Making MPI Intelligent

Dirk Tetzlaff, Sabine Glesner
Chair Software Engineering for Embedded Systems
Technische Universität Berlin, Germany
dirk.tetzlaff@tu-berlin.de
sabine.glesner@tu-berlin.de

Abstract: Mapping parallel applications to multi-processor architectures requires information about the execution times of the concurrent processes to find an optimal allocation and must take into account the interprocessor communication at runtime, whose overheads have emerged as the major performance limitation. However, both information cannot be statically known in advance. In this paper we present a sophisticated approach for mapping parallel MPI applications to concurrent architectures using *machine learning* techniques. This automatically generates heuristics that provide the compiler with knowledge of the considered runtime behavior, hence yielding more precise heuristics than those generated by pure static analyses. The heuristics can be used to direct the runtime environment of MPI, which enables the reallocation of processes to other processors at runtime and, furthermore, results in a better initial allocation of MPI processes.

1 Introduction

The allocation of processes to processing elements (*PEs*) is a vital part of mapping parallel applications to concurrent architectures. Finding an optimal solution requires information about execution times of processes during runtime of applications, which is not known in advance at compile time. Furthermore, it must take into account interprocessor communication at runtime, whose overheads have emerged as the major performance limitation in parallel applications. Hence, one has to predict loop iteration counts if interprocessor communication arises within loop bodies that do not have statically determinable loop bounds. However, static analyses cannot predict iteration counts precisely. Consequently, it is necessary to automatically incorporate knowledge of related dynamic behavior into the compiler but without the need for manual annotations or profiling to preserve an automatic, continuous and efficient compilation flow. Moreover, the technique should be highly scalable to facilitate the integration in industrial-strength compiler environments used for compilation of real-world applications.

We tackle the problem of incorporating statically unknown or imprecise information about the execution time and communication overhead during runtime of applications into the compilation flow with the use of *machine learning* (*ML*) techniques, especially *supervised classification learning*. Our approach thus automatically generates heuristics that efficiently provide the compiler with knowledge of runtime behavior, namely to predict loop iteration counts and execution times of processes. Our experimental results for learning the loop iteration count can be found in [TG10].

In this paper, we focus on mapping MPI programs to concurrent architectures. We define how the runtime behavior predictions are used to allocate MPI processes to PEs and we present in detail the static code features used for learning the loop iteration count. Using our machine learned information for the mapping of MPI processes to concurrent architectures improves the initial allocation because processes that communicate most frequently with each other can be automatically allocated to PEs with the minimal communication latency in between. Furthermore, it enables the reallocation of processes at runtime whenever their communication behavior changes. Both features are great improvements to current MPI implementations because even the best MPI implementation cannot ensure that for instance frequently communicating processes are placed close to each other [Trä02].

While the learning phase entails a significant overhead, it is only executed once per architecture decoupled from the compilation, so the compile time for applications is not increased. To obtain the training data, we perform several profiling runs with different input for the programs. This data is condensed and abstracted with classification learning that relate static properties of an application to its dynamic behavior at runtime. This bridges the gap between static program analyses on the one hand and dynamic program behavior on the other. Because we use a concise representation (decision trees), the obtained predictors can be efficiently implemented (the resulting code consists of nested `if-else` statements). Due to the high scalability of the resulting heuristics, the additional compile-time overhead is negligible and lower than those of conservative program analyses. Thus, our approach preserves an efficient compilation flow. Since the heuristics are automatically generated and incorporated into the compiler without the need for user intervention, our approach also preserves an automatic, continuous compilation flow. Note that the generated heuristics can also be combined with static program analyses. When the analysis knows the results to be exact, that result can be used, otherwise, the heuristics is consulted. Thereby, our approach combines the benefits of static analyses and profiling without taking their disadvantages.

The paper is organized as follows: First, we outline the background of our approach in Section 2. Our approach to ML-based mapping is given in Section 3 and its implementation in Section 4. Then, we discuss related work in Section 5. Finally, conclusions are drawn in Section 6.

2 Background

In this paper, we contribute to a novel research domain, namely the prediction of runtime behavior to facilitate optimizations during compilation by applying machine learning techniques. In the following, we provide the necessary background knowledge of both areas, machine learning in Section 2.1 and compiler optimizations in Section 2.2. Because we aim at mapping MPI processes to concurrent architectures, we give an overview of MPI in Section 2.3.

