Ref. Ares(2016)4890913 - 31/08/2016

GA no. 671657

D1.1
Hardware and System-Software Tuning Plugins

Document type:	Other
Dissemination level:	Public
Work package:	WP1
Editor:	Andreas Gocht (TUD), Zakaria Bendifallah (Intel), Umbreen Sabir Mian (TUD), Othman Bouizi (Intel)
Contributing partners:	TUD, Intel
Reviewer:	Michael Gerndt (TUM) David Horak (IT4I)
Version:	1.0

Document history

Version	Date	Author/Editor	Description
0.1	08/08/16	Andreas Gocht (TUD), Zakaria Bendifallah (Intel), Othman Bouizi (Intel)	First draft
0.2	22/08/16	Andreas Gocht (TUD), Zakaria Bendifallah (Intel), Othman Bouizi (Intel)	Second draft
1.0	30/08/16	Andreas Gocht (TUD), Zakaria Bendifallah (Intel), Umbreen Sabir Mian (TUD), Othman Bouizi (Intel)	Final Version

Executive Summary

The objective of Work package 1 (WP1) “Tuning parameters” is to determine parameters that can be tuned as well as their integration into the READEX tool suite. We split the tuning parameters in three different levels of the HPC stack: Hardware, Runtime System, and Application. We evaluated the parameters proposed in Deliverable 4.1 and integrated the most promising ones in the current READEX tool stack.

As the READEX tool suite supports only static tuning at the time this Deliverable is written, testing the dynamic tuning potential of all parameters was not in scope of this deliverable. However, there is still a certain amount of parameters which show their dynamic tuning potential already during static tuning. Once the READEX tool suite is able to tune for dynamic situations we will re-evaluate all plugins.

After a short introduction this Deliverable starts with an outline of the current READEX tool suite. We describe in which places we have to hook in, to integrate the different tuning parameters as well as the basic concepts of the READEX tool suite.

Furthermore, we describe in detail our initial assumptions as well as our concrete findings about the hardware and runtime system parameters. Our results showed that the processor core and uncore frequencies are the most promising hardware tuning parameters. Moreover the amount of threads used for OpenMP computations as well as the scheduler policy and the scheduler chunk size are promising parameters for runtime system parameters. The proposed MPI parameters will need further investigation once we are able to evaluate the dynamic tuning potential. We also give a small outline about the integration into the READEX tool suite for these parameters.

Finally, we describe our concept for the application parameters. These parts are still under development and result in a refinement during the next project months. However, we will revisit the application parameters within Deliverable 1.2.

Contents

1 Introduction **5**

1.1 Experiment Platform . . . 5

2 Integration in the READEX Tool Suite **6**

3 Hardware Parameters **8**

3.1 Dynamic Voltage and Frequency Scaling and Uncore Frequency Scaling . . . 8

3.2 Dynamic Duty Cycle Modulation . . . 10

3.3 Energy Performance Bias . . . 11

3.4 Hardware Prefetcher . . . 12

4 System Software Parameters **15**

4.1 MPI . . . 15

4.2 OpenMP . . . 17

5 Application Parameters **20**

5.1 Application Level Tuning . . . 20

5.2 Parameter Exploitation Scenarios . . . 20

6 Summary **23**

1 Introduction

The READEX project focuses on the dynamic tuning of applications in order to improve their energy efficiency. Therefore, we need different parameters that have a noticeable impact on the energy efficiency of programs. In this Deliverable we describe the parameters that we investigated.

For energy efficiency tuning each parameter needs a corresponding parameter control plugin which will allow an integration into the READEX tool suite. Therefore we will start in Section 2 with a short description of the READEX tool suite. We will outline how these plugins are controlled and where these plugins have to be integrated.

The following Sections 3 - 5 present hardware parameters, system software parameters, and application-level parameters. Each section will outline our findings about the proposed tuning parameters. For selected parameters we give a small overview about some parameter specific implementation details.

Finally, we will summarize our findings and give an outlook about our future work.

1.1 Experiment Platform

All experiments for the tuning parameters are conducted on the Taurus system installed at TU Dresden. This system is comprised of 1456 nodes each containing two 12-core Intel Xeon CPUs E5-2680 v3 (Intel Haswell processor family) running with a default frequency of 2.50 GHz. The nodes contain between 64 and 256 GB of memory.

All nodes are equipped with the HDEEM energy measurement system [4]. This system allowed us to do reasonably accurate measurements as the measurement error is 2% for measurements of the whole node [3].

Additionally, 44 nodes are equipped with two Nvidia Tesla K20x GPUs and 64 nodes contain four Nvidia Tesla K80 GPUs per node.

2 Integration in the READEX Tool Suite

All tuning plugins and parameters that we describe in the following sections have to be integrated in the READEX tool suite. Therefore, we start with a small overview about the READEX tool suite stack.

Figure 1 outlines the interaction between the Periscope Tuning Framework (PTF), Score-P and the READEX Runtime Library (RRL).

