Towards Automatic Compiler-assisted Performance and Energy Modeling for Message Passing Parallel Programs

Philipp Gschwandtner*, Alexander Hirsch*, Shajulin Benedict*, and Thomas Fahringer†
*Department of Computer Science, University of Innsbruck, Innsbruck, Austria
Email: philipp,alex,tf@dps.uibk.ac.at
†Indian Institute of Information Technology Kottayam, Kerala, India
Email: shajulin@iiitkottayam.ac.in

***Abstract*—Optimizing programs for modern distributed memory parallel architectures is a notoriously difficult task that generated the need for modeling tools that can estimate the execution time and energy consumption for message passing programs. Many prediction tools require substantial manual effort, excessive training for every given architecture or limit the class of input programs that can be handled. We present a compiler-based approach that automatically generates parametrized analytical models. While requiring only a minimum training overhead on target architectures it still provides reasonably accurate models for execution time and energy consumption of message passing programs.**

Our method uses compiler analyses to identify the structure of code regions of input programs, and extracts important parameters such as loop iteration counts or message buffer sizes. We can then predict the performance of these code regions for new problem sizes and target machines.

We show that compiler knowledge can be effectively used to minimize training overhead and evaluate our approach on multiple target applications with varying problem and machine sizes. Initial results obtained with our prototype implementation show a mean coefficient of determination (R^2) of 0.93 over 7 input programs.

***Index Terms*—prediction, performance, energy, static analysis, compiler**

I. Introduction

Optimizing message passing parallel programs is a widespread topic of interest and research, whether done manually by developers or automatically by tools such as compilers or runtime systems. Many optimization and parallelization efforts are guided by performance analysis or prediction. Whereas performance analysis [1] is often dynamic in nature and therefore applicable to wide classes of applications (as it is based on program executions), performance prediction can be invoked statically [2] with a reduced runtime overhead at the cost of a more restricted class of target applications.

For example, modern auto-tuning tuning techniques [3], [4], [5] could greatly benefit from low overhead performance prediction methods to speed up performance evaluation because they commonly invoke large numbers of program executions [6]. Performance prediction is a useful technology to reduce the effort in the search for effective code optimizations.

To be automatic in nature, many prediction approaches — whether incorporating static, analytical information or purely observation-based— themselves rely on a series of program executions to train their models. In this paper, we introduce a novel prediction method that incorporates compiler knowledge about the target application and uses compiler transformations to reduce model training overhead to a single program execution. We consider a large class of iterative message passing parallel programs that follow the bulk synchronous parallel (BSP) and single program multiple data (SPMD) models.

We use the Insieme research compiler and runtime system [7], capable of compiling most C/C++ language constructs, and its INSPIRE intermediate representation to analyze an input program. It extracts static information such as the structure and boundaries of loops, or message sizes of communication primitives. The compiler then invokes a single execution of the input program for a small problem and machine size on a target architecture. Combining the static analysis data with runtime data from this single execution, a parametrized model can be generated to predict execution time and energy for larger problem and machine sizes. The model can be ported to a new target architecture with a single execution of the input program. Additionally, only a handful of offline-measurable hardware parameters are required that specify cache properties, as well as bandwidths and latencies for the memory hierarchy and network.

The major contributions of this work are:

- a definition and automatic localization of target code regions that matches a large number of distributed memory parallel programs,
- a new method for the automatic generation of parametrized performance models for execution time and energy that is problem and machine size sensitive based on compiler analysis and a single program execution, and
- evaluation of this model for several target applications and a wide set of problem and target machine sizes on two hardware architectures.

The paper is structured as follows: Section II lists and compares work related to our method, presented in Section III. Section IV introduces our evaluation methodology and experimental setup, Section V provides results and model output analysis. Finally, Section VI concludes and provides future work.

II. Related Work

There is a plethora of related work in the field of predictive modeling, however we focus on key aspects relevant to our work. First, there are a number of automatic approaches such as *COMPASS* [8], *PEMOGEN* [9], [10], or *Kerncraft* [2]. Like ours, these methods are designed for automatic modeling with little to no user interaction. *PEMOGEN* is of particular relevance since it is an LLVM-based prediction method. Employing a regression approach, it requires a series of training executions of the target program, whereas we target a single training execution only. Moreover, contrary to our work, the main aim of *PEMOGEN* is the reduction of storage costs. There are also semi-automatic approaches such as *PALM* [11] that rely on user annotations for describing models or model parameters. To the best of our knowledge, none of the aforementioned methods have targeted energy consumption prediction. A recent work using *ExaSAT* [12] includes limited energy concerns, however, lacks actual measured energy consumption data. Additional automatic approaches and tools exist, but they focus on hardware characterization [13], low-level assembler prediction [14] or purely sequential programs [15].

By contrast, a number of models focus more strongly on energy prediction, whether built empirically in an analytical fashion [16], regression-based [17] or using machine learning [18], [19]. However, they do not provide automatically derived execution time and energy predictions for message passing programs.

Moreover, there are a number of works that provide predictive models although specifically tailored to aid in aspects such as task aggregation [20], or resilience and reliability [21], [22], whereas we aim for a more generic model design.

Finally, there are a number of invaluable pen-and-paper model works that alone cannot predict execution time for code regions or full programs but aim at characterizing hardware behavior under certain conditions such as *Roofline* [23], [24] or *ECM* [25]. We use their concepts as the foundation for our proposed method as mentioned throughout this work.

