2: Source code for computing the velocity in x direction. The same computation is repeated for computations along the other directions.

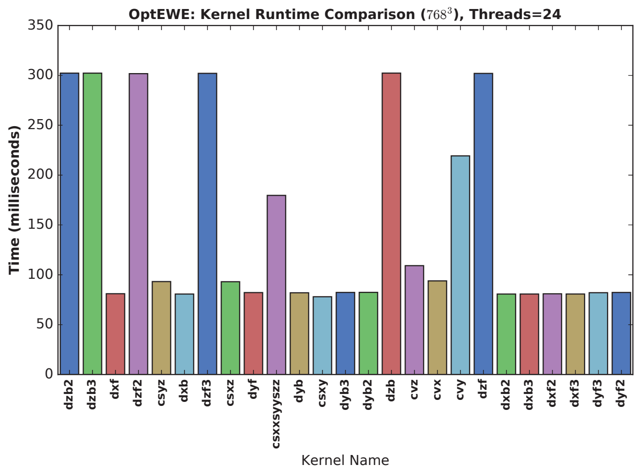


Figure 1: The runtime distribution for each kernel for input size 768^3 involving all 24 threads.

individual iterations. In total, 25 different kernels are computed per iteration. Listing 2 displays the source-code for computing the velocity in x direction. All kernels resemble typical stencil computations, making them amenable to shared-memory parallelization using OpenMP [35, 36]. We apply the roofline model [41] to better understand the performance bounds for the different kernels. This requires the computation of the operational intensity (OI) by dividing the number of FLOPS by the number of bytes. The OI for the differentiator kernels is 0.167, while it is 0.250 for the velocity and stress fields. As three of the stress field kernels iterate over the same memory region, they are fused in order to improve performance. The respective OI is 0.286.

Our intuition for tuning this application is to decrease core frequency and increase uncore frequency for kernels with low OI, and increase core frequency and decrease uncore frequency for kernels with high OI. The challenge is to find the best tradeoff point between energy efficiency and performance degradation. This requires careful tuning.

We chose three different input sizes for our wave propagation application: 512^3 (small), 768^3 (medium), and 1024^3 (large). This permits to highlight the energy-performance ratio as a function of different input sizes. The medium size input was chosen due to its poor memory alignment, which resulted in worse than expected performance for AVX2 vectorized code. All computations are executed in single-precision as this is sufficient.

Figure 1 shows the runtimes of all kernels for the reference implementation when using 24 cores and the medium input size. The slowest kernels compute along the z direction due to unfavorable memory accesses, leading to large strides between loop iterations.

5 FINE-GRAINED DYNAMIC AUTOTUNING

Figure 1 does not show which system configuration provides the best performance for the individual kernels, nor does it show the best system configuration for a given tuning objective. In order to make full use of the capabilities provided by DVFS, we need to tune every kernel individually. Current static autotuners do not provide us with that level of granularity, and even if they did, fined-grained autotuning alone is not enough. A runtime system that can apply the best-found system configuration for each kernel is needed in addition. To the best of our knowledge, there currently exists no such tool chain with a full end-to-end pipeline.

To start addressing this issue, we have developed a custom tuning setup that lets us tune each individual kernel. Our approach consists mainly of using C++ preprocessor directives, which encapsulate each kernel, allowing it to be tuned individually. In addition to enabling hardware tuning inside each directive, we simultaneously record the runtime and the timestamp of each kernel. The timestamp is later used as an offset into an FPGA based energy measurement infrastructure, which is described in more details in Section 6.

The use of C++ preprocessor directives means that we avoid changing the actual code base, which is important since the application presented in this work is in daily production. It also lets us conveniently access the different code versions by passing the appropriate arguments at compile time.

For the actual tuning, we use a small bash script to traverse over the required tuning vectors. The tuning results are then stored to disk. Once the tuning has been completed, the resulting dataset is fed to a Python post-processing tool, which acts as a recommendation system. The task of the Python tool is to automatically find the best combination of threads, core and uncore frequency for each individual kernel. In addition, it also computes the energy-performance trade-offs compared to other versions of the code.

Based on the user-selected tuning objective, the Python tool generates the appropriate compile flags including the necessary preprocessor directives. The code is then recompiled and the end result is the *dynamically tuned* version of the code. The overall workflow is depicted in Figure 2. Listing 3 displays the content inside each individual preprocessor directive.

For experimental evaluation purposes, we created three versions of the code which we refer to as: *reference*, *static* and *dynamic*. The former version is simply the default production version of the code, i.e. **no** autotuning is applied. When the reference implementation

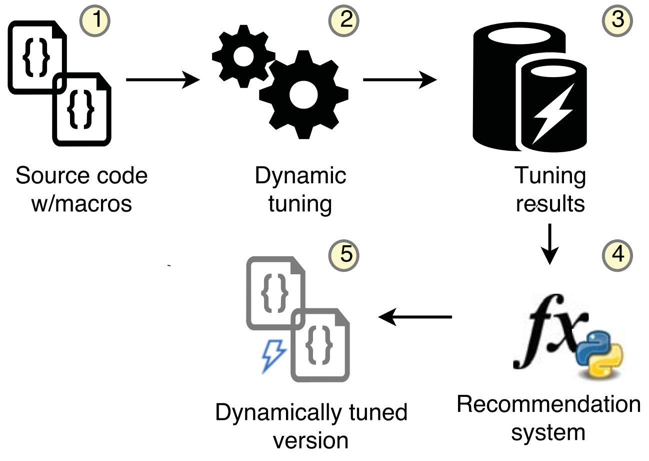


Figure 2: The workflow of our current approach.

is executed using all 24 cores of the test system, the cores run at 2.5 GHz while the uncore frequency is 3.0 GHz.