2.1 Machine Learning

Machine learning can be used to automatically infer information from a series of observations. It has been central to artificial intelligence research from the beginning [Tur50]. Here, we consider classification learning for which several algorithms have been proposed. One of the most popular methods is classification tree learning [Ste09]. It has become popular because of its clear interpretation and its ability to provide a good fit in many cases [GLM04]. The advantage compared to other methods like *neural networks* (see [Wan03]) or *nearest neighbors* (see recent survey paper [Ind04]) is that the constructed predictors have concise representations (decision trees). Thus, once trained, making predictions takes a negligible amount of time. Moreover, knowledge extraction from decision trees for explanation about the classification process is provided. That is, the decision tree can be inspected to determine which features are important for classification.

For classification learning, each observation is described by its properties, or features, and is categorized concerning a set of given classes. The aim is to build a model that best explains the relationship from features to classes for the training data. The details of building the model via recursive partitioning can be found in [Alp10]. Once trained, the model can also be used to classify new observations, based on their features. Especially, it can be used to generate an executable heuristics. This provides the compiler with intelligence about the dynamic program behavior and can be used for a precise cost estimation. The idea of the heuristics is to focus on typical program behavior instead of considering all possibilities, as it is done by most static analyses.

To evaluate the precision of a predictor, it is applied to data for which the correct classes are known. Then, correct and predicted classes can be compared. For realistic results, predictors should be applied to unseen data for evaluation. As precision measure, the average deviation between predictions and correct classes can be used (mean absolute error). Additionally, considering the correlation between predictions and correct classes helps to decide whether a relationship was actually learned.

2.2 Code Generation

Compilers for high-level programming languages convert the source code to an *intermediate representation* (*IR*), which typically is transformed by subsequent optimizations whereby the semantics of the program has to be preserved. Finally, the IR is translated into machine code. For compilers, program analyses are vital to identify optimization opportunities and to ensure that the program behavior is not changed. Since these analyses are static, they can only make assumptions about how the program may behave at runtime. The obligation to consider every possible eventuality enforces the analyses to over-approximate, which can lead to highly imprecise results. As a consequence, the optimizations are limited and optimization potential is sacrificed.

However, in many cases, programs do not behave as bad as the compiler expects. Hence, gathering information about the runtime behavior, called *profiling*, can be used to focus optimization efforts on realistic behavior. That is, the program has to be compiled first without runtime information. Then, the binary must be executed with typical input data, and afterwards gathered information is available to direct optimizations. As a result, profile-guided compilation is able to achieve noticeable performance improvements, though having the penalty to re-compile the program after its execution. In general, it works well only when the actual runtime tendencies of the program match those collected during profiling [Smi00]. That is, profiling is strongly dependent on the input data set of that profiling run.

Integrating a beforehand machine learned model into the compiler that holds observations from several profiling runs and relates them to static code features, as we do, eliminates the need for profiling at compile time and has the advantage to focus optimizations on more realistic behavior, not on one single behavior. Thus, we can make a more educated guess whether some transformation may be beneficial to motivate compiler optimizations.

2.3 Message-Passing Interface (MPI)

The Message-Passing Interface (*MPI*) standard is an interface specification of a message-passing library. An MPI program consists of autonomous processes which are identified with a unique ID, called *rank*. Each process is an instance of the same MPI program, executing its own code in an MIMD style. That is, the code executed by each process does not need to be identical. Which portion of this program will be executed by a process can be selected via the rank. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible [Mes09].

The processes communicate via calls to MPI communication primitives like `MPI_Send` and `MPI_Recv`, which are *blocking* operations. In other words, the operations do not return until the message data is stored either into the matching receive buffer or into a temporary system buffer. Message buffering decouples the send and receive operations. A blocking send can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver. There are also *synchronous* communication operations, for instance `MPI_Ssend` and `MPI_Srecv`. The synchronous send will complete only if the corresponding receive operation has started to receive the message. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. Other synchronization calls exist, where a number of processes perform a rendezvous, for instance `MPI_Barrier` or `MPI_Win_fence`.

