Partitioning Streaming Parallelism for Multi-cores: A Machine Learning Based Approach

Zheng Wang
Institute for Computing Systems Architecture
School of Informatics
The University of Edinburgh, UK
jason.wangz@ed.ac.uk

Michael F.P. O'Boyle
Institute for Computing Systems Architecture
School of Informatics
The University of Edinburgh, UK
mob@inf.ed.ac.uk

ABSTRACT

Stream based languages are a popular approach to expressing parallelism in modern applications. The efficient mapping of streaming parallelism to multi-core processors is, however, highly dependent on the program and underlying architecture. We address this by developing a portable and automatic compiler-based approach to partitioning streaming programs using machine learning. Our technique predicts the ideal partition structure for a given streaming application using prior knowledge learned off-line. Using the predictor we rapidly search the program space (without executing any code) to generate and select a good partition. We applied this technique to standard StreamIt applications and compared against existing approaches. On a 4-core platform, our approach achieves 60% of the best performance found by iteratively compiling and executing over 3000 different partitions per program. We obtain, on average, a 1.90x speedup over the already tuned partitioning scheme of the StreamIt compiler. When compared against a state-of-the-art analytical, model-based approach, we achieve, on average, a 1.77x performance improvement. By porting our approach to a 8-core platform, we are able to obtain 1.8x improvement over the StreamIt default scheme, demonstrating the portability of our approach.

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors—*Compilers, Optimization*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Experimentation, Languages, Measurement, Performance

Keywords

Compiler Optimization, Machine Learning, Partitioning Streaming Parallelism

1. INTRODUCTION

Multi-core processors are now mainstream, offering the promise of energy efficient hardware performance [12]. To make use of such potential, however, new and existing applications must be written or transformed so that they can be executed in parallel. While there has been considerable research effort focused on (semi-) automatically transforming existing sequential programs into a parallel form [38], it remains uncertain whether this is a long term viable approach to achieving scalable parallelism [2].

An alternative approach has been to develop new high level programming languages and models where the parallelism is explicit. Although the application developer has to think explicitly about parallel structure in the application, he is freed from implementation concerns. As an application may be ported to new platforms many times in its life-time, this is a significant advantage. Examples of such languages and models include UPC [8], X10 [29], Galois [17] and HTA [3].

One popular language domain focuses on streaming applications exposing task, data and pipeline parallelism [33]. Parallelism in streaming languages is explicit and it is now the system's responsibility to effectively map this parallelism to the underlying hardware. Mapping is typically broken into two stages: partitioning the program graph into regions which are allocated to threads and then scheduling, which allocates the threads to the underlying hardware. This is by no means a new challenge and there is an extensive body of work on mapping task, data and pipeline parallelism to parallel architectures using runtime scheduling [28], static partitioning [18, 30, 35], analytical models [7, 25], heuristic-based mappings [10], or ILP solvers [16, 39]. They can each achieve good performance but are restricted in their applicability. Fundamentally, such approaches are based on the developers' view about the most significant costs of the target platform and typical programs, encoding a hardwired human-derived heuristic. However, as we show later, the best form of the program varies across programs and platforms.

The problem with hardwired heuristics has been addressed by several researchers [23, 34] who advocate the use of machine learning (`ML`) as a methodology to automatically construct optimisation heuristics [34, 41]. Such an approach has the advantage of being portable across different platforms without requiring expert knowledge. However, until now, it has been limited to relatively straightforward problems where the target optimization to predict is fixed, e.g. determining compiler flag settings [15], loop unroll factors [22] or the number of threads per parallel loop [41]. Determining the best partitioning of a stream program is fundamentally a more difficult task. First of all, rather than predicting a fixed set of optimisations we are faced with predicting an unbounded set of coalesce and split operations on a program graph. As the graph changes structure after each operation, this further increases the complexity. A secondary issue limiting the applicability of ML is its reliance on sufficient training data. This problem is particularly acute for emerging parallel programming languages where the application code base is small as is the case for streaming languages.