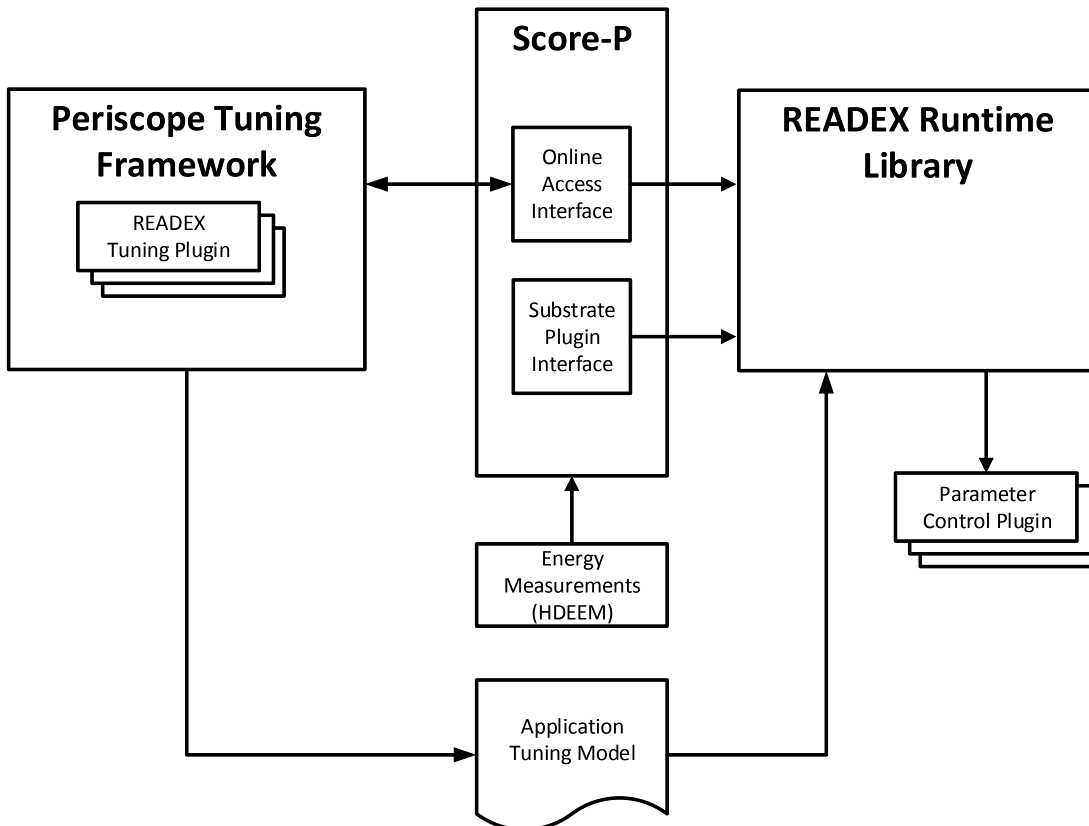

Figure 1: Overview about the READEX tool suite. The graph shows the information flow between all the tools we are going to use or to develop. The “Parameter Control Plugins” are the plugins that will change the tuning parameters.

The RRL controls the so called Parameter Control Plugins (PCPs). These plugins are responsible for changing different hardware or software parameters. Each PCP provides a common interface, which contains the name of a parameter and a set and an unset function.

During design time, the RRL receives its tuning commands through the Score-P online access interface. The interface in turn is connected to PTF. A tuning command contains the name

of the parameter to change, the parameter value and the program region where it shall be applied. PTF obtains the information about the available parameters and the parameter values from so called tuning plugins. Each tuning plugin can target different parameters at once. Once PTF finds an optimal configuration, it saves this configuration in the so called tuning model.

During runtime, the RRL requests the optimal configuration from the tuning model and passes it to the PCPs. In a future version of the READEX tool suite we will implement a calibration mechanism, which will be activated during runtime. If a scenario occurs that has not been seen during design time, the calibration mechanism will search for an optimal configuration of this scenario.

The different parts of the software are maintained in three different repositories. We use a modified version of Score-P, which we will merge to the official Score-P once our changes are final. Our Score-P version can be accessed about the non-public Score-P SVN project archive hosted by TU Dresden. PTF together with the tuning plugins, the RRL and the PCPs are located at a non-public Git project archive hosted by TU Munich. Access to these archives are granted after requests to responsible project partners.

3 Hardware Parameters

The most relevant hardware parameters that we consider are processor related parameters. This is mainly due to the fact that the processor has the highest power dissipation in a computer system. In the following sections, we will describe each of the hardware tuning parameters and tuning aspects that we investigated as part of the READEX project.

3.1 Dynamic Voltage and Frequency Scaling and Uncore Frequency Scaling

The method of Dynamic Voltage and Frequency Scaling (DVFS) has been investigated since the 1990's [13] as a means of reducing the energy consumption of computer systems. Reducing the frequency of a CPU core through DVFS results in the reduction of the required power draw of the platform, where energy savings of up to 32% have been reported [9]. DVFS can be implemented through various means, e.g., by changing the governor. A governor describes a pluggable infrastructure that commonly controls the CPU frequency settings in the Linux operating system based on defined policies such as *performance*, *powersave*, or *ondemand*. For full control over the frequency settings, the so-called *userspace* governor allows applications to select the so-called P-state, which are essentially frequency steps of the processor.

One interesting point to emphasise is that the Intel Haswell processor family allows the independent selection of P-states for individual cores as opposed to full sockets as seen in previous processor lines. The Intel Haswell processor also introduces a switching window for frequency changes. Each switching request can be given to the processor in a certain time window. At the end of this window the frequency is changed. If the request occurs after a time window is closed, the switching is executed after the next window is closed. This causes a delay of up to 500 μs before switching decisions become effective [5].

Moreover, there is the uncore, which controls for example the communication between processor caches and the DRAM. Together with the Haswell processor generation Intel reintroduced an independent uncore frequency. This allows to do so called Uncore Frequnecy Scaling (UFS). The Haswell processor usually sets the uncore frequency automatically. But it is also possible to control the uncore frequency by setting machine specific registers (MSRs).

3.1.1 Research summary

In order to investigate the effect of UFS and DVFS we created a modified version of the well-known STREAM benchmark [8, 7]. To allow energy measurements we change the stream benchmark to support the HDEEM measurement environment [4]. We are measuring the energy consumption for different combinations of core and uncore frequencies using two different data array sizes. In the first case, shown in Figure 2 the data array is significantly larger than the L3 processor cache. In the second case, shown in Figure 3 the size of the data array was chosen to fit into the L3 cache of the processor. Both cases have a different amount of repetitions. The energy consumption is therefore not directly comparable between