Static refers to the statically tuned version of the code. For the sake of completeness, static tuning was performed using an exhaustive search algorithm for the following three tuning vectors: core frequency, uncore frequency and the number of OpenMP threads. In the remainder of the paper, we will refer to different system configurations using the following notation: core | uncore | thread. Thus, 1.7|2.2|24, corresponds to a system configuration where the core frequency is 1.7 GHz, uncore frequency is 2.2 GHz and the number of threads is 24.

We also searched for better compiler flags than the flags used by the reference version, but this search showed that the best compiler flags, `-O2 -qopenmp -ipo`, were already being used by the reference implementation.

In *dynamic*, i.e. the dynamically tuned version of the code, each of the 25 compute kernels is tuned individually for the tuning parameters mentioned above. Like static tuning, an exhaustive search algorithm is used to find the best system configuration for each kernel. The choice of the search algorithm is motivated by the fact that it is easy to grasp. A better search algorithm would e.g. rely on a domain-specific energy model such as the one described in [2] that can predict the best combination of core and uncore frequency for each kernel. Such a prediction would dramatically reduce the search space to only a handful alternatives, which may be found through testing or careful analysis.

Once the tuning process is completed, the recommendation system will pick the best system configuration for a given tuning objective function. As an example, let us consider the case where the tuning objective is ED2P. The recommendation system will then pick the system configuration for each kernel with the lowest ED2P number. The same procedure is used for the other tuning objectives.

Moreover, in the dynamically tuned version, the best DVFS setting found is applied inside the main compute loop and for each kernel, in contrast to the statically tuned version. When using static tuning, the best-found DVFS settings are applied only once, which

Kernel	**Core [GHz]**	**Uncore [GHz]**	**# Threads**
dzb	1.5	2.2	24
cvz	1.5	2.5	24
dyf3	1.3	1.8	24
cvy	2.5	2.1	24
dxf	1.2	2.1	24
cvx	1.4	1.9	23
csxxsyyszz	1.8	1.7	24

Table 2: The best system configurations found for the various kernels when the tuning objective is energy. Due to space considerations, only system configurations for a selection of compute kernels in x, y and z directions from Figure 1 are shown.

is immediately after the application start. The total dynamic DVFS overhead can be computed by multiplying the number of iterations with the number of kernels and with the overhead for core and uncore frequency scaling.

Table 2 shows an example of the best found system configurations used by the dynamically tuned version for the medium problem size when the tuning objective function is energy. The best static configuration was 1.7|2.3|24.

The same tuning procedure was repeated for the vectorized version of the code where AVX2 instructions are applied. The procedure is the same, except for the fact that the vectorized version also uses the following compiler flags: `-O3 -axCORE-AVX2`. In the vectorized version the `-O2` flag is discarded. Unlike the non-vectorized version, the vectorized reference implementation runs at 2.1 and 2.8 GHz for core and uncore, respectively.

6 EXPERIMENTAL PLATFORM

For all our experiments, we use the Taurus [33] cluster which is located at Technische Universität Dresden (TUD), Germany. A large partition, consisting of 1456 homogeneous compute nodes, is equipped with Haswell-EP CPUs. Table 3 provides a summary of a Taurus compute node. For core and uncore clock frequencies, we refer to Table 1.

Cluster	**Taurus**
Processor	Intel Xeon E5-2680v3
Architecture	Haswell-EP
Cores	12
Sockets	2
Base core clock freq. [GHz]	2.5
L3 cache/chip [MB]	30
Memory size [GB]	64
Peak DP [GFLOPs]	960
Peak BW [GB/s]	136
STREAM [GB/s]	116
Compiler	icpc 16.2.181
Thread layout	compact,fine

Table 3: Architectural overview of a Taurus compute node.

```
#ifdef CVX_NEMI
set_cpu_core_freq(core_freq_setting);
set_uncore_freq(uncore_freq_setting);

hdeem_tstart.time_since_epoch();

cvx_time -= std::chrono::high_resolution_clock::now();
compute_velocity( v_i );
cvx_time += std::chrono::high_resolution_clock::now();

hdeem_tstop.time_since_epoch();
#endif CVX_NEMI
```

Listing 3: Example code demonstrating how the energy usage for individual code regions were captured using HDEEM. The same excerpt also shows how core and uncore frequency were changed.

Because each compute node is directly water cooled, no electrical power is used for cooling. This makes it easier to report precise energy results for an entire compute node. Furthermore, each compute node is instrumented using the High Definition Energy Efficiency Monitoring (HDEEM) infrastructure [15]. By equipping each compute node with an FPGA, the energy usage of various components such as the CPUs, memory modules and the compute node itself can be monitored without interrupting the CPU. The benefits of this approach are many, but for our purpose, the two most important features are high accuracy (a maximum error rate of 2% is reported) and the ability to profile individual code regions with high spatial and temporal granularity (1 kSa/s).

Another important feature of the Taurus cluster is the ability to use DVFS in the Haswell-EP partition. We have used the low-level `x86_adapt` [30] library for this purpose. Listing 3 displays how `x86_adapt` and HDEEM was used for DVFS and energy instrumentation in our codes.

7 EXPERIMENTAL RESULTS

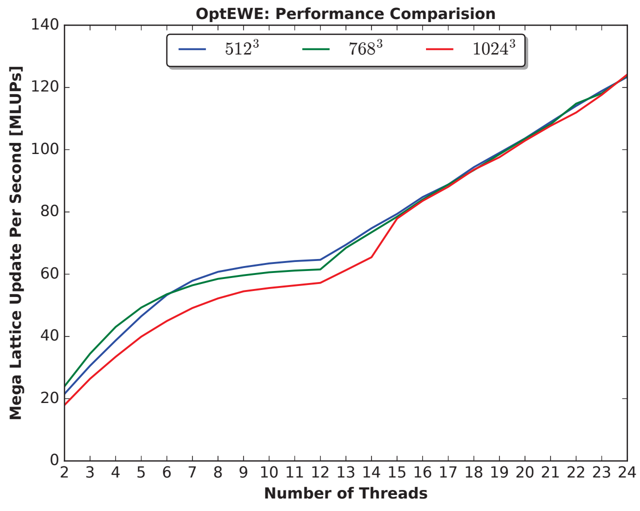
In this section, we present experimental results for the following three tuning objective functions: energy, EDP and ED2P. We do not tune for runtime because finding a system configuration that is better than the reference implementation for our application is not possible. In other words, the highest performance is reached when the application is executed with the highest core and uncore frequency, in addition to the maximum number of threads.