III. Model

In order to properly motivate the presented model, we first need to establish several goals that we aim for:

1 build parametrized, analytical models that require only a few constants to be filled in for evaluation, rendering them automatically applicable to a range of iterative distributed memory parallel programs without user interaction,

2 minimize model generation and training overhead (i.e. compilation time) to facilitate the integration of such modeling techniques in existing compilers, and

3 show a proof-of-concept of our approach by applying the presented modeling technique to a range of input programs for multiple machine and problem sizes, and validating all predictions with measurements

We meet Goal **1** by designing our modeling technique from a compiler's point of view, without involving the user in the generation of the parametrized models. Goal **2** can be achieved by partly shifting the requirements of large training data sets to static analysis performed during compilation. The compiler can inspect the input program with regard to properties such as the structure of loops and communication points, loop iteration counts, or message buffer sizes. This allows us to reduce the number of training runs otherwise needed to e.g., obtain supporting points for regression models. We further ensure the compiler considers all known major factors that affect a message passing program's behavior, such as computational boundness, memory hierarchy boundness and communication boundness. Finally, Goal **3** is covered by showing the evaluation of our model using a prototype implementation in a research compiler.

A. Software Model

While our method works on the INSPIRE intermediate representation of the Insieme compiler, we introduce a more compact software model for describing our work here. A target code region is defined as a tree

$$
\begin{aligned}
S &::= \mathit{exp} \\
&\;|\;\; \texttt{for(}\ \mathit{var} = \mathit{exp}\ \texttt{..}\ \mathit{exp}\ \texttt{:}\ \mathit{exp}\ \texttt{)}\ S \\
&\;|\;\; \texttt{f(}\ \mathit{exp}\texttt{,}\ \mathit{exp}\texttt{,}\ \texttt{...}\texttt{,}\ \mathit{exp}\ \texttt{)} \\
&\;|\;\; \texttt{\{}\ S\texttt{;}\ S\texttt{;}\ \texttt{...}\texttt{;}\ S\ \texttt{\}} \\
\mathit{exp} &::= \texttt{a(}\ \mathit{exp}\texttt{,}\ \texttt{\{}\ \mathit{exp}\texttt{,}\ \texttt{...}\texttt{,}\ \mathit{exp}\texttt{\}}\texttt{,}\ \texttt{\{}\ \mathit{exp}\texttt{,}\ \texttt{...}\texttt{,}\ \mathit{exp}\ \texttt{\})} \\
&\;|\;\; \mathit{var} \\
&\;|\;\; \mathit{num}
\end{aligned}
$$

where S is a statement, *exp* an expression, *var* a variable, and a an accessor function. This grammar allows us to form code regions that consist of loops (*for*) with an iterator variable and fixed (but not necessarily known) lower and upper bounds and step expressions, external function calls (*f()*) and compound statements ($\{\ldots\}$). We choose this selection of statements since for loops and external function calls (whose definition is unknown to the compiler) are good candidates for high resource consumption, whereas compound statements allow composition. We also include an accessor function a to model array and pointer subscript expressions, with a base expression (i.e. the pointer or array) and two lists of index expressions as arguments, the first for read operations and the second for write operations. This accessor function aids us in identifying loops with steady-state cache properties (further detailed in Section III-D).

B. Automatic Code Region and Parameter Detection

Figure 1 illustrates the overall architecture, the use of which is briefly outlined here. A more elaborate description of the work performed by each component follows in the remainder of this section. The compiler loads an input program and automatically identifies target regions and derives static analysis information, based on the INSPIRE intermediate representation of the compiler. The program is then instrumented to be

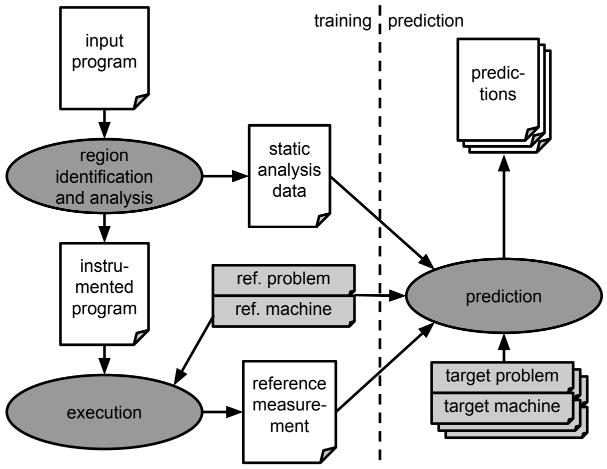

Fig. 1: Workflow of our prediction approach.

executed on the target machine using a small reference problem size and reference machine size. The collected reference measurement data is handed to the predictor in combination with the static analysis information and the reference problem and machine sizes, allowing it to predict the measured metrics for larger problem sizes and machine sizes.

As a first step, we use a pattern-based approach [26] to search the INSPIRE intermediate representation for target code regions at compile-time. For this work, we consider loop nests that meet the following criteria:

- all loops in the nest have fixed (but not necessarily known) lower and upper bounds and increments, and
- there are communication primitives anywhere within the loop nest.