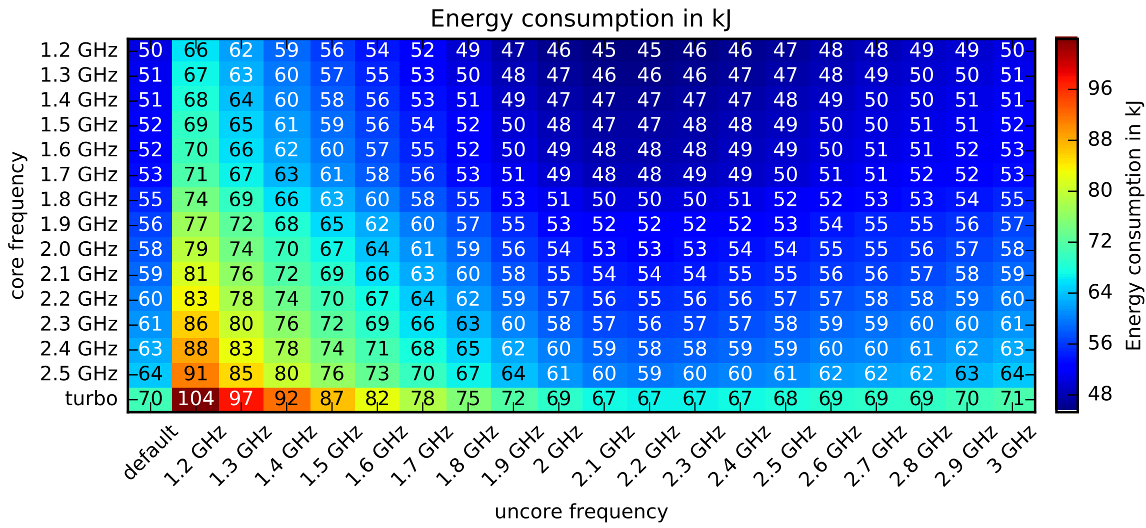

Figure 2: Heatmap of the energy consumption of a stream benchmark for different core and uncore frequencies. The data array does not fit in the processor's L3 processor cache

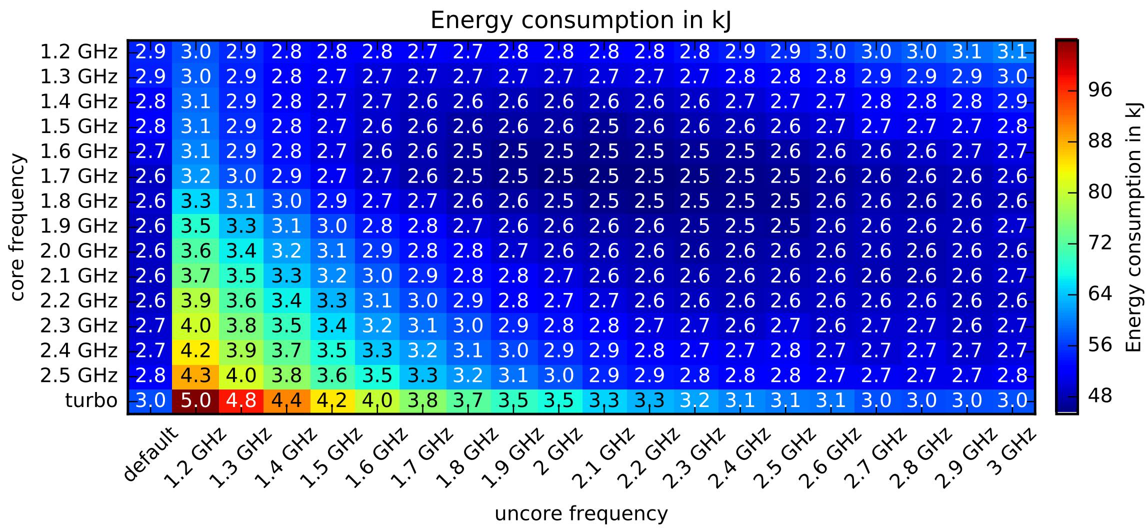

Figure 3: Heatmap of the energy consumption of a stream benchmark for different core and uncore frequencies. The data array does fit in the processor's L3 processor cache, which results in a different characteristic than in Figure 2.

these figures. However, while the first case has an optimal configuration at a core frequency of 1.2 GHz and an uncore frequency of 2.1 GHz, the second case has its optimum between a core frequency of 1.5 GHz and 1.9 GHz and an uncore frequency between 1.8 GHz and 2.5 GHz. We conclude that application regions working on data that fits into the L3 cache have a different optimal configuration than regions that access a large amount of DRAM memory.

3.1.2 Implemetation of the tuning plugin

We implemented and tested a parameter control plugin and a tuning plugin for both, the core and the uncore frequencies. The UFS plugin uses the x86_adapt library [11] to set the relevant MSR of the processor. The information about the controlling MSR is non-public and available to us under an NDA agreement with Intel.

Both plugins allow the user to specify a frequency range that should be searched as well as the size of each step. At the moment it is up to the user to define the search space. This might change in coming versions of the READEX tool suite.

3.2 Dynamic Duty Cycle Modulation

Another parameter we wanted to investigate is a feature known as Dynamic Duty Cycle Modulation (DDCM). This technique involves so-called T-states which instruct the processor to statistically skip a user-defined number of clock cycles, i.e., between 12.5 % and 87.5 % of the overall clock cycles for a given time period. We assumed that this could be beneficial in program regions where not all cycles could be used effectively, e.g., memory- or I/O-bound regions or MPI wait-states. Furthermore, we expected DDCM to suffer less from the switching window introduced with the Haswell processor generation as described in Section 3.1

3.2.1 Research summary

Our experiments confirmed that DDCM can reduce the energy consumption. But on the Haswell processor DVFS is more effective to achieve energy savings than DDCM. This is mainly due to the per core P-States that where introduced together with the Haswell processor generation. As DVFS does not only reduce the core frequency but also the core voltage it is able to save more energy. For example skipping 87.5 % of the overall clock cycles has a similar effect than reducing the processor frequency by 50 % using DVFS. Consequently, the performance loss by using DDCM is much higher when we try to achieve the same power savings. Also, a significant region in the READEX context is around 100 ms long. Therefore, the switching window doesn't have a significant impact on our tuning approach.

That's why DDCM will no longer be considered in READEX.

3.3 Energy Performance Bias

The Energy Performance Bias (EPB) is a setting that influences different energy efficiency related features on the processor, e.g., the uncore frequency and the energy-efficient turbo [5]. The EPB can be changed using a MSR register, which offers 16 different settings. However, accorind to Hackenberg et.al. [5], only three different settings are defined on the Haswell processor, which can be represented by 0, 6 and 15. These settings map to the policies *performance*, *energy saving*, and *balanced*. The remaining settings are mapped to these three policies.