While a full production run typically consists of 1000 or more iterations, it is not necessary to run more than five iterations to evaluate the tuning objective functions because the iterations have essentially the same runtime. Since the number of iterations required for the production runs increases with problem size, the cost of tuning becomes smaller in comparison. The results presented in this section are based on five iterations. They measure the energy consumption of the entire compute node, including CPU, DRAM, storage, etc.

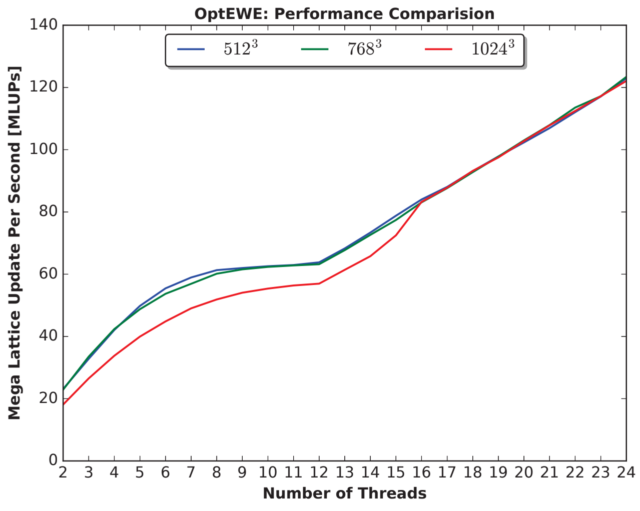
7.1 Reference Implementation Performance

We start by presenting the performance of the reference implementation to establish a baseline, before presenting the result of the statically and dynamically tuned implementations. We use Mega Lattice Site Updates Per Second (MLUPs) as our performance metric of choice. The number of MLUPs can be computed by multiplying the number of grid points with the number of iterations divided by the runtime in seconds.

Figures 3(a) and 3(b) show the performance scaling of the reference implementation as a function of the number of OpenMP threads, with and without vectorization. As previously mentioned, our application is inherently memory-bound, which explains the increase in performance when the 12 core (single socket) boundary is crossed. The main benefit of an additional socket lies in the additional memory channels it adds to the system, which result in



(a) Baseline performance with AVX2 disabled



(b) Baseline performance with AVX2 enabled

Figure 3: The sustained performance results measured in MLUPs for the three model sizes with and without vectorization.

higher memory bandwidth. We also observe that for all problem sizes, the highest performance is achieved when all 24 cores are used, and it appears that the memory bandwidth is never saturated. This behavior is explained by the choice of thread affinity. We use compact rather than scatter. A similar curve was observed when running the well-known STREAM Triad benchmark [25].

Recall from Section 5 that the reference implementation operates at 2.5 and 3.0 GHz for the non-vectorized version and 2.1 and 2.8 GHz for the vectorized version. We observe that despite running at a lower frequency, all vectorized implementations deliver a similar result as the non-vectorized versions.

7.2 Tuning for Energy

Figure 4(a) displays the results for the different problem sizes when tuning for energy. Regardless of the approach or implementation (vectorization/no-vectorization), both static and dynamic tuning can save a considerable amount of energy compared to a highly-optimized reference implementation. In Figure 4(a) (going from the smallest to the largest problem size), the dynamically tuned version consumes 20.35%, 24.28% and 21.28% less energy, with an increase in runtime of 10.85%, 12.91% and 10.11%. For the vectorized version, the dynamic version consumes 13.19%, 14.39% and 21.28% less energy, with an increase of 9.16%, 9.32% and 18.49% in runtime compared to the reference implementation. Moreover, in all cases, the statically tuned version consumes, on average, 1.97% more energy, with an increase of 3.56% in runtime compared to the dynamic version.

The reason why the dynamically tuned version can only beat the statically tuned version with a small margin is simply due to the high DVFS overhead. As previously mentioned, the advantage of the statically tuned version is that DVFS is called only once, *before* the main compute loop is executed. In the dynamically tuned version however, the DVFS functions are placed inside the compute loop and are called once for each kernel and in every iteration. Since we know the DVFS overhead, we can exclude it from the runtime to create an ideal scenario where the cost of DVFS is virtually free. This version constitutes an upper bound on the potential energy savings. By comparing the dynamically tuned version with the ideal scenario, we observe that the DVFS overhead amounts to 10-50%, depending on the kernel runtime and the position of switches within the DVFS switching window. In practice, this means that for the current Haswell-EP microarchitecture, performing DVFS for kernels with a short runtime may not be worthwhile. We consider the task of identifying exactly which kernels to tune as future work and expect that this will increase the benefit of dynamic tuning.

7.3 Tuning for EDP

Figure 4(b) displays our EDP values for the various implementations and problem sizes. Compared to the EDP values of the non-vectorized reference implementation, the dynamically tuned version reduces the EDP values by 16.11%, 18.59%, and 16.78% when the model size is 512^3, 768^3 and 1024^3, respectively. For the vectorized implementation, the reduction in EDP is (going from small model size to large) 9.24%, 9.98%, and 3.09%. On average, the dynamically tuned version reduces the EDP by 4.23% for the non-vectorized implementations and 4.45% for the vectorized implementations, compared to the statically tuned version.