Communication primitives are external function calls, identified by their signature (function identifier and parameter types). This allows easy porting of our region detection to support new communication libraries. For this work, we explore MPI blocking and non-blocking communication primitives. For the latter, we capture both the non-blocking communication operation itself, as well as the corresponding wait function call (identified by the compiler via its matching `MPI_Request*` argument) and treat them as a single unit.

After detecting these target regions, we need to automatically pinpoint their code parameters that are crucial for our model. For loops, these parameters include the bounds as well as the increment expression, allowing us to compute the iteration space of the loop. The parameters of communication primitives include their names (which indicates the type of communication operation they implement), involved source buffers, target buffers, and ranks, as well as size and type of the data transmitted.

C. Automatic Parameter Extraction

The aforementioned parameters of loops and communication primitives have been identified at this stage, but we still need to link them to the input parameters of the entire program for our approach to be fully automatic. Hence, we employ a work-in-progress *compiler integrated* analysis framework that provides various data-flow analyses (DFA) based on the *constraint-based analysis* approach described in [27]. It is capable of inter-procedural analyses, which is essential for fulfilling Goal **1**.

The analysis framework models identified parameters such as loop boundaries, loop step expressions, and communication buffer sizes as symbolic formulas Such a symbolic formula is constructed by traversing the input code from a parameter back to its declaration and keeping track of applied operations and operands. This is done automatically by the compiler without any user interaction required, and yields arithmetic expressions that compute e.g., loop boundaries as a function of given input parameters of the program. The framework computes multiple arithmetic formulas, one for each possible control-flow. Therefore, given the full parameter assignments at prediction time, the control-flow with respect to an identified model parameter must be fully determined. Otherwise, the prediction model cannot pick the correct arithmetic formula from the set returned by the analysis.

Currently, only basic integer arithmetic is supported (i.e. addition, subtraction, multiplication, modulo, and division if the remainder evaluates to zero), while support for floating-point arithmetic is under development. Nevertheless, integer operations are sufficient for our use case of evaluating expressions that compute e.g., grid slice sizes, the MPI ranks of neighbors, and similar parameters.

D. Execution Time Prediction

While our method can be extended to a number of metrics, without loss of generality, we present predictors for execution time here and energy in Section III-E.

To predict a metric for a given target code region, we predict this metric in parametrized fashion for its individual statements and then aggregate the metric data to obtain values for the entire code region. Our execution time prediction follows the idea of the Roofline model [23] and ECM [2] since we attribute individual busy times for all hardware components involved and interpret these results as the lower bound of the overall execution time:

$$T_{\text{all}} = \max\left(\phi(T_{\text{comp}}, T_{\text{mem}}), T_{\text{comm}}\right)$$
$$T_{\text{mem}} = \phi(T_{\text{cache}_0}, T_{\text{cache}_1}, \ldots, T_{\text{RAM}})$$

where ϕ is a metric-specific aggregation function, with $\phi = \max$ for execution time prediction. T_{comp} represents the time a *core* is busy with computation, T_{mem} is the active time of the memory hierarchy (includes caches and RAM) and T_{comm} is the amount of time for which messages are exchanged. Throughout the remainder of this section, we will detail on predicting these metrics.

1) Computation time: For T_{comp} we employ a two-stage approach: First, we collect a single measurement —so-called *reference measurement*— for the previously selected statements in our target code region for a small problem and machine size on a given machine architecture. This is necessary since our work is based on a source-to-source compiler, and therefore we lack any knowledge about to-binary compiler optimizations (e.g., vectorization, software prefetching, etc.) or hardware computing speeds. We can then use static analysis information from the compiler to extrapolate the behavior of these statements for larger target problem and machine sizes. Knowing all loop iteration counts for both the reference and the target problem and machine sizes (obtained via the analysis described in Section III-C), we can compute T_{comp} assuming ideal scaling and hence equal memory hierarchy or communication contention. Naturally, this ignores e.g., increased memory hierarchy traffic for larger problem sizes or increased per-rank cache sizes for larger machines.

2) Memory hierarchy time: Hence, we require a prediction of T_{mem}. Since we want to cover as many applications as possible without limiting the acceptable input structure or syntax (as often required by analytical models), we employ ready-to-use cache simulators to obtain cache miss data and combine these with measured cache bandwidth information to arrive at T_{mem}. However, the large overhead of cache simulators requires us to perform overhead reduction for this approach to be practical.

For this reason, we introduce a new compiler analysis component. Let $\mathbb{S}$ be the set of all statements, then

$$\sigma : \mathbb{S} \to \mathcal{P}(\mathbb{S})$$

is a function that, for a given statement S, returns a set holding all statements contained within S. Second, let S be a statement, then

$$\text{isTimeLoop}(S) = \begin{cases} \text{true}, & \text{if } S = \text{for}(x = l..u : s)\ S' \\ & \text{and } \text{a}(\_, R, W) \notin \sigma(S'), x \in R \cup W \\ \text{false}, & \text{otherwise} \end{cases}$$

is a function which identifies a for loop with $l \le x < u$ and step size s as a *time loop* if the loop iterator x does not appear in any accessor function (i.e. does not occur in any array or pointer subscript) within the body of the loop. We hypothesize that, first, many HPC codes have loops that iteratively compute e.g., physical processes over time. Second, we further hypothesize these loops to have foreseeable cache effects, as the first iteration warms up the CPU caches after which the system is in a steady state (barring any noise from the OS). Therefore, it should be possible to simulate the first few iterations only and extrapolate to the full number assuming a simple linear relationship. Hence, for a time loop with $n > 1$ iterations, its overall misses can be approximated by

$$\sum_{i=0}^{n} M_i \approx (n-1) \cdot \frac{\sum_{i=1}^{n-k} M_i}{n-k-1} + M_0$$

where M_i denotes the number of cache misses induced by loop iteration i, and k is the number of omitted loop iterations. If our assumptions hold, the compiler can transform the target code regions to reduce the number of time loop iterations and yet obtain well-approximated cache miss counts. However, there is a trade-off between large k (causing less overhead) and small k (yielding higher accuracy).