3.3.1 Research summary

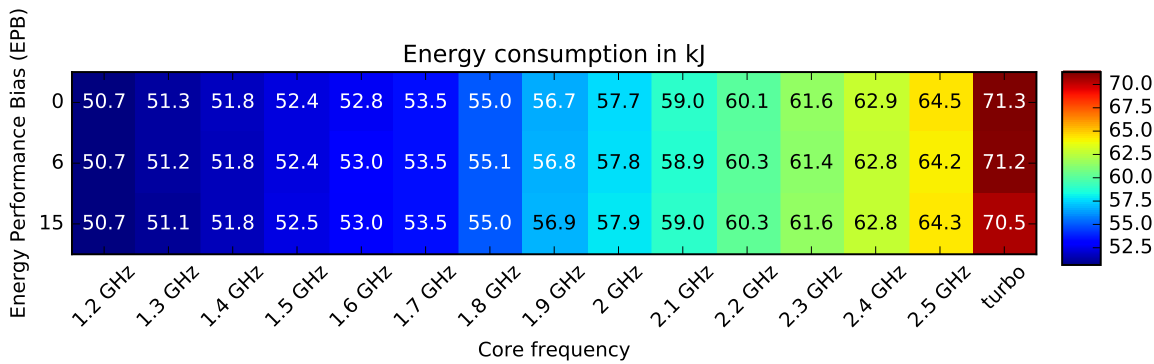

Figure 4: Energy consumption of the STREAM benchmark for different core frequencies and EPB settings.

We expected that changing the EPB can change parameters on the CPU that are not accessible through software. Unfortunately, we realised that the savings that can be achieved by changing the EPB are not as high as expected in our test cases. For example the STREAM benchmark profits only a little from varying the policies as shown in Figure 4. As the memory requirements of the benchmark can't be fulfilled by the memory bandwidth, the processor has a lot of stalls, which leads to a non optimal setting of the uncore frequency [5]. The core frequency is set using the userspace governor. The most noticeable effect can be shown at turbo frequency where the *energy saving* setting leads to a reduction of the turbo frequency, which in turn leads to small energy savings.

3.3.2 Implementation of the tuning plugin

We assume, that in certain scenarios the EPB might still be able to achieve significant energy reductions, e.g. if UFS is not available for the user. We therefore implemented a control and a tuning plugin for the EPB. Both plugins are similar to the UFS plugin. It uses as well the x86_adapt library [11] in order to set the related MSR from the userspace. As we want

to be able to achieve most energy savings even on future processor, we decided to make all 16 settings for the EPB accessible to the user. Therefore, to use the plugin the user has to specify a EPB range that should be searched as well as the size of each step.

3.4 Hardware Prefetcher

Hardware prefetching has proven to be efficient in many cases to hide the latency of moving data to higher cache levels. However, in some cases it does not lead to any better performance, mainly because the placement of data in caches is hardware defined, it may even induce lower performance in some rare cases. These characteristics and the possibility to turn hardware prefetching on and off makes it a candidate for energy saving in READEX.

Intel processors from Nehalem to Broadwell provide four types of hardware prefetchers. Two of them are associated with the L1 data cache known as DCU. The other two prefetchers are associated with the L2 cache. These are listed in Table 1. The counters can be turned on and off through MSR registers. We use the X86_adapt library to do so.

Prefetcher Name	**Description**
L2 hardware prefetcher	Fetches additional lines of code or data into the L2 cache
L2 adjacent cache line prefetcher	Fetches the cache line that comprises a cache line pair (128 bytes)
DCU prefetcher	Fetches the next cache line into L1-D cache
DCU IP prefetcher	Uses sequential load history (based on Instruction Pointer of previous loads) to determine whether to prefetch

Table 1: Types of hardware prefetchers on Intel processors.

3.4.1 Research summary

In order to assess the impact of hardware prefetchers on energy consumption, we used a suite of 27 diverse compute kernels written in Fortran and extracted from Numerical Recipes [10]. The kernels can be classified into six categories following their access patterns:

- 1D loop with stride one access (1D_loop_stride_1),
- 1D loop with diagonal access (1D_loop_stride_CLDA),
- 1D loop with line wise access (1D_loop_stride_LDA),
- 2D loop with stride one access (2D_loop_stride_1),
- 2D loop with diagonal access (2D_loop_stride_LDA),

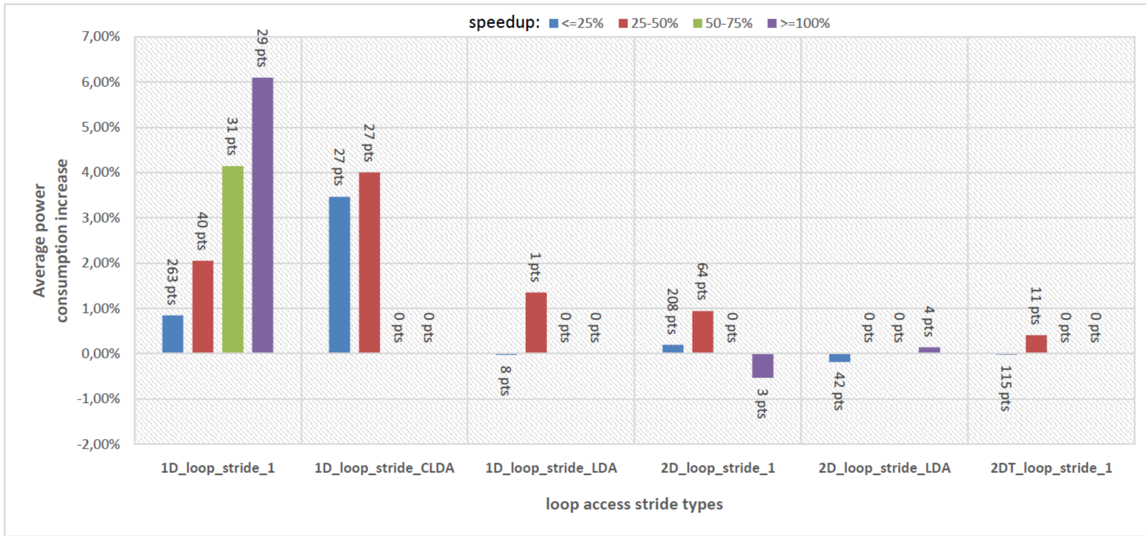

Figure 5: Average increase in power consumption following the achieved speedup for different data access patterns. The number of points collected for each case is noted on top of the corresponding histogram

- 2D loop with triangular access (2DT_loop_stride_1).

In order to have reliable energy measurements using the HDEEM measurement environment [4], we inject a repetition loop around the main loop of each kernel to create an execution window sufficiently big for HDEEM to return stable and reliable energy consumption measurements. Furthermore, for each kernel we run the experiments on several problem sizes that cover the entire cache hierarchy. However, the fact that prefetchers are active does not necessarily mean they are working. The hardware can still decide to shut them down. In order to track the cases where the prefetchers are actually active and working, we detect the cases where speedups have been achieved due to hardware prefetching.