This paper tackles both these problems. It develops a technique to automatically derive a good partitioning for StreamIt [36] programs on multi-cores making no assumption on the underlying architecture. Rather than predicting the best partitioned graph, it develops a *nearest-neighbour* based machine learning model that predicts the ideal partitioned *structure* of the StreamIt program. It then searches through a program transformation space (without executing any code) to find a program of the suitable structure. To overcome the problem of insufficient training programs we have developed a micro-kernel stream program generator. This generator is able to provide many small training examples for the predictive model.

To show the automatic portability of our approach, we have evaluated our ML based approach on two different multi-core platforms. On a 4-core machine, on average, our approach achieves 1.90 times speedup over the dynamic programming based StreamIt partitioner, which translates to 60% of the performance gained by exhaustively evaluating over 3000 different partitions per program and selecting the best. Compared to a state-of-the-art analytical model-based approach, we achieve a 1.77x performance improvement. When ported to an 8-core machine, we achieved 1.80x performance improvement over the the dynamic-programming based StreamIt partitioner.

The remainder of this paper is structured as follows. We motivate our work based on simple examples in section 2. This is followed by a description of our machine learning based approach in section 3 and 4. Our experimental methodology and results are discussed in sections 5 and 6, respectively. We establish a wider context of related work in section 7 before we summarize and conclude in section 8.

2. BACKGROUND AND MOTIVATION

2.1 StreamIt Language

StreamIt [11] is a language supporting streaming programming based on the *Synchronous Data Flow* (SDF) model [19]. In StreamIt, computation is performed by *filters* which are the basic computational units. Filters communicate through dataflow channels, which are implemented as FIFO queues. StreamIt provides a simple means of constructing rich hierarchically parallel structures such as *pipeline* and *split-join* (i.e., data parallelism).

Each StreamIt program is represented by a stream graph. Figure 1 (a) illustrates a simplified stream graph for the MP3DECODER benchmark. Each node is a task that can be executed in pipeline fashion. Concurrent task parallelism is achieved after each splitter node. Communication between tasks is defined by and restricted to the arcs between nodes. It is the compiler's responsibility to partition this graph and allocate partitions to threads which are then

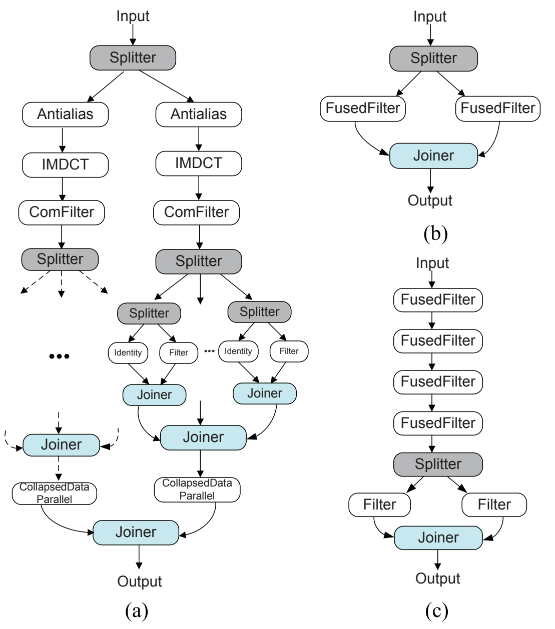


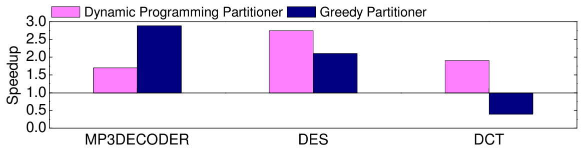
Figure 1: A simplified stream graph of the MP3DECODER StreamIt program (a). Each node is a task that can be executed in pipeline fashion. Concurrent task parallelism is achieved after each splitter node. A greedy partitioner applied to this program gives the graph (b) with just 4 nodes. A dynamic programming based partitioner gives the graph (c) with 8 nodes.

scheduled on the underlying hardware. This is a 2 stage process and is illustrated in figure 3. First, the nodes in the original graph are merged into larger nodes or spilt into smaller nodes by a sequence of fuse and fission operations. This gives a transformed program where each node is allocated to a separate thread. Each thread is then scheduled to the hardware by the runtime. The first stage we call *partitioning* as it is concerned with determining those regions of the stream program that will be eventually allocated to a thread. The second stage we call *scheduling* and is responsible for the allocation of threads to processors. In this paper, we are interested in the mapping of nodes to threads and hence focus purely on the first stage of the process[1], partitioning.