Figure 2 shows the results of evaluating this hypothesis with Cachegrind for decreasing k for a naïve *jacobi* implementation. We simulated the L3 cache of an Intel Xeon E5-4650, and chose two problem sizes, one smaller and one larger than the cache size. Our expectations are confirmed, as using only a single time loop iteration ($k = n - 1$) is not sufficient for problem sizes that fit in the cache (yielding a relative error of $1.27 \cdot 10^2$), however increasing the number of iterations beyond 2 ($k = n - 2$) does not amortize the increase in overhead, with a relative error of already only $-5.47 \cdot 10^{-3}$. Since we verified similar behavior for all our input programs, we always choose $k = n - 2$ for this work.

However, there are additional opportunities for decreasing overhead. Since our input programs follow the BSP/SPMD models, we can simulate a single MPI rank and extrapolate to the full target machine, reducing simulation time by a factor of the target machine size. As it is impossible to execute arbitrary MPI programs with a single rank, the communication primitives require special treatment — we want to remove them while preserving the cache behavior of the primitives and the surrounding code region (e.g., buffer reuse for computation and communication). For this reason, we define a compiler transformation function as follows (for brevity we only show the semantics of send and receive primitives):

$$\text{transformCommPrimitive}(S) = \begin{cases} \text{for}(x = 0..s : 1)\ \ \text{a}(b, \{x\}, \{\}), & \begin{array}{c} \text{if } S = \\ f(\_, b, s, \_, \_, \_, \_) \\ \text{and } f = \text{MPI\_Send} \end{array} \\ \text{for}(x = 0..s : 1)\ \ \text{a}(b, \{\}, \{x\}), & \begin{array}{c} \text{if } S = \\ f(\_, b, s, \_, \_, \_, \_, \_) \\ \text{and } f = \text{MPI\_Recv} \end{array} \\ \vdots & \\ S, & \text{otherwise} \end{cases}$$

It replaces all communication primitives with corresponding linear reads and writes of buffer b as they are expected to occur in the actual library calls.

Furthermore, since our input programs are homogeneous in their workload but lack application data sharing (contrary to e.g., OpenMP programs), we hypothesize that shared cache effects can be simulated by reducing the available shared cache size per core by the factor of the number of cores sharing this cache and participating in the computation. Figure 3 confirms this theory, as the predicted T_{mem} of the target code region matches the actual execution time when the code region is memory bound (machine sizes 8–64).

It should be noted that one could also think of simply executing the transformed application on the target hardware instead of executing it in a cache simulator for performance reasons. However, we aim at minimizing the use of the target hardware (a key motivation of predictive modeling to begin with). Additionally, using real hardware would remove the

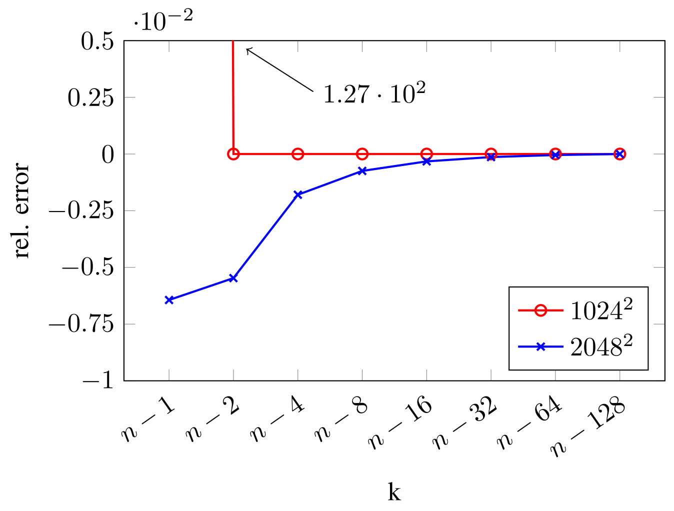

Fig. 2: Relative L3 miss prediction error for *jacobi* for $n = 128$ and decreasing k.