We observe that for the largest model size, the best static tuning configuration is the same as the reference implementation if the implementation is vectorized. This behavior is similar to the one observed in Figure 4(a) for the same problem size. In general, the spread among the system configurations for the vectorized implementations is smaller compared to the non-vectorized implementations. This suggests that running kernels with suboptimal system configurations for the vectorized implementation is costlier than for the non-vectorized versions. The reason for this lies in the fact that AVX2 operations tend to be more power hungry than conventional instructions. However, because the dynamically tuned version runs all kernels with the best system configuration, it reduces such energy spills, something the statically tuned version is not capable of.

7.4 Tuning for ED2P

ED2P places more emphasis on performance than the EDP and energy objectives. Figure 4(c) shows the ED2P values for the various implementations and problem sizes. For the non-vectorized implementations, our ED2P values continue the same trend first observed for the EDP figures, except that the difference between the dynamically tuned versions and the statically tuned versions are now larger. The explanation for this is simply because the best-found static system configuration is very close or identical to the reference implementation. Although the same situation applies to the dynamically tuned versions too, there are some kernels that favor a different system configuration than the default.

The reduction in ED2P is 13.51%, 15.40%, and 14.00%, when comparing the dynamically tuned version to the reference implementation. Moreover, the dynamically tuned version reduces the ED2P values on average by 6.42% compared to the statically tuned version.

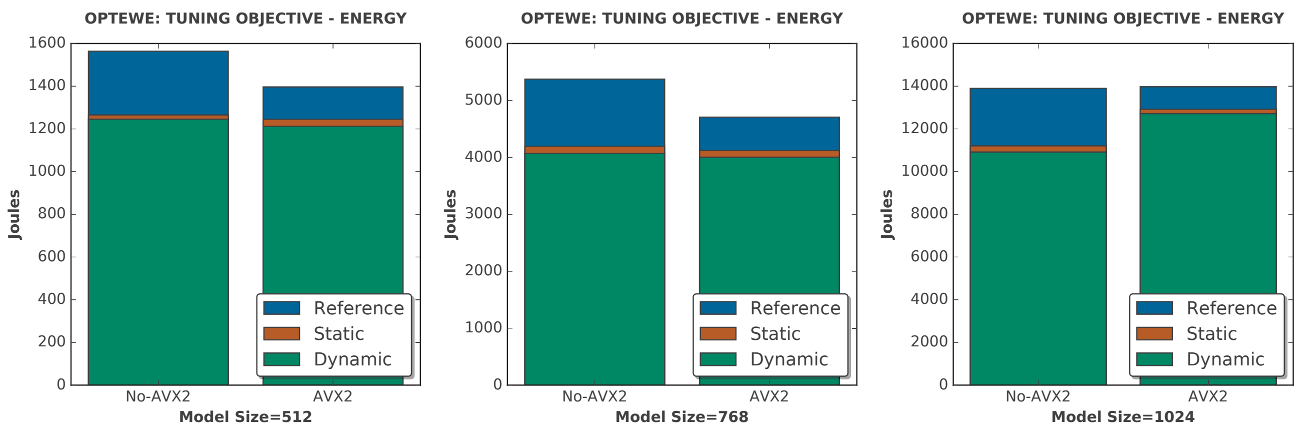
For the vectorized implementations, the dynamically tuned version reduces the ED2P values by 7.15%, 15.40% and 2.12%. Compared to the statically tuned version, the dynamically tuned version reduces the ED2P values by 4.87%, which is more than the reduction for the other two objective functions.

7.5 Tuning for Multiple Nodes

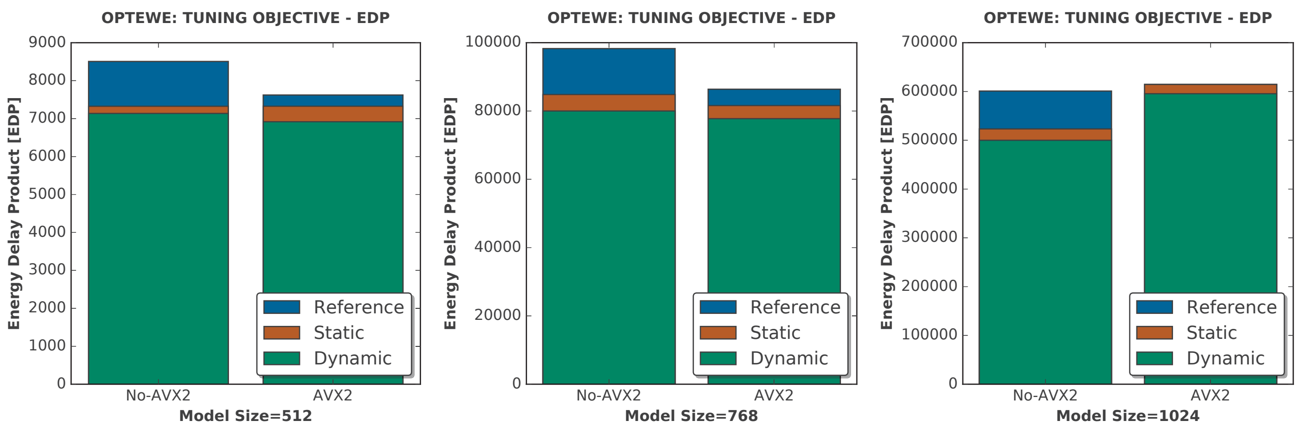
The focal point of our work is on dynamic tuning for a single node even though the presented application was designed with multiple nodes in mind. The understanding gained for a single node is required in order to tackle the increased complexity of analyzing multi-node codes due to multiple processes and their communication.

We foresee that the introduction of multiple processes impacts our work in two important ways. First, unlike core frequency scaling, where each individual core runs at a separate frequency, uncore frequency scaling can be performed only on a per-socket basis. This means that a multi-node implementation of our code should preferably be based on the hybrid MPI+OpenMP programming model. Moreover, in order to avoid race-conditions with respect to applying the dynamic settings, the application should be launched using only a single MPI process per node. Only a single thread within each process should be responsible for applying DVFS settings.