Performance experiments revealed that for the entire suite, the only prefetcher that was not neutral in terms of performance change is the *L2 hardware prefetcher*. The other prefetchers did not induce any changes neither in performance nor in energy consumption. Therefore, we only show results for this prefetcher.

Figure 5 illustrates for each of the six access pattern categories the average increase in power consumption following the performance speedup achieved. From figure 5 we can make the following observations: 1) The power increase is proportional to execution time speedup, which means that either active prefetchers consume more power or lead to better use of the core resources which induces more power consumption too. A mix between the two is also possible. 2) Besides the stride one access pattern categories, the other categories are composed mostly of small numbers of instable points. Therefore, we could not take them as

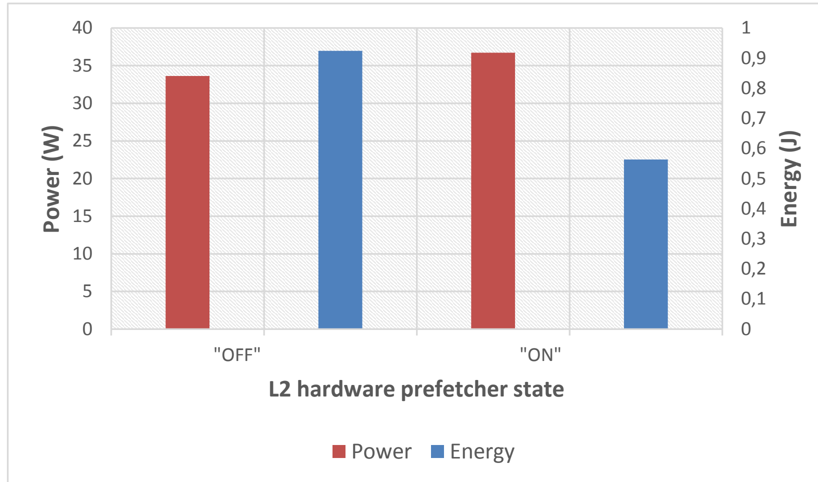

Figure 6: Comparison between the total energy consumed to finish kernel execution and the power consumption for the Elmhes_10 kernel when data are in RAM

reliable results, we may also conclude from this that it is likely that prefetchers are set to target stride one accesses.

However, in all our experiments we observed that the decrease in execution time reduces the overall energy consumption of the kernels. Figure 6 is an example of such case, where it illustrates a comparison between power and energy in the case where a speedup is achieved. From the figure, we can see an average increase of 9% in power consumption for the *Elmhes_10* kernel when the prefetchers are active, but a reduction of execution time which led to 64% of decrease in total energy consumption.

Consequently, we concluded that there is no need for us to control hardware prefetchers as we noticed that they can be either neutral both for performance and energy or beneficial for the two of them.

4 System Software Parameters

The majority of scalable HPC applications employ the Message Passing Interface (MPI) for distributed memory parallelism. Many HPC applications now are also employing OpenMP to target thread-level parallelism as part of a so-called hybrid model. Due to their widespread adoption within the HPC community, both of these programming models are investigated in the READEX project.

4.1 MPI

Tuning of MPI implementations for different HPC environments is a well-known topic. Most HPC centres have a unique combination of computing hardware and network hardware. Therefore, the used MPI implementation needs to be tuned to reach the best possible bandwidth. But the MPI environment cannot just be optimised to a certain hardware configuration but to different programs as well. A. Sikora et. al. [12] showed that it is possible to improve performance of a program using an automatic static tuning approach.

The MPI 3.0 standard has introduced the so-called MPI-T interface that allows MPI implementations to offer a set of parameters that can be read and written by the user during runtime. These are so called CVars. They are allowing us to do dynamic tuning as well. We initially focus our investigation on the latest version of the well-known open source MPICH implementation [1] of MPI, which offers the largest and most relevant set of parameters for runtime tuning. For example, MPICH exposes the definition of short and long messages for many MPI commands as well as parameters influencing shared memory operations.

4.1.1 Research summary

As outlined in Deliverable 4.1, we investigated parameters for the MPI commands `MPI_Reduce` and `MPI_Alltoall`. The commands represent so-called collective communication operations. Relative to direct point-to-point communication between two MPI processes, collective operations are more expensive, as more processes are involved. For example, `MPI_Reduce` collects results from different given processes and can carry out different calculations, e.g. it is possible to determine the minimum of a value from all processes. However, the parameters we investigated can be found in Table 2.

During our experiments we realised that some parameters just have an effect, if other parameters are set or unset. For example, changing `MPIR_CVAR_REDUCE_SHORT_MSG_SIZE` has only an effect if `ENABLE_SMP_REDUCE` is set to 0, which means it is disabled. The results are shown in Figure 7. We used two nodes with 24 MPI processes each. It can be seen that there is an effect if we modify `MPIR_CVAR_REDUCE_SHORT_MSG_SIZE`. In detail the parameter influences the decision about the used reduction algorithm. While large messages do not profit from tuning, the results for small messages are not unambiguous.

Table 2: MPI tuning parameters to be further investigated.

Tuning Aspect	Tuning Parameter (MPIR_CVAR_*)	Description
MPI_Reduce	REDUCE_SHORT_MSG_SIZE	The short message algorithm will be used if the send buffer size is ≤ this value (in bytes)
	ENABLE_SMP_REDUCE	Enable SMP aware reduce
	MAX_SMP_REDUCE_MSG_SIZE	Maximum message size for which SMP-aware reduce is used. A value of '0' uses SMP-aware reduce for all message sizes
MPI_Alltoall	ALLTOALL_SHORT_MSG_SIZE	The short message algorithm will be used if the per-destination message size (sendcount*size(sendtype)) is ≤ this value
	ALLTOALL_MEDIUM_MSG_SIZE	The medium message algorithm will be used if the per-destination message size (sendcount*size(sendtype)) is ≤ this value and larger than the short message size

Messages with sizes of about 64 Byte show an optimal configuration of 0 or 4 for MPIR_CVAR_REDUCE_SHORT_MSG_SIZE. A closer look shows that there are huge variations over the 20 taken measurements, which can be seen in Figure 8. Each measurement consists of 16384 MPI_REDUCE operations for messages with the length of 64 Bytes. As the results are as unstable and don't have any additional value, we decided to not present energy measurements or further MPI parameter results in this Deliverable.