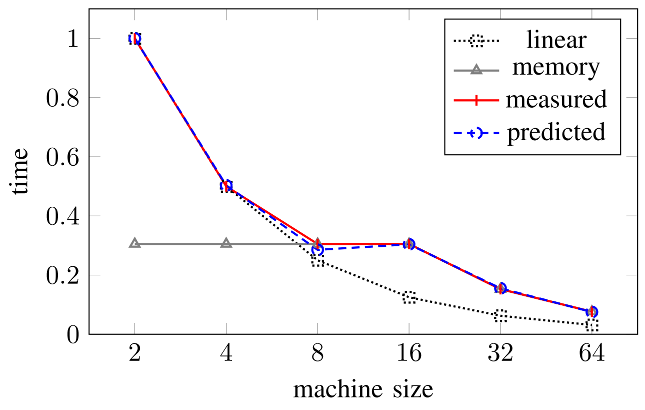

Fig. 3: Time prediction details of *jacobi* for $N = 32768$ for increasing machine sizes

possibility of simulating the effect of reduced cache sizes and hardware-specific performance counters would be required to ascertain e.g., the number of cache misses. Contrary to that, we only rely on information that can be obtained automatically via the `cpuid` instruction (cache levels, cache sizes, line sizes, associativities). Finally, using a cache simulator allows us to evaluate our model for multiple target problem and machine sizes in parallel without the risk of measurement perturbation, increasing model prediction throughput.

The compiler also confines cache simulation to the target code regions by inserting control statements that start the cache simulation only prior to execution of the first target regions, and exit the input program after the last one. The final cache miss data is then combined with cache and memory bandwidth information (obtained via offline measurements, once per target architecture) to compute T_{mem}.

3) Communication time: Finally, we need to predict T_{comm}, which we only outline due to spatial constraints. The compiler extracts arithmetic formulas for source and target ranks as well as message sizes from communication primitives in the target code region. We require these to depend only on integer operations as described in Section III-C and program parameters, `MPI_comm_rank`, and `MPI_comm_size`. We then examine these formulas for popular communication patterns such as neighbor exchange. Collective operations can also be handled assuming their communication pattern is known for a given message and rank size [28]. We use this information in combination with bandwidth and latency information (to be measured once per target architecture) and a given rank-core mapping policy to compute the data transfer time T_{comm}.

4) Aggregation: At this stage, we predicted T_{comp}, T_{mem}, T_{comm} and hence T_{all} for individual statements. To get predictions for the entire target code region, we require a recursive aggregation function. Let A be the type of the metric to be predicted and *eval* an evaluation function for arithmetic expressions. Then

$$\text{get}(S) = \begin{cases} \text{data}, & \text{if data available} \\ p(S), & \text{otherwise} \end{cases}$$

$$p(S) = \begin{cases} \text{get}(S') \cdot \text{eval}(\frac{u-l}{s}), & \text{if } S = \text{for}(x = l..u : s)\ S' \\ \sum_{i=0}^{n} \text{get}(S_i), & \text{if } S = \{S_0, S_1, \dots, S_n\} \\ 0 & \text{otherwise} \end{cases}$$

are functions retrieving and aggregating data for loops and compound statements, returning the identity element (0 for this work) for all other cases.

E. Energy Prediction

Energy prediction is done similarly to the one described in Section III-D, except that we use energy measurements as the reference, and we choose $\phi = \sum$ to express each hardware component's contribution to the overall energy consumption (contrary to busy times of hardware components, which can overlap):

$$E_{\text{all}} = \sum \Big(\sum (E_{\text{computational}}, E_{\text{memory}}), E_{\text{communication}} \Big)$$

while for each hardware component x we consider

$$E_x = \overline{P}_{x,\text{idle}} \cdot T_{x,\text{idle}} + \overline{P}_{x,\text{load}} \cdot T_{x,\text{load}}$$

$$\overline{P}^{\text{ref}}_{x,\text{load}} = \frac{E^{\text{ref}}_{x,\text{load}}}{T^{\text{ref}}_{x,\text{load}}} \qquad T_{x,\text{idle}} = \begin{cases} T_{\text{all}} - T_x, & \text{if } T_x < T_{\text{all}} \\ 0, & \text{otherwise} \end{cases}$$

where $\overline{P}_{x,\text{idle}}$ is the average idle power consumption of hardware component x (to be measured offline, once per target architecture), $T_{x,\text{load}}$ is obtained via prediction as described in Section III-D, and $T^{\text{ref}}_{x,\text{load}}$ and $E^{\text{ref}}_{x,\text{load}}$ are execution time and energy consumption of our reference measurement. Note that this implies that $\overline{P}_{x,\text{load}} \approx \overline{P}^{\text{ref}}_{x,\text{load}}$. We then predict $\overline{P}^{\text{target}}_{x}$ of hardware component x of reference machine M and a target machine M' as

$$\overline{P}^{\text{target}}_{x} = \frac{\overline{P}^{\text{ref}}_{x}}{|\{y \in M | y = x\}|} \cdot |\{y' \in M' | y' = x\}|.$$

TABLE I: Machine characteristics.

name	nodes	CPUs per node	cores per node	cache sizes	RAM	OS	compiler	MPI/network
ortlerSandy	4	4x E5-4650 2.7 GHz	32	priv.: 32 KB, 256 KB, shared: 20 MB	256 GB	CentOS 6.7, 2.6.32-573	gcc 5.1 -O3	Open MPI 1.10.2 on Gigabit Ethernet
ortlerIvy	4	2x E5-2690 v2 3.0 GHz	20	priv.: 32 KB, 256 KB, shared: 25 MB	128 GB	CentOS 6.5, 2.6.32-431		

TABLE II: Input programs, properties and problem sizes.