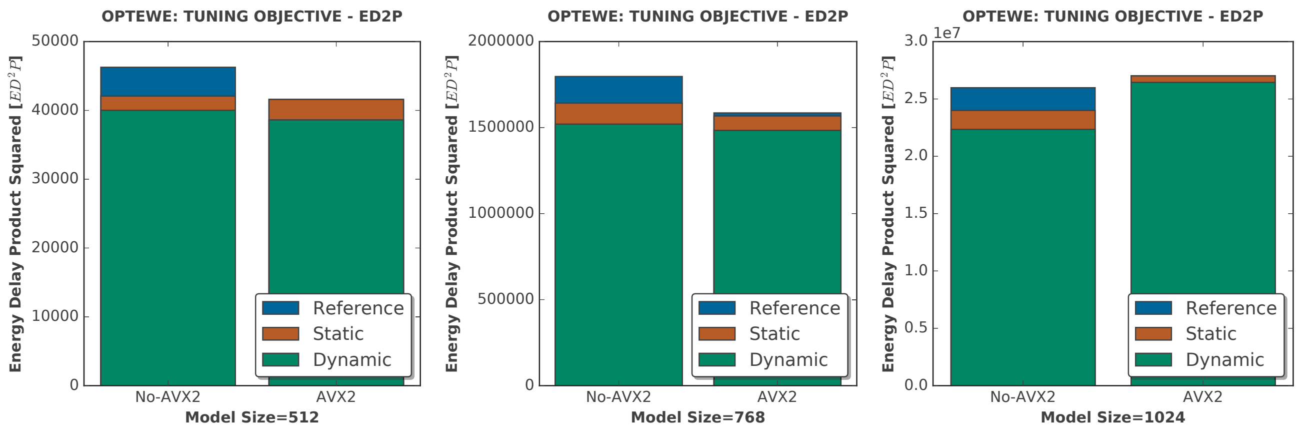
Other researchers developed generic frameworks for multi-node tuning [24] that rely on the use of agents to coordinate the tuning



(a)



(b)



(c)

Figure 4: Tuning results for the entire compute node with varying problem size. The figures on the top represent results when the tuning objective is energy, the figures in the middle, EDP, while the figures displays EDP2 results.

process. Typically, an additional compute node is allocated for the agent. However, we plan to implement a domain-specific approach where the load-imbalance between MPI processes located on the boundary of the partitioned domain as well as the center of the domain is exploited.

Processes located on the boundaries have a much lighter workload, while processes located at the domain center compute a bigger fraction of the global domain. This means that application performance is dominated by processes from the domain center, and the reduction of core and uncore for the boundary processes could potentially conserve energy.

8 CONCLUSION

Energy conservation remains a key challenge, both in today's supercomputing landscape and with respect to future systems. Luckily, with the increasing proliferation of user-controllable hardware switches such as dynamic voltage, core, and uncore frequency scaling on modern multi-core architectures, programmers can now tune their application for improved energy-efficiency.

So far, existing tools generally rely on static tuning, with the ultimate goal of finding a "one system configuration fits all" approach. However, we believe that the drawback of such a coarse-grained approach is that it fails to take rapidly changing computational behavior of an application into account. Instead, we advocate for a more fine-grained approach were each individual kernel is tuned and all kernels are executed with the best possible settings.

We have developed a custom dynamic tuning setup that automatically finds the best system configuration for different compute kernels by executing a small number of trial runs. Once the different configurations have been tested, the recommendation system suggest the best settings for the different kernels. The suggestions are determined based on a user-specified objective function, and can therefore change depending on the requirements. Next, the application is recompiled so that the suggested system configurations are mapped to their corresponding kernels at runtime. The end result is a dynamically tuned application that is ready to enter production using energy-conscious system configuration settings.

In order to quantify the impact of our tuning framework with respect to energy-performance, we have applied it to two versions of a long-running seismic wave simulator. In one version, the code is vectorized, and in the other version the code is not vectorized. Depending on the model size, our overall results show energy saving of up to 20% at the cost of at most 3.5% loss in performance compared to the corresponding reference implementation.

So far, many performance programmers targeting multi-core architectures have been advised to vectorize their code for increased performance and energy-efficiency. Although this advice still holds, we have demonstrated that it is possible to unlock new levels of energy-efficiency using a more fine-grained tuning. Because AVX2 instructions are more energy-hungry, running the application with a suboptimal system configuration proves to be costlier compared to the non-vectorized version.

Based on that, there are several different directions for future work. One direction is to automatically identify kernels that should not be tuned. Here, we plan to use a tool that performs a series of analysis to evaluate the operational intensity for different kernels and their runtime. Performing DVFS is associated with an overhead, and this overhead must be taken into consideration so that we only tune and perform DVFS on kernels that contribute substantially to the overall performance-energy ratio. Moreover, the encouraging results motivates development of a tool suite to fully automate the entire workflow. This is in line with work currently being pursued in the READEX project [32]. We believe that with the aid of this tool, we can more easily extend our current approach to other applications and architectures.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union's Horizon 2020 Programme in the READEX project under grant agreement number 671657. We thank the Centre for Information Services and High Performance Computing (ZIH) at TU Dresden for providing HPC resources that contributed to our research. Mohammed Sourouri dedicates his portion of this research to the memory of *Hans Petter Langtangen* (1962-2016).