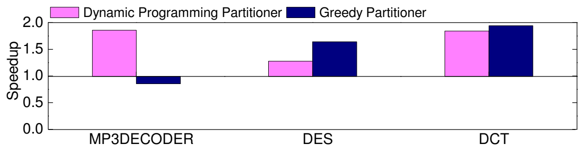
2.2 Motivation

Finding a good partitioning for a streaming program is difficult due to the large number of possible partitions. Figure 1 (b) and (c) show two possible partitioned versions of the original MP3DECODER program shown in figure 1 (a). Both are obtained by applying a sequence of fuse and fission operations on the original graph. The first partitioning shown in figure 1 (b) corresponds to a greedy partitioner, the second partitioning, figure 1 (c) corresponds to a dynamic programming based method; both are StreamIt compiler built in heuristics [36]. The question is which is the best one? This problem of graph partitioning in its general

[1]Note: as we use a machine learning approach, we implicitly consider the behaviour of the scheduler along with the rest of the underlying system (hardware, operating systems etc.) when generating training data.



(a) 4-core platform



(b) 8-core platform

Figure 2: The relative performance of 2 partitioning schemes with respect to a naïve partitioning scheme. The results are shown for 2 platforms and 3 distinct StreamIt programs. Greedy partitioning performs well on MP3DECODER on the 4-core platform but not as well as dynamic programming partitioning on the 8-core. This relative ordering is reversed for DES and DCT. Determining the best partitioning depends on program and platform.

form is known to be NP-complete and it is difficult to devise a general heuristics [5].

To illustrate this point, consider figure 2, which shows the performance of each partitioning approach on two different multi-core platforms: a 2x dual-core (4-core) machine and a 2x quad-core (8-core) machine. On the 4-core the greedy scheme performs well for MP3DECODER; it has a lower communication cost, exploiting data parallelism rather than the pipeline parallelism favoured by the dynamic programming partitioner. On the 8-core platform, however, the dynamic programming-based heuristic delivers better performance as load balancing becomes critical.

When examining two further programs, DES and DCT, we see the best partitioning algorithm for a particular machine is reversed. The figure shows that there is no current "one-fits-all" heuristic and the best heuristic varies across programs and architectures. Rather than relying on heuristics, we would like a scheme that automatically predicts the right sequence of fuse and fission operations for each program and architecture. In the case of MP3DECODER, this means we want to select the operations that give the partitioned code in figure 1 (b) for 4-cores and the partitioned code in figure 1 (c) for 8-cores. However, predicting the correct sequences of fuse and fission is highly non-trivial given the unbounded structure of the input program graphs.

In the next section we describe our novel approach. Rather than predicting the sequence of fuse and fission operations directly, it tries to predict the right *structure* of the final partitioned program. Given this target structure, it then searches for a sequence of fuse and fission operations that generates a partitioned program that fits the predicted structure as closely as possible.

3. PREDICTING AND GENERATING A GOOD PARTITION

One of the hurdles in predicting the best sequence of fusion and fission operations is that the graph keeps changing structure after each operation. In figure 3, the second operation fiss(2.3) would have to be renamed fiss(1.6) if the first operation (fuse(1.2,1.3,1.4,1.5)) had not taken place. Any scheme that tries to predict a sequence of fuse and fission operations has therefore to take into consideration the structure of the graph at each intermediate stage. The supervised predictive modelling schemes explored to date are incapable of managing this [9]. We take a different approach. Instead of trying to predict the sequence of fuse and fission operations, we divide the problem into two stages as illustrated in figure 4:

1. Predict the ideal structure of the final partitioned program.
2. Search a space of operation sequences that delivers a program as close as possible to the ideal structure.

The first stage focuses on determining the goal of partitioning, i.e., the structure of the partitioned program without regard to how it may be actually realised. The second stage explores different legal operation sequences until the generated partitioned program matches the goal. This frees us from the concern of correctly predicting the syntactically correct sequence of fuse and fission operations. Instead we can try arbitrary sequences until we reach a partition that closely matches our goal. The next section describes how we can predict a good partitioning goal and is followed by a section describing how we can generate a sequence of fuse and fission operations to reach that goal.