program	description	comp.	memory.	iter.	problem sizes (S: ortlerSandy, I: ortlerIvy)
cg	conjugate gradient	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	1000	S: 1024 2048 3072 4096 5120 6144 7168 8192 I: 1040 2080 3120 4160 5200 6240 7280 8320
homb	laplace solver	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	100	S: 1024 2048 4096 8192 16384 32768 I: 960 1920 3840 7680 15360 30720
jacobi	2d jacobi solver	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	128	S: 1024 2048 4096 8192 16384 32768 I: 960 1920 3840 7680 15360 30720
mm_ijk	matrix multiplication, ijk loop order	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$	50	S: 448 640 960 1280 1600 1920 2240 2560 2880 I: 400 600 800 1000 1200 1400 1600 1800 2000
mm_ikj	matrix multiplication, ikj loop order	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$	50	S: 448 640 960 1280 1600 1920 2240 2560 2880 I: 400 600 800 1000 1200 1400 1600 1800 2000
shs	simple hyperbolic solver	$\mathcal{O}(N^2)$	$\mathcal{O}(N^2)$	40	S: 128 256 512 1024 2048 4096 I: 80 160 320 640 1280 2560
stencil3d	generic 3x3x3 3d stencil	$\mathcal{O}(N^3)$	$\mathcal{O}(N^3)$	100	S: 128 256 384 512 640 I: 160 240 320 400 480

Using the aggregation formula of Section III-D4, we are able to predict E_{all} for a given target code region.

IV. Experimental Setup

We implemented a prototype of our method in C++ as part of the Insieme research compiler and runtime system [7]. This research compiler provides all facilities required to analyze an input program, transform it, and generate an instrumented version. The runtime system then executes the instrumented application and feeds back all measured data to the compiler. To increase prediction throughput of the model, our prototype implementation relies on std::async for simultaneous prediction of multiple problem and machine sizes.

The experimental testbed used for our experiments consists of two distributed memory machines named *ortlerSandy* and *ortlerIvy*, Table I lists their characteristics. The CPU clock frequency was fixed as listed in Table I, and Hyperthreading was disabled on all machines. The nodes are connected via a dedicated Gigabit Ethernet network. We enforced process binding to cores, with a mapping that uses at least 2 nodes with one core each (machine size 2), and then increasing first the number of cores on sockets already in use before employing new sockets. Similarly, new nodes are only added when all current sockets are fully utilized.

Measurements were obtained via x86's *rdtsc* instruction for execution time and Intel's *RAPL* interface for energy consumption. The latter offers a data resolution of 15.3 microjoules and time resolution of 1 millisecond, and related work has shown it to be accurate enough for our purpose [29]. Since *RAPL* only captures the CPU package, we present energy prediction results of the CPUs. Note that our method does not specifically rely on RAPL or Intel-based micro-architectures, but works with any energy instrumentation infrastructure that provides the desired accuracy. However, for availability reasons, we employ Intel hardware. The cache simulator in use for cache miss prediction is Cachegrind 3.11 [30].

A selection of input programs for our work, their basic properties as well as tested problem sizes are listed in Table II. Since our work is based on a research compiler, we employ such representative proxy apps instead of full-sized applications for validation. *cg* is an iterative conjugate gradient solver, *homb* [31] the Hybrid OpenMP MPI Benchmark, *jacobi* a 2D jacobi solver, *shs* [32] the Simple Hyperbolic Solver, and *stencil3d* a generic 3x3x3 3D stencil. We also include two matrix multiplication kernels, *mm_ijk* and *mm_ikj*, which exhibit substantially different cache behavior due to their respective loop orders. All of these codes are written in C and rely on MPI for parallelism (*homb* also offers OpenMP parallelism, which we disabled for this work). We do not consider workloads stressing hardware components such as the IO subsystem of a parallel computer, as we lack the capabilities for measuring the energy consumption of the respective hardware components.

To validate the model, we predict time and energy for multiple target problem and machine size parameter combinations and verify our predictions with measurements. The tested machine sizes have been selected with respect to the available resources in our experimental testbed, and the selection of input program problem sizes reflect execution times in the order of several minutes. To reduce any inaccuracy caused by external load such as the OS, all reported measurement data represents the median over 5 runs. For predicted data however, since our model is fully deterministic, a single run is sufficient. We furthermore evaluate the accuracy of the model by computing the normalized root-mean-square error (NRMSE) and the coefficient of determination (R^2).

V. Results

To illustrate and analyze the predictions, we focus on the first of our input programs, a simple 2D Jacobi (*jacobi*), since it is well-studied and shows all our considered aspects of mod-

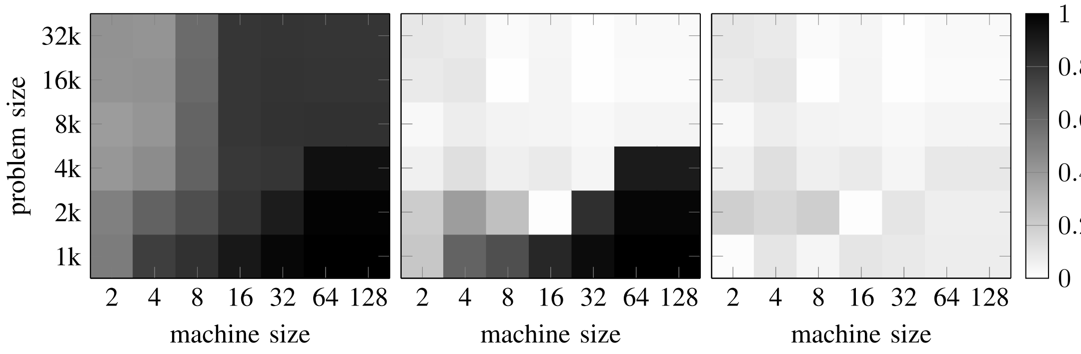

Fig. 4: Relative error of the execution time prediction of the *jacobi* application for increasing problem and machine sizes on *ortlerSandy* for T_{comp} (left), $\max(T_{\text{comp}}, T_{\text{mem}})$ (center), and $\sum(\max(T_{\text{comp}}, T_{\text{mem}}), T_{\text{comm}})$ (right).