To start an MPI program, the user has to specify how many processes shall execute the program. The *MPI runtime daemon* then starts this number of processes and is responsible for the communication data transport and the coordination between the processes. The user can specify to which processing element (*PE*) a process will be allocated, but this allocation is fixed even if the runtime behavior indicates that reallocation to another PE would be beneficial. If no advice for the allocation is given, the MPI runtime daemon starts the processes in a Round Robin fashion on available PEs.

3 ML-based Mapping of MPI-Processes

Our approach for learning loop iteration counts and execution times of processes automatically generates classifiers, which relate the static code features to the dynamic behavior, thus providing the compiler with intelligence. At compile time, these classifiers can be used as precise and fast heuristics for communication-aware mapping of MPI processes to a concurrent architecture.

In the following, we outline the two phases of our approach for mapping MPI processes to concurrent architectures using ML-based runtime-behavior predictions, namely the training phase in Section 3.1 and the compilation phase in Section 3.2.

3.1 Training

In the one-off training phase per architecture, we collect code features by means of static analysis as well as dynamic runtime behavior through several profiling runs from a comprehensive suite of programs. Since each program is profiled with different input data, we collect realistic behavior which is less strongly input data dependent. Both information is fed into a learning algorithm, which is intended to find a statistical model (or more precisely, a collection of classifiers) that represents the relationship of static features to dynamic behavior with minimum error (see Figure 1). Since we are looking at supervised classification learning, the considered dynamic behavior has to be discretized to a set of classes.

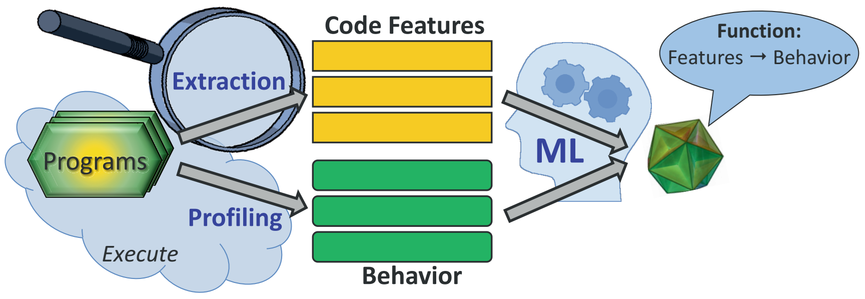

Figure 1: Training phase

The learning algorithm strives for two goals that are discussed in detail below. First, in Subsection 3.1.1, we consider learning loop iteration counts to determine the amount of interprocessor communication. Second, in Subsection 3.1.2, we employ ML techniques to predict precisely the execution time of a process on a particular processing element (*PE*) in order to map processes efficiently to concurrent architectures.

3.1.1 Learning Loop Iteration Counts

To learn statically undeterminable loop iteration counts, we consider static code features, which hold characteristics of the loop body, the loop bounds, and the function that contains the loop. In total, we use 114 code features (see Table 1 for a complete list of features).

For example, we consider code features such as the kind of the loop and the loop exit branches. We distinguish as kind of loops between `do-while` loops and `while-do` or `for` loops because the former ones iterate at least once. We consider the kind of exit branches to differentiate between exits that stem from the loop bound, exits that stem from `break` statements, and exits that arise from `goto` statements. We also consider the number of exit branches out of the loop because the more exit branches a loop contains (e.g., due to `continue` or `break` statements), the greater the likelihood that the loop iteration is actually terminated. If arrays are referenced within a loop body, the loop can be expected to iterate over all elements. We therefore consider the maximum size of the referenced arrays as a static feature.

We also examine characteristics of the structure of loop bounds like, for instance, the number of con- and disjunctions and (un-)equality comparisons between pointers or values. The more conjunctions a loop bound contains at the top level, the greater the likelihood that the loop will iterate less often. Parallel to this, the more disjunctions a loop bound contains at the top level, the greater the probability that the loop will iterate more often. Assuming a typical distribution of data values, the probability of two data values being equal is low. Hence, a comparison of values or pointers as to whether they are equal (or unequal) within a loop bound will probably result in more (or less) iterations. As characteristics of the function that contains the loop, we use the cyclomatic complexity [McC76] and the nesting depth [GS85]. Since the static prediction of execution frequency proposed by Wu and Larus [WL94] is a good estimation of how often a loop will execute, it is used to weight several features. Our experimental results for learning the loop iteration count can be found in [TG10].