Although our measurements don't look as promising as we had hoped, we will do further investigations for these MPI parameters. We are expecting real world applications to behave different as our benchmark is hardly able to cover real world effects like latency. To do so we need a prototype of the READEX tool suite, which allows us dynamic tuning. Unfortunately, this prototype is not available at the time this Deliverable is written, as the alpha is scheduled at the end of Project Month 18.

4.1.2 Implemetation of the tuning plugin

We implemented a control plugin and a tuning plugin for the MPIT CVars. The tuning plugin allows to specify the CVar that the user likes to change as well as the value to set. Unfortunately, the set of available CVar parameters can vary between different MPI implementations or even versions of these implementation. Moreover, the current static PTF API does not allow flexible choosing of tuning parameters. Therefore, we decided to design a user

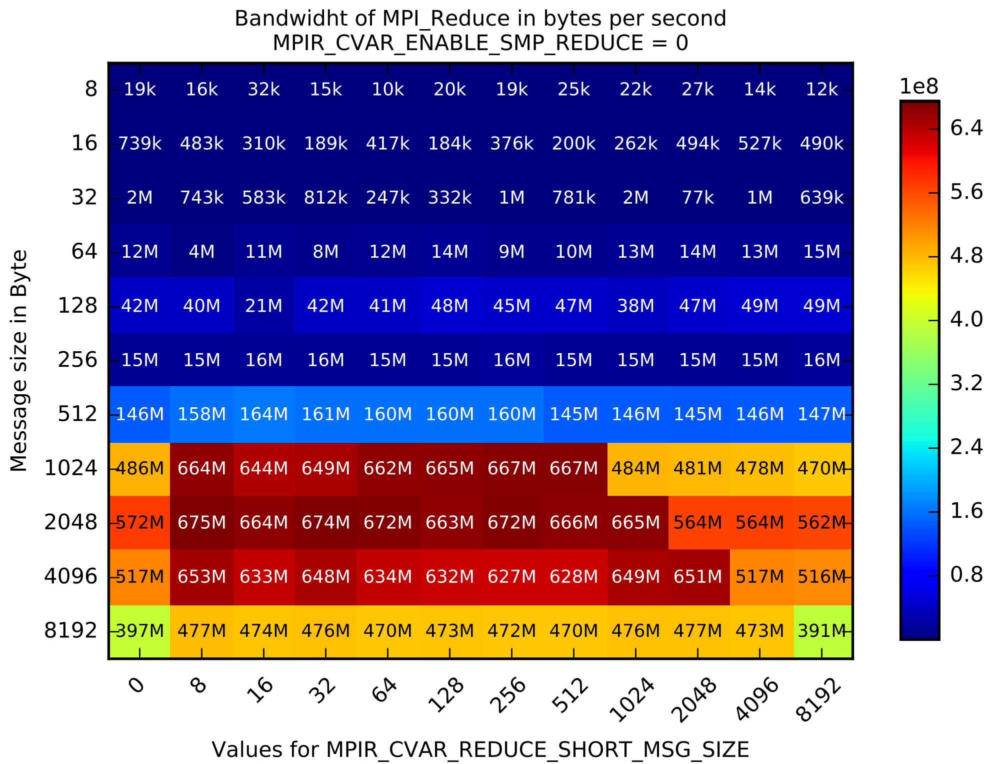

Figure 7: MPI_Reduce bandwidth for different settings of MPIR_CVAR_REDUCE_SHORT_MSG_SIZE, the SMP-awareness is disabled.

interface that is as flexible as possible. In the current implementations of the plugin, the user has to specify each parameter he would like to tune and to implement these CVar in the parameter control plugin. To make this as easy as possible to do, we built a C++ abstraction around the MPIT CVar interface.

4.2 OpenMP

The OpenMP standard offers users a way to implement thread-parallelism through directives, which are translated by the compiler into thread-parallel code. For example, one of the most commonly used OpenMP parallelisation technique is offered by the OpenMP `parallel-for` directive. The directive allows the distribution of all iterations of a loop among a defined number of threads. At the same time, the OpenMP standard provides an API to control the behaviour of the OpenMP runtime library like the number of threads or the scheduler policy.

Setting the number of threads to a number lower than the amount of processor cores in a system allows unused processor cores to reduce their C-state. In detail, the core changes from the active C0 state into any of the power-saving states C1–C6, in which no computation can

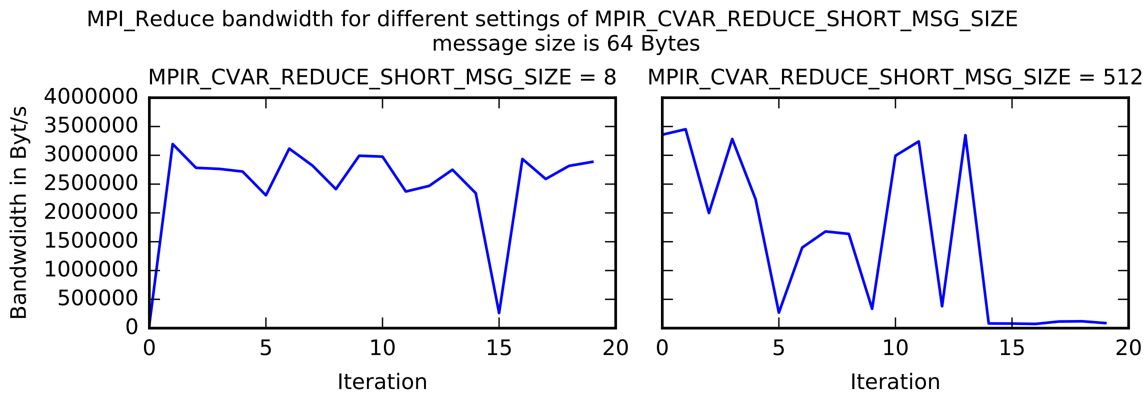

Figure 8: MPI_Reduce bandwidth for a message size of 64 Byte. MPIR_CVAR_REDUCE_SHORT_MSG_SIZE is set to 8 (left) or 512 (right). We can see that there is a huge variation of the bandwidth over the 20 measurements. Therefore, the results are not reliable.

be performed [6, 11]. The C-States differ from each other in how fast they can be woken up, and how much power can be saved by using them. Thus, reducing the number of threads and letting a subset of the available cores go into a sleep-state can improve energy efficiency. Changing the workload scheduler policy can also be used to optimize the energy efficiency by controlling the distribution of work among the threads. This can have a positive impact if not all loop iterations require the same amount of computational work.