TABLE III: Overall model errors.

	ortlerSandy				ortlerIvy			
	time		energy		time		energy	
code	NRMSE	R^2	NRMSE	R^2	NRMSE	R^2	NRMSE	R^2
cg	0.018	0.978	0.038	0.928	0.025	0.984	0.028	0.979
homb	0.035	0.955	0.047	0.929	0.008	0.996	0.045	0.947
jacobi	0.020	0.980	0.068	0.905	0.014	0.995	0.091	0.849
mm_ijk	0.044	0.929	0.080	0.812	0.068	0.856	0.078	0.840
mm_ikj	0.025	0.980	0.053	0.931	0.031	0.975	0.076	0.893
shs	0.082	0.945	0.068	0.947	0.060	0.971	0.058	0.932
stencil3d	0.053	0.940	0.092	0.839	0.016	0.995	0.064	0.921
mean	0.040	0.950	0.064	0.899	0.032	0.967	0.063	0.909

eling distributed memory parallel applications. Subsequently, we will show results for all other input programs.

Figure 4 presents the results of predicting execution time for *jacobi* with a reference problem size of 1024 and a reference machine size of 2 (two nodes with one core each, as per our mapping policy detailed in Section IV) on *ortlerSandy*. The shading denotes the relative error of predicting T_{comp}, $\max(T_{\text{comp}}, T_{\text{mem}})$, and $\sum(\max(T_{\text{comp}}, T_{\text{mem}}), T_{\text{comm}})$ compared to actual measurements, and illustrates the incremental increase in accuracy for each prediction step added. The shapes indicate a mainly memory-hierarchy-bound program, with prediction of only T_{comp} yielding a mean relative error of 0.69. The memory contention is evident by the visible column-like separation of machine sizes 2–4, 8 and 16–64, due to the fact that *jacobi* is already memory bound at machine size 8 (as also indicated by Figure 3) for problem sizes larger or equal to 4096 (resulting in a working set of 32 MB for 20 MB of L3 cache on *ortlerSandy*). Including the prediction of T_{mem} already substantially improves accuracy for these cases, lowering the overall mean error to 0.25. However, a number of cases with small problem but large machine sizes are naturally not predicted well, as communication time contributes a major part of T_{all} here. Including T_{comm} in our prediction also covers these cases, lowering the overall mean error to 0.06.

Table III presents the overall results for time and energy for all input programs over all problem sizes on both machines. As the data illustrates, our prediction generally achieves higher accuracy across all input programs for execution time (mean R^2 of 0.95) compared to energy consumption (mean R^2 of 0.90). This is a result of the mapping of our ϕ function, partially described by the differences between Roofline [23] and ECM [25]. When predicting the execution time and thus aggregating the maximum over multiple sub-predictions, only the largest element directly impacts the result, provided the relative order of the sub-predictions is correct. In contrast, energy consumption is aggregated as the sum over all sub-predictions, requiring high-quality predictions for all of them for high overall accuracy.

The highest error case for our method is *mm_ijk*, explained by its expensive column-wise matrix traversal. The results for *mm_ikj* confirms this, performing better without this expensive traversal. While Cachegrind is likely more precise than many analytical models, it is limited to assuming idealized caches without noise, only considers two cache levels, an LRU replacement policy, and lacks advanced knowledge about hardware prefetching. Furthermore, the results on *ortlerIvy* do not always correlate with the ones obtained on *ortlerSandy*. Apart from different memory controller speeds and CPU clock frequencies, the former holds CPUs with 25 MB of 20-way associative L3 cache — a parameter combination that results in set sizes other than powers of two, which Cachegrind cannot simulate. For these reasons, we simulate the first and last level cache, and as a fallback switch to 25-way associativity whenever required. Overall, our method achieves a mean NRMSE of 0.036 for execution time and 0.064 for energy consumption across all benchmarks and both architectures.

VI. Conclusion

We presented a novel compiler-based prediction method that automatically generates models for execution time and energy for a large set of message passing parallel programs. We introduced a code region definition matching the structure of these programs, and illustrated the benefits of using compiler analysis for deriving analytical models and minimizing model generation overhead. We demonstrated that a single reference execution per input program and target architecture is sufficient for training. Our prototype implementation showed the validity of our approach, with a mean coefficient of determination of 0.93 over 7 input programs. Future work includes examining the prediction accuracy with regard to varying the reference problem and machine sizes, improving cache miss prediction, supporting derived data types in MPI, and exploring new hardware architectures and input programs.

Acknowledgment

This project was funded by the FWF Austrian Science Fund as part of the project I 1523 "Energy-Aware Autotuning for Scientific Applications".