3.1 Predicting the Ideal Partitioning Structure - Setting the Goal

We wish to predict the ideal partitioned structure of any input graph program. In order to cast this as a machine learning problem, we wish to build a function f which, given the essential characteristics or features X_{orig} of the original program predicts the features of the ideal partitioned program X_{ideal}. Building and using such a model follows the well-known 3 step process for supervised machine learning [4]: (i) generate training data (ii) train a predictive model (iii) use the predictor. We generate training data by evaluating (executing) randomly generated partitions for each training program and recording their execution time. The features of the original and best partitioned program are then used to train a model which is then used to predict the best ideal partitioning structure for any *new unseen* program. One of the key aspects in building a successful predictor is developing the right program features in order to characterise the original and goal program. This is described in the next section. This is followed by sections describing training data generation, building the predictor using nearest neighbors and then using the predictor.

3.2 Extracting Features

Rather than trying to deal with unbounded program graphs as input and outputs to our predictor, we describe the essential characteristics of such graphs by a fixed feature vector of numerical values. The intention is that programs with similarly feature vectors have similar behaviour. We empirically evaluate this assumption in section 6.3. In this work, we use program *features*, to characterise a streaming application. The set of program features are summarized in table 1. We extract two sets of those features from the overall stream graph and the critical path of the program. Thereby, one set

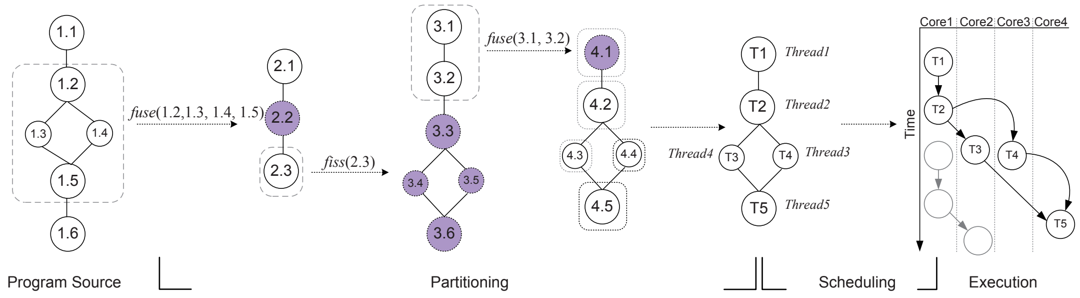


Figure 3: The mapping process can be broken into 2 main stages partitioning and scheduling. Partitioning is responsible for mapping nodes to threads. This is achieved by a series of fuse and fission operations on the original source program. At the end of this process, each node of the final graph is allocated to a thread and so the original graph has been partitioned. Scheduling allocates each of these threads to cores. Scheduling may be dynamic especially if the number of threads is greater than cores. We focus solely on partitioning.

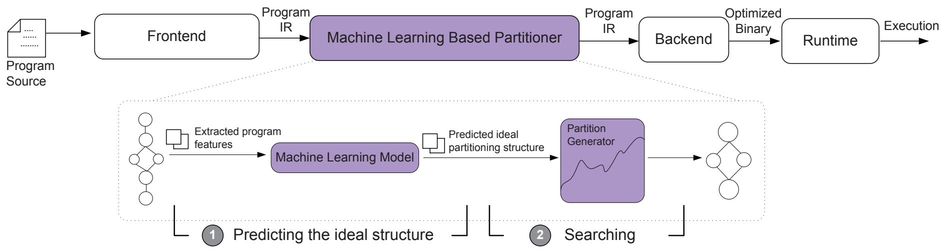


Figure 4: Work flow of our compiler framework. The compiler takes in program source code and produce an optimised binary. In the middle of the compiler is a machine learning based partitioner. The partitioner firstly predicts the features of an ideal partitioning structure of the input program. This is done by checking the similarity of the input program features to prior knowledge. Then, it searches the transformation space to generate a program whose features are as close as possible to the predicted ideal structure.