3.1.2 Learning Execution Times

We propose learning the execution time of a process based on static code features such as the latency of the most probable path, the weighted sum of latencies of all instructions, the fraction of control instructions, loop-related features and the amount of interprocessor communication. The rationale is the following. The most probable path, which can be determined by static profile analysis [WL94], defines the likeliest runtime behavior. We use the predicted execution probabilities to also weight the sum of the latencies of instructions. Note that this latency is architecture-dependent, but every compiler has to know it

No.	Name	Description
1 – 2	kind-{struc,exit}	kind of loop structure and exit branches
3	cnt-exit	number of exit branches
4 – 9	{min,max}-sz-{arr,rec,un}	min. / max. size of referenced arrays, records, unions
10 – 15	{min,max}-bsz-{arr,rec,un}	ditto when referenced in loop bound
16 – 18	cnt-{arr,rec,un}	number of accesses to elements of arrays, records, unions
19 – 20	nest, max-nest	nesting of current loop, max. nesting of contained inner loops
21 – 23	cnt-{or,and,log}	number of disjunctions, conjunctions, and logical terms in loop bound
24 – 35	cnt-{i,f}-{l,le,g,ge,eq,ne}	number of comparisons of integer / floating points to be less / less or equal / greater / greater or equal / equal / not equal
36 – 47	cnt-{iz,fz}-{l,le,g,ge,eq,ne}	ditto when compared to zero
48 – 59	cnt-{im,fm}-{l,le,g,ge,eq,ne}	ditto when compared to minus one
60 – 71	cnt-{ic,fc}-{l,le,g,ge,eq,ne}	ditto when compared to another constant
72 – 77	cnt-ptr-{l,le,g,ge,eq,ne}	ditto when pointer are compared
78	cnt-ptr-nil	number of comparisons of pointer against `NULL`
79 – 86	cnt-{f}{scanf,printf,getc,putc}	number of calls to these functions
87 – 94	w-{f}{scanf,printf,getc,putc}	ditto weighted by static execution frequency
95 – 97	cnt-{fopen,fclose,fflush}	number of calls to these functions
98 – 100	w-{fopen,fclose,fflush}	ditto weighted by static execution frequency
101 – 103	cnt-{bb,ass,expr}	number of basic blocks, assignments, expressions within assignments
104 – 106	w-cnt-{bb,ass,expr}	ditto weighted by static execution frequency
107 – 109	frac-{call,ctrl,if}	fraction of function calls, statements altering the control flow, `if`-expressions
110 – 112	w-frac-{call,ctrl,if}	ditto weighted by static execution frequency
113	cc	cyclomatic complexity by McCabe
114	nd	nesting depth by Gong and Schmidt

Table 1: Static code features for learning loop iteration counts

for scheduling instructions. We can thus apply our ML-based mapping without the need to annotate latencies for new architectures. That is, we do not have to modify the feature extraction algorithm because it gets the latency information as an input parameter. The fraction of control instructions defines how much of the code will be executed conditionally. Loop-related features, such as the number of loop iterations, the loop nesting and the amount of interprocessor communication are relevant because they contribute most to the execution time. For example, the deeper the loop nesting, the longer the execution time can be expected to be.

Learning loop iteration counts enables us to predict precisely the amount of interprocessor communication, and the learned execution time enables us to map processes power-efficiently to PEs. Both learned information can be used to improve the mapping of MPI processes, as we show in the following section. Since different kinds of programs cannot be expected to behave similarly, we propose using program classification [AG09] to improve the preciseness of the resulting classifiers.

3.2 Compilation

At compile time, we aim at mapping MPI programs to concurrent architectures. Hence, we have to analyze the synchronization between the processes. That is, we analyze the MPI synchronization calls. On encountering such calls, we split each participating process into two tasks, the part of the process until the synchronization and the part of the process after the synchronization. The rationale is that the communication behavior of a process might change during execution. This modelling facilitates to identify points in the program where a reallocation of a process to another PE can be beneficial. To find an optimal mapping, we must take into account the interprocessor communication at runtime and the execution times of processes. To that end, we simply have to extract the code features of each task via highly scalable static analysis. From this, we obtain behavior predictions for execution times of tasks and the communication amount between tasks with our learned statistical model (see Figure 2). Because we also obtain the communication latency between PEs from profiling during the training phase, we have the *communication cost*.