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[2] J. Hammer, G. Hager, J. Eitzinger, and G. Wellein, "Automatic Loop Kernel Analysis and Performance Modeling with Kerncraft," in *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*. ACM, 2015, p. 4.

[3] R. Suda, K. Naono, K. Teranishi, and J. Cavazos, *Software Automatic Tuning: Concepts and State-of-the-Art Results*. New York, NY: Springer New York, 2010, pp. 3–15.

[4] P. Gschwandtner, J. J. Durillo, and T. Fahringer, "Multi-objective Auto-tuning with Insieme: Optimization and Trade-off Analysis for Time, Energy and Resource Usage," in *Euro-Par 2014 Parallel Processing*. Cham: Springer International Publishing, 2014, pp. 87–98.

[5] K. Kofler, J. J. Durillo, P. Gschwandtner, and T. Fahringer, "A Region-aware Multi-objective Auto-tuner for Parallel Programs," in *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, Aug 2017, pp. 190–199.

[6] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A Scalable Auto-tuning Framework for Compiler Optimization," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–12.

[7] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A Multi-objective Auto-tuning Framework for Parallel Codes," in *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. Washington, DC, USA: IEEE, 2012, pp. 1–12.

[8] S. Lee, J. S. Meredith, and J. S. Vetter, "COMPASS: A Framework for Automated Performance Modeling and Prediction," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 405–414.

[9] A. Bhattacharyya and T. Hoefler, "PEMOGEN: Automatic Adaptive Performance Modeling During Program Runtime," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 393–404.

[10] A. Bhattacharyya, G. Kwasniewski, and T. Hoefler, "Using Compiler Techniques to Improve Automatic Performance Modeling," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct 2015, pp. 468–479.

[11] N. R. Tallent and A. Hoisie, "Palm: Easing the Burden of Analytical Performance Modeling," in *Proceedings of the 28th ACM international conference on Supercomputing*. ACM, 2014, pp. 221–230.

[12] D. Unat, C. Chan, W. Zhang, S. Williams, J. Bachan, J. Bell, and J. Shalf, "ExaSAT: An Exascale Co-design Tool for Performance Modeling," *International Journal of High Performance Computing Applications*, vol. 29, no. 2, pp. 209–232, 2015.

[13] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, "Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Cham: Springer International Publishing, 2015, pp. 129–148.

[14] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby, "MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2," in *Workshop on Explicitly Parallel Instruction Computing Techniques*, Santa Jose, CA, USA, 2005.

[15] S. H. K. Narayanan, B. Norris, and P. D. Hovland, "Generating Performance Bounds from Source Code," in *2010 39th International Conference on Parallel Processing Workshops*, Sept 2010, pp. 197–206.

[16] T. Rauber and G. Rünger, "Modeling the Energy Consumption for Concurrent Executions of Parallel Tasks," in *Proceedings of the 14th Communications and Networking Symposium*. San Diego, CA, USA: Society for Computer Simulation International, 2011, pp. 11–18.

[17] S. Pakin and M. Lang, "Energy Modeling of Supercomputers and Large-scale Scientific Applications," in *2013 International Green Computing Conference Proceedings*, June 2013, pp. 1–6.

[18] A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely, "Modeling Power and Energy Usage of HPC Kernels," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, May 2012, pp. 990–998.

[19] S. Benedict, R. S. Rejitha, P. Gschwandtner, R. Prodan, and T. Fahringer, "Energy Prediction of OpenMP Applications Using Random Forest Modeling Approach," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, ser. IPDPSW '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 1251–1260. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2015.12

[20] D. Li, D. S. Nikolopoulos, K. Cameron, B. R. de Supinski, and M. Schulz, "Power-aware MPI Task Aggregation Prediction for High-end Computing Systems," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010, pp. 1–12.

[21] B. Mills, T. Znati, R. Melhem, K. B. Ferreira, and R. E. Grant, "Energy Consumption of Resilience Mechanisms in Large Scale Systems," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2014, pp. 528–535.

[22] Z. Zheng, L. Yu, and Z. Lan, "Reliability-aware Speedup Models for Parallel Applications with Coordinated Checkpointing/Restart," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1402–1415, May 2015.

[23] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[24] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A Roofline Model of Energy," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp. 661–672.

[25] H. Stengel, J. Treibig, G. Hager, and G. Wellein, "Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 207–216.

[26] H. Jordan, P. Thoman, and T. Fahringer, "A High-level IR Transformation System," in *Euro-Par 2013: Parallel Processing Workshops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 647–656.

[27] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 1999.

[28] T. Hoefler and D. Moor, "Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations," *Supercomput. Front. Innov.: Int. J.*, vol. 1, no. 2, pp. 58–75, Jul. 2014.

[29] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring Energy Consumption for Short Code Paths Using RAPL," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 3, pp. 13–17, 2012.

[30] N. Nethercote, "Dynamic Binary Analysis and Instrumentation," Ph.D. dissertation, PhD thesis, University of Cambridge, 2004.

[31] M. L. Hutchinson, "Hybrid OpenMP MPI Benchmark," https://sourceforge.net/projects/homb/, Apr 2013, accessed: Oct 7, 2016.

[32] L. Arnold, "IBM BG/P workshop," http://www.training.prace-ri.eu/uploads/tx%5Fpracetmo/BlueGeneP.pdf, Oct 2009, accessed: Oct 7, 2016.