Program Features	
#Filter	#Joiner
Pipeline depth	Splitjoin width
Avg. unit work	Max unit work
Pipeline work	Splitjoin work
Computation	Computation of stateful filters
Branches per instruction	Load/store per instruction
Avg. dynamic rate	Max dynamic rate
Avg. commun. rate	Computation-commun. ratio
Avg. commun. / unit work	Avg. bytes commun. / unit work
Max commun. rate / unit work	Work Balance

Table 1: Program features extracted from a streaming application.

represents the overall characteristics of a streaming program and the other captures characteristics *solely* of the program's critical path. Features are extracted from the stream graph (i.e., program `IR`) without running the program, thus the overhead of extracting features is insignificant.

For a given streaming program, our model firstly extracts a feature vector, $X_{orig} = [x^1_{orig}, x^2_{orig}, \cdots, x^n_{orig}]$ from the original stream graph. X_{orig} is used to characterise it. We use the same feature set X_{part} to characterise any partitioning of a given program. We normalise the features and use principal component analysis (`PCA`) [4] to reduce redundancies between features.

3.3 Generating Training Data

Once we have a means of describing the original and partitioned programs, we can start generating training data. Training data are generated by evaluating on average 3000 *different* randomly generated partitions for a program and recording the execution time, t. For each program, we also extract program feature sets, X_{orig} and X_{part}, of both the program and its partition respectively. Program features and execution time are put together to form an associate training dataset $T = \{(X^i_{orig}, (X^{i,j}_{part}\ t^{i,j})\}i \in 1, \cdots, N$ and $j \in 1, \cdots, R$, for N training programs in which each program has R different partitions.

Synthetic Stream Program Generation. One particular problem encountered in training for new languages is that the training set is small. There simply is not a large enough program base with to work. To overcome this problem, we build a micro-kernel stream program generator to generate many small training examples, supplementary to existing benchmarks. This allows us to train our model on larger data sets. Our stream program generator automatically extracts micro-kernels (i.e., working functions and communicating patterns) from any subset of existing StreamIt programs. It limits the generated programs to a space with parallel parameters (i.e., pop and push rate, pipeline depth, split-join width, and loop iteration counts). Then, it generates a large number of small training examples in which the parallel parameters and working functions are varied.

The cost of generating new benchmarks can be neglected because millions of programs can be generated in an order of minutes. Running programs to generate training data from potentially thousands programs, however, is prohibitively expensive. Therefore, we select a limited number of representative programs by using a clustering technique that is also used to choose the ideal partitioning structure (as de-

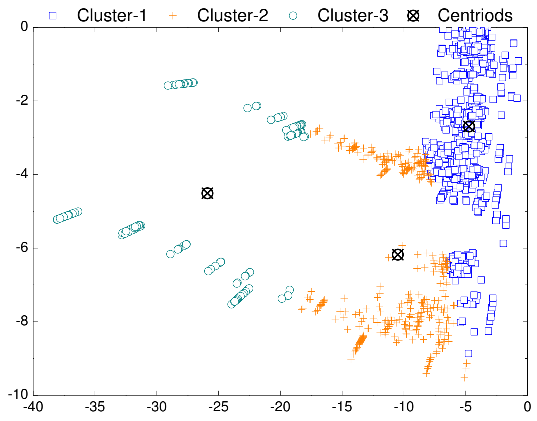


Figure 5: The feature space is projected into 2 dimensions in this diagram for presentation purposes. Each of the points represents a good partitioning structure for the LATTICE program. Using clustering there are 3 clusters found the centre of each is marked as the centroid. Cluster-1 is selected as it has the best mean speedup. The features around the centroid of cluster-1 are then averaged and used as the ideal partitioning structure.

scribed in section 3.4.1). Informally, we do this by examining the features of the programs generated and selecting only those which are sufficiently distinct form the existing training set. Although producing training data takes time, it is only a one off cost incurred by our model. Furthermore, generating and collecting data is a completely automatic process and is performed off-line. Therefore, it requires far less effort than constructing a heuristic by hand.

3.4 Building a Model

Once we have generated sufficient training data, we are in a position to build a predictive model. Our model is based on a straightforward *nearest-neighbour classifier* [9]. For each training program, we record the features of the original program X_{orig} and those of its best found partition X_{ideal}. When used on a new unseen program, we find the program from the training set whose features most closely match the new program's features. We then simply return the features, X_{ideal} of the training program as the predicted best ideal partition for the new program.