REFERENCES

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman P. Amarasinghe. 2014. OpenTuner: an extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada*. ACM, 303–316. DOI:https://doi.org/10.1145/2628071.2628092

[2] Axel Auweter, Arndt Bode, Matthias Brehm, Luigi Brochard, Nicolay Hammer, Herbert Huber, Raj Panda, Francois Thomas, and Torsten Wilde. 2014. A Case Study of Energy Aware Scheduling on SuperMUC. In *29th International Conference, ISC 2014, Leipzig, Germany, June 22-26*. 394–409. DOI:https://doi.org/10.1007/978-3-319-07518-1_25

[3] Protonu Basu, Mary W. Hall, Malik Murtaza Khan, Suchit Maindola, Saurav Muralidharan, Shreyas Ramalingam, Axel Rivera, Manu Shantharam, and Anand Venkat. 2013. Towards making autotuning mainstream. *International Journal of High Performance Computing Applications* 27, 4 (2013), 379–393. DOI:https://doi.org/10.1177/1094342013493644

[4] JeeWhan Choi, Daniel Bedard, Robert J. Fowler, and Richard W. Vuduc. 2013. A Roofline Model of Energy. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS, Cambridge, MA, USA*. 661–672. DOI:https://doi.org/10.1109/IPDPS.2013.77

[5] Pietro Cicotti, Ananta Tiwari, and Laura Carrington. 2014. Efficient speed (ES): Adaptive DVFS and clock modulation for energy efficiency. In *International Conference on Cluster Computing, CLUSTER Madrid, Spain*. 158–166. DOI:https://doi.org/10.1109/CLUSTER.2014.6968750

[6] Yifeng Cui, Kim B. Olsen, Thomas H. Jordan, Kwangyoon Lee, Jun Zhou, Patrick Small, Daniel Roten, Geoffrey Ely, Dhabaleswar K. Panda, Amit Chourasia, John M. Levesque, Steven M. Day, and Philip Maechling. 2010. Scalable Earthquake Simulation on Petascale Supercomputers. In *Conference on High Performance Computing Networking, Storage and Analysis, SC '10, New Orleans, LA, USA*. 1–20. DOI:https://doi.org/10.1109/SC.2010.45

[7] Maja Etinski, Julita Corbalán, Jesús Labarta, and Mateo Valero. 2012. Understanding the future of energy-performance trade-off via DVFS in HPC environments. *J. Parallel Distrib. Comput.* 72, 4 (2012), 579–590. DOI:https://doi.org/10.1016/j.jpdc.2012.01.006

[8] Vincent W. Freeh, David K. Lowenthal, Feng Pan, Nandini Kappiah, Robert Springer, Barry Rountree, and Mark E. Femal. 2007. Analyzing the Energy-Time Trade-Off in High-Performance Computing Applications. *IEEE Trans. Parallel Distrib. Syst.* 18, 6 (2007), 835–848. DOI:https://doi.org/10.1109/TPDS.2007.1026

[9] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W. Cameron. 2010. PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications. *IEEE Trans. Parallel Distrib. Syst.* 21, 5 (2010), 658–671. DOI:https://doi.org/10.1109/TPDS.2009.76

[10] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. 2009. System-scenario-based design of dynamic embedded systems. *ACM Trans. Design Autom. Electr. Syst.* 14, 1, Article 3 (2009), 45 pages. DOI:https://doi.org/10.1145/1455229.1455232

[11] Ricardo Gonzalez and Mark Horowitz. 1996. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits* 31, 9 (1996), 1277–1284. DOI:https://doi.org/10.1109/4.535411

[12] Corey Gough, Ian Steiner, and Winston A. Saunders. 2015. *Energy Efficient Servers: Blueprints for Data Center Optimization* (1st ed.). Apress.

[13] Robert W. Graves. 1996. Simulating seismic wave propagation in 3D elastic media using staggered-grid finite differences. *Bulletin of the Seismological Society of America* 86, 4 (1996), 1091–1106.

[14] Philipp Gschwandtner, Juan José Durillo, and Thomas Fahringer. 2014. Multi-Objective Auto-Tuning with Insieme: Optimization and Trade-Off Analysis for Time, Energy and Resource Usage. In *20th International Conference on Parallel Processing Euro-Par, Porto, Portugal.* 87–98. DOI:https://doi.org/10.1007/978-3-319-09873-9_8

[15] Daniel Hackenberg, Thomas Ilsche, Joseph Schuchart, Robert Schöne, Wolfgang E. Nagel, Marc Simon, and Yiannis Georgiou. 2014. HDEEM: high definition energy efficiency monitoring. In *Proceedings of the 2nd International Workshop on Energy Efficient Supercomputing, E2SC '14, New Orleans, LA, USA.* 1–10. DOI:https://doi.org/10.1109/E2SC.2014.13

[16] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. 2015. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In *IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPSW.* 896–904. DOI:https://doi.org/10.1109/IPDPSW.2015.70

[17] Per Hammarlund, Alberto J. Martinez, Atiq A. Bajwa, David L. Hill, Erik G. Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, Randy B. Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stéphan Jourdan, Steve Gunther, Thomas Piazza, and Ted Burton. 2014. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro* 34, 2 (2014), 6–20. DOI:https://doi.org/10.1109/MM.2014.10

[18] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin C. Rinard. 2011. Dynamic knobs for responsive power-aware computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS Newport Beach, CA, USA.* 199–212. DOI:https://doi.org/10.1145/1950365.1950390

[19] Johannes Hofmann, Dietmar Fey, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2016. Analysis of Intel's Haswell Microarchitecture Using the ECM Model and Microbenchmarks. In *International Conference on Architecture of Computing Systems, Nuremberg, Germany.* Springer International Publishing, Cham, 210–222. DOI:https://doi.org/10.1007/978-3-319-30695-7_16

[20] Intel Corporation. 2016. Intel Xeon Processor E5 v3 Product Family - Processor Specification Update. http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v3-spec-update.pdf. (September 2016). [Online; accessed 08-March-2017].