From this, we build a *task graph* with vertices representing tasks and edges representing dependencies as shown in Figure 3 on the left. To reflect synchronization, we insert synchronization points into the graph. The tasks are labeled with the first number being the rank of the corresponding MPI process (i.e., the unique ID which is known by the MPI

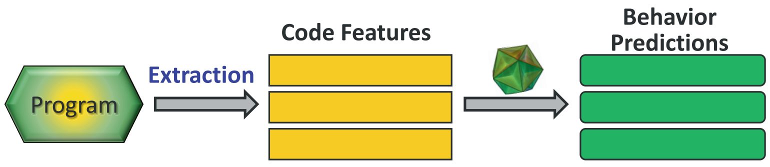

Figure 2: ML-based behavior prediction

runtime daemon) and the second number representing the part of the process. In Figure 3, tasks $T_{0.1}$ and $T_{1.1}$ as well as tasks $T_{1.3}$ and $T_{2.3}$ communicate with each other (indicated by a red arrow), and tasks that have dependencies (illustrated by gray arrows) execute in sequence while the other can execute in parallel. Next, we annotate the task graph with our predictions for the execution times and the communication amount. This information can then be used for communication-aware and power-efficient allocation of tasks to PEs.

The architecture we consider can be described by an undirected, weighted *host graph* $H = (V, E)$. The vertices in V correspond to PEs, edges in E represent the communication medium, and the weight of an edge models the communication cost between PEs. The goal of mapping parallel applications to concurrent architectures therefore can be seen as the problem to find an embedding of the corresponding annotated task graph $TG = (TV, TE)$ into H with respect to minimize the overall execution time and communication cost. To minimize overall the execution time we assign a priority ρ to each task T_i with predicted execution time t_i. We define this priority as

$$\rho(T_i) = t_i + \max_{T_j \in TV, (T_i, T_j) \in TE} \rho(T_j)$$

It can be interpreted as the length of the execution time of the critical path in TG from T_i until the end of the executions of all its descendants. Additionally, we partition the task graph into *regions* such that all tasks within a region can be executed in parallel. That is, there are no dependencies among the tasks. In Figure 3, we have three regions consisting of the tasks that are shown on the same horizontal level. Then we order the parallel executable tasks of each region with an priority-weighted list scheduling algorithm.

The final step is the allocation of tasks to PEs, which we perform per region. We do not constraint the characteristic of the PE. For instance, it can be a vector processor, a scalar processor with multiple cores, or an FPGA. Hence, we have a hierarchical communication structure. For example, cores of a processor can communicate via a fast cache, processors can communicate via shared memory or more expensive communication medium like a network connection. Träff [Trä02] shows that the embedding of the communication graph into the host graph H with a hierarchical communication structure can be solved by *weighted graph partitioning*, optimizing for *totalcut*. Since the problem is NP-complete, using a Kernighan-Lin like linear time heuristics is proposed [Trä06]. The heuristics can give better results than standard algorithms using recursive bipartitioning. We extend this

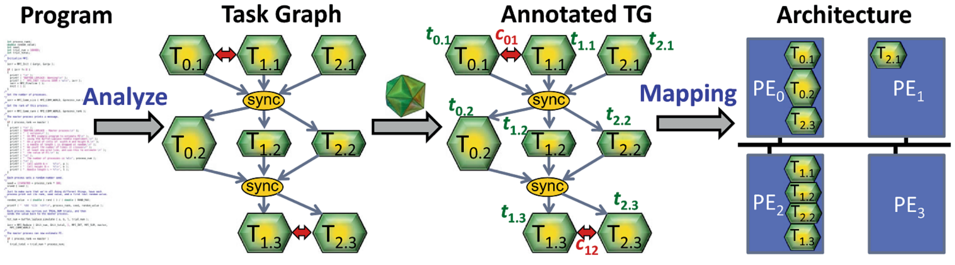

Figure 3: Mapping of MPI processes to concurrent architectures

heuristics by allowing tasks to be mapped to the same PE if the sum of its execution times do not exceed the execution time of other tasks in the same region. The resulting mapping of processes to PEs is automatically given to the MPI runtime daemon.