In our setting, each program in fact has a number of partitions that give good performance. To capture this we group the best partitions into regions using *clustering*. We consider the cluster with the best average performance and select a representative candidate as described in the next section.

3.4.1 Selecting the Ideal Partitioning Structure

Each training program has a number of good partitions. Our task is to select the most useful one, i.e., the partition that is likely to be good for similar programs. We cluster the good partitions using a *k-means* clustering algorithm [4]. In order to determining the right number of clusters K, we use the standard *Bayesian Information Criterion* (`BIC`) score [26, 31] to decide how many clusters should be selected. `BIC` is a measurement of the "goodness of fit" of a clustering parameter (i.e., K) to a given data set. The larger the `BIC` score, the higher chance that we find a good clustering number for the data set. We use the `BIC` formulation given in [26], which is:

$$BIC_j = \hat{l}_j - \frac{p_j}{2} \cdot logR \qquad (1)$$

where $\hat{l}_j$ is the log-likelihood of the data when K equals to j, R is the number of points in the data (i.e., the number of generated partitions), and p_j is the number of free parameters to estimate, which is calculated as: $p_j = (K-1)+dK+1$ for a d-dimension feature vector plus 1 variance estimate [32]. $\hat{l}_j$ is computed as:

$$\hat{l}_j = \sum_{n=1}^{K} -\frac{R_n}{2}log(2\pi) - \frac{R_n \cdot d}{2}log(\hat{\sigma}^2) - \frac{R_n - K}{2} + R_n log(R_n/R) + logS_n \qquad (2)$$

where R_n is the number of points in the nth cluster, $\hat{\sigma}^2$ is the average variance of the distance from each point to its cluster center, and S_n is the normalised average speedup of the nth cluster.

We apply the *k-means* clustering algorithm for the generated programs by varying the cluster number K. For each clustering result, we firstly calculate its corresponding `BIC` score. We choose a clustering number, K_{best}, which gives us the highest `BIC` score. After this point, we know the partition space can be represented by K_{best} clusters.

Once we have found the number of clusters, we select the cluster that has the largest number of good partitions. We select the 10% of partitions that are close to the cluster centriod of the selected cluster and normalise their feature values. The normalised features are considered as the ideal partition structure.

Figure 5 visually depicts the use of this clustering technique. Each point in the figure represents a good partition of the StreamIt benchmark LATTICE. To aid clarity, we have projected the dimension of the feature space down to 2. For these program there are 3 distinct clusters of good partitions. Cluster-1 is chosen because it contains 66% of all the partitions and has the highest mean speedup.

3.5 Using the Model to Predict the Ideal Partitioning Structure

Once we have gathered training data and built the model as described above, we can now use the model to predict the ideal partition structure of a *new, unseen* program as shown in figure 4. We firstly extract features of the input program, normalise its program features using `PCA`, and use the *nearest neighbour* model to predict the ideal partitioning structure of the input program. The nearest neighbor scheme picks a program in the training set, that is the most similar to the input program. This is done by comparing the input program's features to known programs' features. Once the nearest neighbor has been selected, we use its ideal partitioning X_{ideal} as the predicted ideal structure for the new program. In rare instances, our model may not be able to find any training programs that are similar enough to the input program (i.e., no programs in the training set are close to the new input program). It then simply uses the default partitioner provided by the compiler.

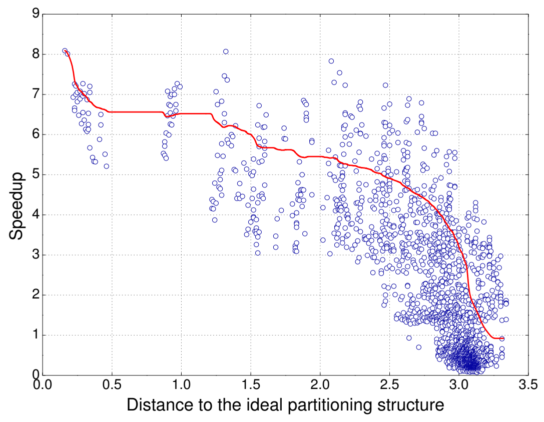


Figure 6: Separating partition candidates with Euclidean distances. The x-axis represents the distance to the ideal partitioning structure and the y-axis represents the speedup relative to the StreamIt default graph partitioner. Each dot represents a partition choice and the line represents the mean speedup of that partition within a distance.