[21] Herbert Jordan, Peter Thoman, Juan Jose Durillo Barrionuevo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. 2012. A multi-objective auto-tuning framework for parallel codes. In *Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA.* 10. DOI:https://doi.org/10.1109/SC.2012.7

[22] Dong Li, Bronis R. de Supinski, Martin Schulz, Kirk W. Cameron, and Dimitrios S. Nikolopoulos. 2010. Hybrid MPI/OpenMP power-aware computing. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS, Atlanta, GA, USA Proceedings.* 1–12. DOI:https://doi.org/10.1109/IPDPS.2010.5470463

[23] Dong Li, Dimitrios S. Nikolopoulos, Kirk W. Cameron, Bronis R. de Supinski, and Martin Schulz. 2010. Power-aware MPI task aggregation prediction for high-end computing systems. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS, Atlanta, Georgia, USA.* 1–12. DOI:https://doi.org/10.1109/IPDPS.2010.5470464

[24] Andrea Martínez, Anna Sikora, Eduardo César, and Joan Sorribes. 2014. ELASTIC: A large scale dynamic tuning environment. *Scientific Programming* 22, 4 (2014), 261–271. DOI:https://doi.org/10.3233/SPR-140392

[25] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec 1995), 19–25.

[26] Anna Morajko, Tomàs Margalef, and Emilio Luque. 2007. Design and implementation of a dynamic tuning environment. *J. Parallel Distrib. Comput.* 67, 4 (2007), 474–490. DOI:https://doi.org/10.1016/j.jpdc.2007.01.001

[27] Espen Birger Raknes, Børge Arntsen, and Wiktor Weibull. 2015. Three-dimensional elastic full waveform inversion using seismic data from the Sleipner area. *Geophysical Journal International* 202, 3 (2015), 1877–1894. DOI:https://doi.org/10.1093/gji/ggv258

[28] Barry Rountree, Dong H. Ahn, Bronis R. de Supinski, David K. Lowenthal, and Martin Schulz. 2012. Beyond DVFS: A First Look at Performance under a Hardware-Enforced Power Bound. In *26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPSW 2012, Shanghai, China.* 947–953. DOI:https://doi.org/10.1109/IPDPSW.2012.116

[29] Mazeiar Salehie and Ladan Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. *TAAS* 4, 2, Article 14 (2009), 42 pages. DOI:https://doi.org/10.1145/1516533.1516538

[30] Robert Schöne and Daniel Molka. 2014. Integrating performance analysis and energy efficiency optimizations in a unified environment. *Computer Science - R&D* 29, 3-4 (2014), 231–239. DOI:https://doi.org/10.1007/s00450-013-0243-7

[31] Robert Schöne, Jan Treibig, Manuel F. Dolz, Carla Guillén, Carmen B. Navarrete, Michael Knobloch, and Barry Rountree. 2014. Tools and methods for measuring and tuning the energy efficiency of HPC systems. *Scientific Programming* 22, 4 (2014), 273–283. DOI:https://doi.org/10.3233/SPR-140393

[32] Joseph Schuchart, Michael Gerndt, Per Gunnar Kjeldsberg, Michael Lysaght, David Horák, Lubomír Říha, Andreas Gocht, Mohammed Sourouri, Madhura Kumaraswamy, Anamika Chowdhury, Magnus Jahre, Kai Diethelm, Othman Bouizi, Umbreen Sabir Mian, Jakub Kružík, Radim Sojka, Martin Beseda, Venkatesh Kannan, Zakaria Bendifallah, Daniel Hackenberg, and Wolfgang E Nagel. 2017. The READEX formalism for automatic tuning for energy efficiency. *Computing* (2017), 1–9. DOI:https://doi.org/10.1007/s00607-016-0532-7

[33] Centre For Information Services and High Performance Computing (ZIH). 2017. SystemTaurus. https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/HardwareTaurus. (2017). [Online; accessed 28-February-2017].

[34] Anna Sikora, Eduardo César, Isaías A. Comprés Ureña, and Michael Gerndt. 2016. Autotuning of MPI Applications Using PTF. In *Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications, Kyoto, Japan.* 31–38. DOI:https://doi.org/10.1145/2916026.2916028

[35] Mohammed Sourouri, Scott B. Baden, and Xing Cai. 2017. Panda: A Compiler Framework for Concurrent CPU+GPU Execution of 3D Stencil Computations on GPU-accelerated Supercomputers. *International Journal of Parallel Programming* 45, 3 (2017), 711–729. DOI:https://doi.org/10.1007/s10766-016-0454-1

[36] Mohammed Sourouri, Johannes Langguth, Filippo Spiga, Scott B. Baden, and Xing Cai. 2015. CPU+GPU Programming of Stencil Computations for Resource-Efficient Use of GPU Clusters. In *International Conference on Computational Science and Engineering, CSE'15, Porto, Portugal.* IEEE Computer Society, 17–26. DOI:https://doi.org/10.1109/CSE.2015.33

[37] Cristian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. 2002. Active harmony: towards automated performance tuning. In *Conference on High Performance Computing Networking, Storage and Analysis, SC '02, Baltimore, MD, USA.* 43:1–43:11. DOI:https://doi.org/10.1109/SC.2002.10062

[38] Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. 2008. Data and Thread Affinity in OpenMP Programs. In *Proceedings of the Workshop on Memory Access on Future Processors: A Solved Problem?, MAW'08, Ischia, Italy, May 5-7.* ACM, New York, NY, USA, 377–384. DOI:https://doi.org/10.1145/1366219.1366222

[39] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary W. Hall, and Jeffrey K. Hollingsworth. 2009. A scalable auto-tuning framework for compiler optimization. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS, Rome, Italy.* 1–12. DOI:https://doi.org/10.1109/IPDPS.2009.5161054

[40] R. Clinton Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Conference on High Performance Computing Networking, Storage and Analysis, SC '98, Orlando, FL, USA.* 38. DOI:https://doi.org/10.1109/SC.1998.10004

[41] Samuel Williams, Andrew Waterman, and David A. Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76. DOI:https://doi.org/10.1145/1498765.1498785

A ARTIFACT DESCRIPTION APPENDIX

A.1 Abstract

The work presented in this paper describes a series of energy-efficiency techniques applied to a real-world seismic wave propagation application. The artifacts associated with this work comprise the source code of the seismic wave propagation, the result datasets and a post-processing tool. The post-processing tool is used for computing energy-performance tradeoffs and detecting the best system configuration for a given problem size and implementation.