4.2.1 Research summary

Besides the number of threads, we changed the scheduler policy as well as the chunk size. We executed a few experiments for the NPB BT-MZ benchmark which showed that there is some effect on the energy consumption depending on the number of threads.

Figure 9 shows that there is an energy optimum for the auto scheduler at around 10 threads. This saves up to 35% against the case that uses the maximal number of cores using the same scheduler. However, as the READEX prototype is not available at the time when this Deliverable is written, we are not able to evaluate the dynamic saving potential.

4.2.2 Implemetation of the tuning plugin

We implemented a parameter control plugin and a tuning plugin. The tuning plugin allows the user to specify the number of threads, which he likes to search. Moreover, the plugin itself changes the scheduler policy according to the available policies specified in the OpenMP specification [2]. Finally, the tuning plugin is able to change the chunk size, which is associated to the scheduler.

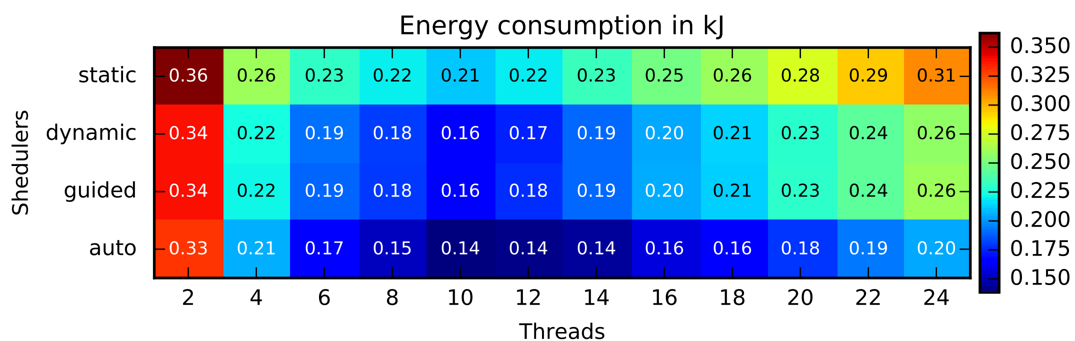

Figure 9: Different OpenMP scheduler policies for a different number of threads. The chunk size is set to 256.

5 Application Parameters

The idea behind READEX is to exploit the dynamicity available inside the application and use it to reduce its energy consumption. The approach to exploit this dynamicity can be subdivided into two main categories:

1. code-agnostic approach, which includes hardware and system level parameters discussed in Sections 3 and 4 and
2. code-aware approach, which is discussed in the current section.

In the first category, no advanced knowledge on the application code itself, what it does or what algorithms are used, is necessary. Profiling techniques on the application are sufficient to detect and define the hints to exploit dynamicity. On the other hand, code-aware approach involves both automatic exploration and the knowledge of the application developer. In this case, the developer himself can give the tool hints, to find spots where dynamicity can be exploited.

5.1 Application Level Tuning

A few examples of the spots that can be exploited for tuning are the possibilities:

- to choose different code paths (types of iterative and direct solvers, preprocessing of stiffness and coarse problem (CP) matrices) correspond to cases where the developer identifies more than one way to solve his problem. This results in different pieces of code to choose from. What he does not know is what their characteristics are and how they can be exploited for energy tuning.
- to control problem decomposition (number of subdomains, cache blocking, ...).
- to give hints on data structures (array length, access strides, indirections, ...).

5.2 Parameter Exploitation Scenarios

Our enumeration of possible types of application parameters and early experiments allowed us to think so far of two approaches for the exploitation of application parameters *offline* and *online* which we describe below.

5.2.1 Offline Approach

In the offline approach, the different alternatives are known before the execution of the application. The selection between these alternatives can be static (before execution) or dynamic (during execution), each of these two approaches has advantages and drawbacks:

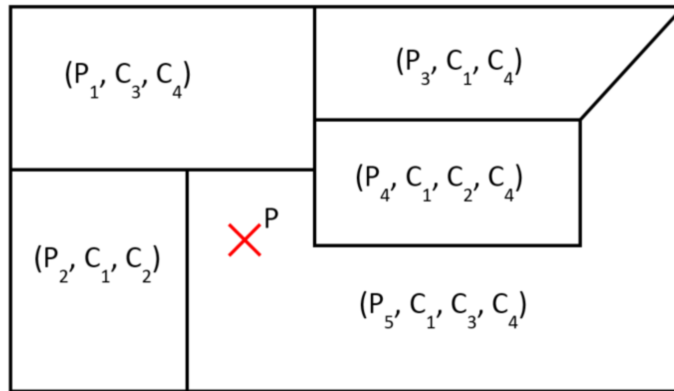

Figure 10: Example of possible space of values for a parameter P and the subdomains in which the code paths $C\{1,2,3,4,5\}$ apply.

- Static: the chosen code paths are set before the application is compiled. This can be done through pragmas which will be inserted in the source code. The approach has the advantage of letting the compiler optimize the code more efficiently. Although, in this approach a default path must be defined in order to allow the code to compile without any knowledge of the alternate code paths.
- Dynamic: the selection of a code path is done dynamically during code execution. The READEX tools suite must instruct the code where to branch, and communicate with the application. Compared to the static method, no recompilation of the application is needed, which can in some heavy applications save time.

As the names already imply only the second approach would fit into the READEX methodology. It would allow to choose the most energy efficient code path during runtime.

A benefit of the offline approach is that the different possible code paths are already known during the Design Time. Therefore it would not lead to any unforeseen situations at runtime.

5.2.2 Online Approach

In the online approach, the search space of the parameters is not known before the execution of the application but discovered or calculated at runtime, therefore the selection is done at runtime too.

We can imagine the case of a region of the code which has access to different call paths that return the same result at the application level. The developer of the application can define a parameter P whose value will drive the choice of the code path C_i. The parameter P belongs to a space of values, e.g. Figure 10.