Consider the mapping of the annotated task graph to the architecture shown in Figure 3 on the right. The concurrent architecture is composed of PEs connected by a bus which results in different communication latencies. The latency between PE_0 and PE_2 is as low as the latency between PE_1 and PE_3, whereas all other latencies are much higher. At release time of the processes, the MPI runtime daemon without having predictions for the communication amount and the execution time typically performs workload balancing. This could result for example in allocating the process with rank 0 (i.e., tasks $T_{0.1}$ and $T_{0.2}$) to PE_0, the process with rank 1 (i.e., tasks $T_{1.1}$ to $T_{1.3}$) to PE_1, and the process with rank 2 (i.e., tasks $T_{2.1}$ to $T_{2.3}$) to PE_2.

Our ML-based predictions make possible to direct the MPI runtime daemon allocating $T_{1.1}$ to PE_2 (instead of to PE_1), which results in lower communication overhead. At each synchronization point, we analyze whether another allocation is beneficial and if so, we inject calls to signal the MPI runtime daemon that reallocation is necessary. Therefore, if the execution times of tasks $T_{1.2}$ and $T_{2.2}$ together are predicted to be no longer than the execution time of task $T_{0.2}$, the MPI runtime daemon will be directed to allocate both to the same PE. At this time, we can put PE_2 and PE_3 in a low power state. Assume that the last part of the processes with rank 1 and 2 have similar predicted execution times. At the second synchronization point the MPI runtime daemon has the information that both processes shall be allocated close to each other but not on the same PE because of their similar execution times and therefore reallocates task $T_{2.3}$ to PE_0. Thus, our approach is communication-aware and power-efficient.

4 Implementation

We have implemented the machine learning algorithm within the *R Project* [Dev08], together with the *rpart* package for the classification tree algorithm. *R Project* is a collection of statistical functions, which we have already successfully used in previous work [TG10]. For program classification [AG09], we use the *hclust* function for hierarchical clustering. We implement the compilation steps within the modular, industrial strength *Compiler Development System* (*CoSy*) of ACE [ACE11]. The steps during compilation like e.g., analyses, optimizations, and code generation are implemented in modules called *engines*. We have implemented the extraction of code features for the learning of the loop iteration counts and the static branch prediction algorithm proposed by Wu and Larus [WL94] for determining the execution probabilities and frequencies. To determine the actual loop iteration count for profiling during the training phase, we have used the *path profiling engine* of CoSy.

5 Related Work

The application of ML techniques in compiler frameworks has become a challenging research area. The key advantage of learning techniques is their ability to find relevant information in a high-dimensional space, thus helping us to understand and control a complex system. Heuristics automatically generated via learning algorithms often outperform hand-crafted models (cf. [MBQ02]). As an example application, this approach learns to decide if Loop Unrolling is beneficial or not. The approach of Fursin et al. [FMT$^+$08] targets the selection of good optimization passes via *machine learning*. During the training phase, information about the program structure is gathered using 55 static program features and the behavior of programs when compiled under different optimization settings is recorded. After training, on encountering a new program the optimizations to be applied are selected.

Close to our approach for predicting execution times using ML techniques is the approach of Powell and Franke [PF09] for predicting cycle counts to speed up instruction set simulation. However, their method allows the training data to be updated, online feature selection, and switching between cycle- and instruction-accurate simulation to ensure the accuracy of the predictions being made. To the best of our knowledge, no approach targets task allocation in compilers using ML techniques.

The mapping and scheduling problems on multiprocessor systems are often modeled as *integer linear programming* (*ILP*) problems ([RGB$^+$06, YH08]). The major drawback of ILP is its limitation to small problems due to its computational complexity. Heuristic approaches such as *genetic* or *evolutionary algorithms* are therefore being increasingly developed (e.g., [GTT09, PLL07, Yoo09, YH09]). Their main disadvantage is the extended compile time due to the iterative approach of finding good solutions. Common to all these approaches is the requirement of having information about communication overhead or execution time of the tasks. Thus, our technique supports these approaches by automatically generating predictions for the claimed information.