A.2 Description

A.2.1 Check-list (artifact meta information).

- **Algorithm:** The paper details a seismic wave propagation application which consists of 25 different kernels. It solves the elastodynamic wave equation over a 3D Cartesian grid. C++ pre-processor directives are used to tune the application. The post-processing tool is used to find best system configuration for a given tuning objective. The best configuration reported by the tool is used to execute the application. The free and open-source x86_adapt library is used to perform dynamic core and uncore frequency scaling on Intel Xeon processors. The system was tested on processors based on the Haswell microarchitecture.
- **Program:** Optimized Elastic Wave Equation (OptEWE) parallelized using the OpenMP programming model. Source code available online.
- **Compilation:** Intel C/C++ compiler version (version 16.2.181 tested) and the linking of the x86_adapt library. The following compiler flags were used for the non-AVX2 version of the code: `-std=c++14 -Wall -O2 -qopenmp -ipo` and `-std=c++14 -Wall -O3 -axCORE-AVX2 -qopenmp -ipo` was used for the AVX2 vectorized version of the code. Use the designated batch scripts located in the batch directory to compile and launch the application.
- **Binary:** Standard single C++ executable for x86 multi-core CPUs compiled from the sources using GNU Make.
- **Data set:** The application itself does not require any data set to run, but the tuning process itself will output raw text files containing tuning results.
- **Run-time environment:** Any Linux based operating system with support for Intel core and uncore definitions compiled with x86_adapt kernel module support.
- **Hardware:** Any Intel Xeon EP CPU based on the Haswell microarchitecture or newer (2xIntel Xeon E5-2680v3 for the tests) and a high-resolution energy measurement infrastructure.
- **Execution:** ./optewe-mp Nx Ny Nz number-of-iterations
- **Output:** Text file reporting runtime for each kernel, the entire application and the problem size in different dimensions and performance throughput quantified as Mega Lattice Updates Per Second.
- **Experiment workflow:** clone the project using `git`, compile the application from the sources using the GNU Makefile, run the application using the bash scripts attached, use the post-processing tool to detect better system configurations, use the recommended system configuration to re-run the application with better system configurations.
- **Experiment customization:** number of OpenMP threads, core frequency and uncore frequency.
- **Publicly available?:** Yes

A.2.2 How software can be obtained. The complete set of artifacts can be found in the OptEWE open source project hosted on Github. Please visit: `https://github.com/mohamso/optewe` for more details. The mentioned git repository contains the application itself, the result dataset and its companion post-processing tool plus detailed instructions on how to build and run the application from the sources.

A.2.3 Hardware dependencies. For complete reproducibility, we recommend that Intel's Xeon EP series of CPUs based on the Intel Haswell microarchitecture or newer is used. It is recommended that Intel HyperThreading is disabled on the CPU(s). Our experiments utilize the HDEEM infrastructure for energy measurements. Although not tested, other measurement infrastructure such as Intel RAPL can be used.

A.2.4 Software dependencies. The OptEWE application requires a C++11 compliant compiler. Any compiler supporting this standard works, however, for adequate reproducibility, the Intel C/C++ compiler suite version 16.2.1.81 or newer is recommended. The x86_adapt library must be installed according to the documentation found on its associated Github web site. The post-processing tool requires Python version 2.6, Pandas Data Analysis library version 1.19 and Seaborn visualization library version 0.7.1 or newer. All artifacts have been tested on a recent version of Red Hat Enterprise Linux Server release 6.9 (kernel version 2.6.32-x86_64) but it is expected that it should work on all recent Linux distributions.

A.2.5 Datasets. The result of the autotuning can be found in the data directory. Use the shell scripts located in this directory to execute the scripts for the example model size used in the paper.

A.3 Installation

To install OptEWE, use `git` to clone the `optewe` repository from Github on your target machine:

```
$ git clone https://github.com/mohamso/optewe
$ cd code
```

Make sure that all required software dependencies are met. Next, use GNU Make to compile and build the code from the sources:

```
$ make clean
$ make
```

Upon successful build, an executable binary will be created in a directory called `bin`. To execute the code with a problem size of 512^3, using five iterations and 24 threads, type the following:

```
$ cd bin
$ ./optewe-mp 512 512 512 5 24 2
```

The last number indicates the source type. The source type represents the type of stress exhibited by the stress source. Valid source type values are 1 (stress monopole), 2 (force monopole) and 3 (force dipole). Experiments presented in the paper used force monopole.

A.4 Experiment workflow

Three versions of the code are presented in the paper: *reference*, *static* and *dynamic*. In the first step of the workflow, exhaustive search is used to tune the application. Next, the post-processing tools is used to find the best system configuration for the statically tuned and the dynamically tuned versions. Based on the result of the tuning, the post-processing tool will also generate the plots presented in the paper. It is sufficient to perform one exhaustive search per problem size in order to find the best static and dynamic system configurations. To tune the application using exhaustive search, do the following:

```
cd batch
$ sh static_wrapper.sh
```

Please note that depending on the problem size, the tuning process might take a while. It is thus recommended to keep the number of iterations low and limit the scope of number of threads. For example, on the tested dual-socket 12-core system, the minimum thread count for any tuning objective was never below 8 threads. Once the tuning process is complete, the post-processing tool can be used to find better system configurations. This can be done as follows:

```
cd results
sh create_all.sh
```

A.5 Evaluation and expected result

The expected results from the post-processing tool include a table that shows the best-found system configuration for each kernel. Moreover, it also contains tradeoff computations between the dynamically tuned, the statically tuned, and the reference versions.

A.6 Notes

For errata, bug-fixes and updated instructions, please visit the code web page: `https://github.com/mohamso/optewe`.