That space can be divided into regions, which contains a finite number of call paths. This imposes a certain degree of flexibility on the parameter passing mechanism. The separation

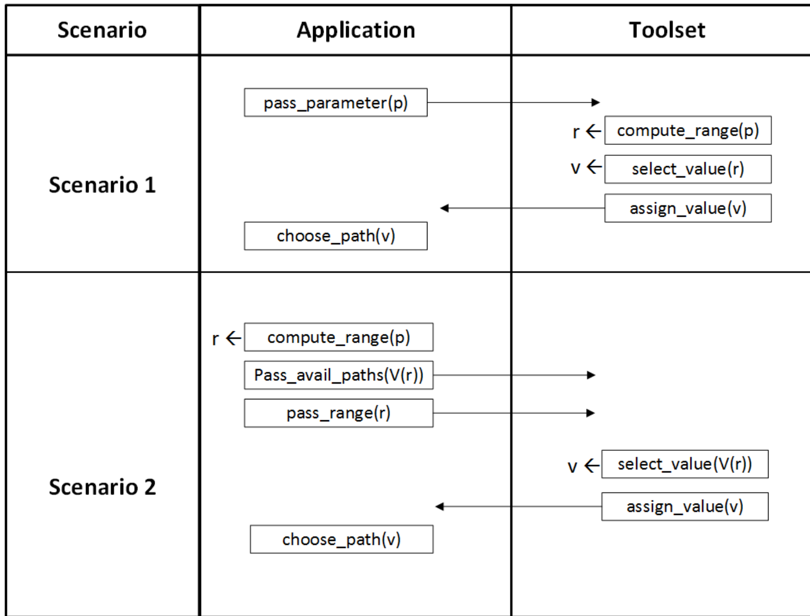

Figure 11: Possible parameter communication scenarios between the application and the READEX tool suite

between what the user should supply and what the toolset can do is important. We therefore identify two notable scenarios for interactions between the application and the READEX tool suite in order to handle application level parameters illustrated in Figure 11.

Scenario 1 could lead to unforeseen situations during runtime when the parameter P takes values during run time which haven't been seen at design time. Scenario 2 would avoid this and offload the responsibility to compute the possible range for P to the user. However, both methods are valid for READEX.

5.2.3 Summary and Outlook

The implementation of the Application Parameters switching mechanism is still in the design phase. In the previous section, we outlined a few possibilities with different benefits and drawbacks. The goal for the READEX Tool Suite is to find an implementation which is easy to use and offers the user the flexibility he needs at the same time. We therefore decided to spend more time on designing the switching mechanism. Deliverable 1.2 at the end of Project Month 24 will present a tuning mechanism, which will allow the developer to tune his application according to his needs. Moreover, it will show different results for energy savings we achieved using this mechanism from our partner programs.

6 Summary

In this Deliverable we give an overview about the implementation of the READEX tool suite from the parameter perspective. We described the integration of the parameter control plugins inside the READEX tool suite and give an impression of the interaction between the different components.

Thereafter, we outlined our expectations and the findings about the different parameters. We started with the hardware parameters and outlined why it is worth to further investigate parameters like Dynamic Voltage and Frequency Scaling, Uncore Frequency Scaling or the Energy Performance Bias. Moreover, we showed why we put certain features like dynamic duty cycle modulation or hardware prefetcher aside for the moment.

The system software parameter section described our findings about MPI and OpenMP. Again a short outline of the implementation of tuning plugins and parameter control plugins is given.

Finally, we showed how we would like to provide the user an interface, which allows tuning decisions even in the application space.

The major work on the hardware and system software parameters is done. But as the project is still in an early stage we expect further findings and changes to both of these topics. We have to re-evaluate some hardware and system software parameters once the first prototype of the dynamic READEX tool suite is available. Additionally, it might happen that some of the tuning plugins will be rewritten, according to changing requirements. Also, the application parameters have to be implemented, which might involve huge changes in tools like PTF or Score-P. The final implementation of the application parameters will be outlined in Deliverable 1.2.

References

[1] MPICH. https://www.mpich.org/. Last accessed February 10, 2016.

[2] Openmp application program interface. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[3] Documentation about energy measurement infrastructure at Taurus phase 2. https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/EnergyMeasurement, Last accessed July 13, 2016.

[4] D. Hackenberg, T. Ilsche, J. Schuchart, R. Schöne, W.E. Nagel, M. Simon, and Y. Georgiou. HDEEM: High Definition Energy Efficiency Monitoring. In *Energy Efficient Supercomputing Workshop (E2SC)*, Nov 2014. DOI: 10.1109/E2SC.2014.13.

[5] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. An energy efficiency feature survey of the Intel Haswell processor. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, May 2015. DOI: 10.1109/E2SC.2014.13.

[6] N. Kurd, M. Chowdhury, E. Burton, T P Thomas, C. Mozak, B. Boswell, P. Mosalikanti, M. Neidengard, A. Deval, A. Khanna, et al. Haswell: A family of IA 22 nm processors. *Solid-State Circuits, IEEE Journal of*, 50(1), 2015. DOI 10.1109/JSSC.2014.2368126.

[7] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[8] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995. http://tab.computer.org/tcca/NEWS/DEC95/dec95_mccalpin.ps.

[9] Joshua Peraza, Ananta Tiwari, Michael Laurenzano, Laura Carrington, and Allan Snavely. PMaC's green queue: a framework for selecting energy optimal DVFS configurations in large scale MPI applications. *Concurrency and Computation: Practice and Experience*, 2013. DOI: 10.1002/cpe.3184.

[10] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in FORTRAN; The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA, 2nd edition, 1993.

[11] Robert Schöne and Daniel Molka. Integrating performance analysis and energy efficiency optimizations in a unified environment. *Computer Science - Research and Development*, 29(3-4), 2014. DOI: 10.1007/s00450-013-0243-7.

[12] Anna Sikora, Eduardo César, Isaías Comprés, and Michael Gerndt. Autotuning of mpi applications using ptf. In *Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications*, 2016. DOI: 10.1145/2916026.2916028.

[13] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In Tomasz Imielinski and Henry F. Korth, editors, *Mobile Computing*, volume 353 of *The Kluwer International Series in Engineering and Computer Science*. Springer US, 1996. DOI: 10.1007/978-0-585-29603-6_17.