Several new approaches target communication- and power-aware task scheduling in compilers for concurrent architectures (though without using ML techniques). For example, the approach of Varatkar and Marculescu [VM03] takes an abstract application task graph and the architecture graph as inputs, assigns the tasks to processors and then schedules them. Nodes in the abstract task graph are annotated with deadlines and the number of cycles taken by that task for computation. Arrows, which indicate a control dependence between two tasks, are annotated with a number representing the quantity of communication traffic between the tasks. Since the amount of communication between the tasks and the processor cycles needed for each task has to be annotated, our technique can be adapted to this approach, thus eliminating the need for these annotations. However, voltage selection is modeled as an ILP problem, which suffers from computational complexity.

A task and data migration scheme that decides whether to migrate tasks or data to satisfy a given communication requirement is proposed by Ozturk et al. [OKSK06]. The choice is made at runtime based on statistics collected off-line through profiling. The major drawback of one profiling run is the strong dependence on the input data set of that run. That is, selecting input sets that generate representative profiles is a difficult task. However, the authors' experience with several applications indicates that in some cases migrating the task itself – instead of data – can be more beneficial from both performance and power perspectives. Based on this observation, it is clear that task allocation and minimizing energy consumption should be jointly addressed during compilation.

To overcome the limitations of ILP due to computational complexity and the drastic over-approximation of static analyses due to unknown runtime behavior, we use supervised classification learning. This automatically generates classifiers, which relate static code features to the dynamic behavior, thus providing the compiler with intelligence. At compile time, these classifiers can be used as precise and fast heuristics for communication-aware task mapping and power minimization, which eliminates the need for manual annotations of forecast values.

6 Conclusions

Using ML techniques in compilers is a challenging research area. In this paper, we have presented a novel, highly scalable ML-based approach for predicting runtime communication overhead and execution time targeting the mapping of MPI processes to concurrent architectures. Our proposed learning algorithm automatically relates static code features to dynamic runtime behavior. This allows us to generate much more precise heuristics than those generated by static analyses, as our experimental results in [TG10] demonstrate.

Since the learning algorithm must be deployed only once per architecture decoupled from the compilation, the compile time itself is not increased, in contrast to other heuristic approaches such as *genetic* and *evolutionary algorithms* or profile-guided compilation. The latter approach is as well strongly dependent on the input data set, hence being too precise. Instead, we perform several profiling runs with different input data during the training phase. Thus, we focus on realistic behavior and not on one single behavior. Furthermore, our technique is applicable to all other approaches where user annotations or profiling are necessary, thus eliminating their requirements. Additionally, our solution is fully retargetable and is not limited to static mapping at compile time. Instead, the established heuristics can be used to make the MPI runtime scheduler more intelligent concerning the runtime behavior of the processes. With our predictions for the execution times and the communication amount, we can direct the MPI runtime scheduler to reallocate processes communication-aware and power-efficient instead of simply performing workload balancing. This is a great improvement to current MPI implementations because even the best MPI implementation cannot ensure that for instance frequently communicating processes are placed close to each other.

In the future, we will apply our approach for learning execution times. Because communication between processes can also arise within recursive functions, we plan to learn the recursion depth of these functions, thus giving us an estimate for the communication overhead in these cases. Finally, we will use all results to guide the mapping of parallel applications to concurrent architectures.

References

[ACE11] ACE. *Associated Compiler Experts bv., Amsterdam, The Netherlands*, 2011.

[AG09] Lars Alvincz and Sabine Glesner. Breaking the Curse of Static Analyses: Making Compiler Intelligent via Machine Learning. In *3rd Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion (SMART'09)*, January 2009.

[Alp10] Ethem Alpaydin. *Introduction to Machine Learning, Second Edition*. Adaptive Computation and Machine Learning. The MIT Press, 2nd edition, 2010.

[Dev08] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008.

[FMT$^+$08] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, François Bodin, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, and Michael O'Boyle. MILEPOST GCC: Machine Learning Based Research Compiler. In *Proceedings of the GCC Developers' Summit*, pages 7–19, June 2008.

[GLM04] Robert B. Gramacy, Herbert K. H. Lee, and William G. Macready. Parameter Space Exploration with Gaussian Process Trees. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, pages 45–53, New York, NY, USA, 2004. ACM.

[GS85] Huisheng Gong and Monika Schmidt. A complexity measure based on selection and nesting. *SIGMETRICS Perform. Eval. Rev.*, 13:14–19, June 1985.

[GTT09] C. Goh, E. Teoh, and K. Tan. A hybrid evolutionary approach for heterogeneous multiprocessor scheduling. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 13:833–846, 2009.

[Ind04] Piotr Indyk. Nearest Neighbors In High-Dimensional Spaces. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 39, pages 877–892. CRC Press LLC, Boca Raton, FL, 2nd edition, April 2004.

[MBQ02] Antoine Monsifrot, François Bodin, and René Quiniou. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Proceedings of AIMSA'02*, pages 41–50, London, UK, 2002. Springer-Verlag.

[McC76] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[Mes09] Message Passing Interface Forum. *MPI: a Message-passing Interface Standard: Version 2.2*. University of Tennessee, 2009.

[OKSK06] O. Ozturk, M. Kandemir, S. W. Son, and M. Karakoy. Selective code/data migration for reducing communication energy in embedded MpSoC architectures. In *Proceedings of GLSVLSI'06*, pages 386–391, New York, NY, USA, 2006. ACM.

[PF09] Daniel Christopher Powell and Björn Franke. Using Continuous Statistical Machine Learning to Enable High-Speed Performance Prediction in Hybrid Instruction-/Cycle-Accurate Instruction Set Simulators. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, pages 315–324, New York, NY, USA, 2009. ACM.

[PLL07] Saeed Parsa, Shahriar Lotfi, and Naser Lotfi. An Evolutionary Approach to Task Graph Scheduling. In *International Conference on Adaptive and Natural Computing Algorithms (ICANNGA 1)*, pages 110–119, 2007.

[RGB$^+$06] Martino Ruggiero, Alessio Guerri, Davide Bertozzi, Francesco Poletti, and Michela Milano. Communication-Aware Allocation and Scheduling Framework for Stream-Oriented Multi-Processor Systems-on-Chip. In *Proceedings of DATE'06*, pages 3–8, 3001 Leuven, Belgium, 2006. European Design and Automation Association.

[Smi00] Michael D. Smith. Overcoming the challenges to feedback-directed optimization (Keynote Talk). In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, DYNAMO '00, pages 1–11, New York, NY, USA, 2000. ACM.

[Ste09] Dan Steinberg. CART – Classification and Regression Trees. In Xindong Wu and Vipin Kumar, editors, *The Top Ten Algorithms in Data Mining*, Chapman & Hall/CRC data mining and knowledge discovery series, chapter 10, pages 179–201. CRC Press, 2009.

[TG10] Dirk Tetzlaff and Sabine Glesner. Intelligent Task Mapping using Machine Learning. In *CiSE '10: Proceedings of the 2010 International Conference on Computational Intelligence and Software Engineering*, pages 1–4. IEEE Computer Society, dec. 2010.

[Trä02] Jesper Larsson Träff. Implementing the MPI Process Topology Mechanism. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Supercomputing '02, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[Trä06] Jesper Larsson Träff. Direct graph *k*-partitioning with a Kernighan-Lin like heuristic. *Operations Research Letters*, 34(6):621–629, 2006.

[Tur50] Alan M. Turing. COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, LIX(236):433–460, 1950.

[VM03] Girish Varatkar and Radu Marculescu. Communication-Aware Task Scheduling and Voltage Selection for Total Systems Energy Minimization. In *Proceedings of ICCAD'03*, pages 510–517, Washington, DC, USA, 2003. IEEE Computer Society.

[Wan03] John Wang, editor. *Data Mining: Opportunities and Challenges*. IGI Publishing, Hershey, PA, USA, 2003.

[WL94] Youfeng Wu and James R. Larus. Static Branch Frequency and Program Profile Analysis. In *Proceedings of MICRO 27*, pages 1–11, New York, NY, USA, 1994. ACM Press.

[YH08] Hoeseok Yang and Soonhoi Ha. ILP Based Data Parallel Multi-Task Mapping/Scheduling Technique for MPSoC. In *Proceedings of ISOCC'08*, volume 01, pages I–134–I–137, Nov. 2008.

[YH09] H. Yang and S. Ha. Pipelined Data Parallel Task Mapping/Scheduling Technique for MPSoC. In *Proceedings of DATE'09*, pages 69–74, 2009.

[Yoo09] Myungryun Yoo. Real-time task scheduling by multiobjective genetic algorithm. *Journ. of System a. Software*, 82(4):619–628, 2009.