4. SEARCHING AND GENERATING A PARTITION CLOSE TO THE PREDICTED IDEAL STRUCTURE

The previous section provides a means to predict the ideal partition structure without actually running the code. We now generate new partitions by applying random fuse and fission operations to the input program's graph. For each generated partition, we measure its Euclidean distance from the predicted ideal structure in the feature space. We repeat this many times, selecting the partition nearest the ideal structure.

Figure 6 illustrates the use of distance as a means of determining the best partition candidate for the StreamIt program LATTICE. Each dot represents a unique partition. There are over 3000 *different* partitions of which only 15% partitions are better than the partition generated by the StreamIt default scheme. Given the large number, selecting a partition to improve is non-trivial. The figure shows that distance to the ideal structure is a good measure of the quality of a partition. If we choose a distance of less than 0.5 as the confidence level, then we will pick a partition that is at least 5.21 times (6.6 times on average) faster than a partition generated by the StreamIt default partitioning heuristic.

For the purposes of this paper, we generate on average 3000 potential partitions for each new program, selecting the one that is nearest to the ideal. We do not execute any of these programs, merely extract their features and measure the Euclidean distance. Each partition takes less than 100 ms to generate and evaluate, so is not a significant cost.

5. EXPERIMENTAL METHODOLOGY

This section describes the platforms, compilers, and benchmarks used in our experiments as well as the evaluation methodology.

Benchmarks. We use the StreamIt benchmark suite version 2.1.1 to evaluate our approach. These applications represent typical streaming parallel programs containing both pipeline and data parallelism. On average, each program contains 46 (up to 168) filters at the `IR` level.

Compilers. We implemented our machine learning model as a stream graph partitioner in the StreamIt compiler (version 2.1.1). The StreamIt compiler is a source to source compiler which translates the partitioned stream graph into C++ code. The Intel C/C++ Compiler (`ICC`) version 11.0 was used to convert the C++ program to binary. We use "`-O3 -xT -aXT -ipo`" as the `ICC` compiler flags.

Hardware Platform. The experiments were performed on two multi-core platforms: a 4-core platform (with two dual-core Intel Xeon 5160 processors running at 3.0GHz and has 8GB memory) and an 8-core platform (with two quad-core Intel Xeon 5450 processors running at 3.0GHz and has 16GB memory). The dual-core Xeon 5160 processor has a 4MB L2-cache while the quad-core Xeon 5450 has a 12MB L2-cache. Both platforms run with 64-bit Scientific Linux with kernel 2.6.17-164 x86_64 SMP.

Cross-Validation. We use *leave-one-out-cross-validation* to evaluate our approach [4]. This means we remove the program to be partitioned from the training program set and then build a model based on the *remaining* programs. This also guarantees that our benchmark generator has not seen the target program before. The trained model is used to generate partitions for the removed target program. We repeat this procedure for each program in turn. It is a standard evaluation methodology, providing an estimate of the generalization ability of a machine-learning based model in predicting for an *unseen* programs.

Synthetic Benchmarks. We generate roughly over 100K synthetic programs and select around 60 for training. The benchmark generation and selection process takes less than 15 minutes.

5.1 Comparison

StreamIt has its default partitioning strategy, a sophisticated dynamic programming based partitioning heuristic [37]. All results are presented relative to this default and its provides a challenging baseline. To provide a wider comparison, we also evaluate a recently proposed analytical-based pipeline parallelism model [25] and an alternative greedy-based heuristic available within the StreamIt compiler [10]. The analytical-based model finds a suitable parallel mapping by predicting the execution time of a given streaming application. We have implemented the analytical-based partitioner in the StreamIt compiler. For each program, the partitioner generates 50,000 partitions of a single program and selects a mapping which has the best predictive performance as output. Our scheme, in contrast, predicts the best *structure* and selects the partition closest to it, using an order of magnitude fewer candidates.

5.2 Best Performance Found

In addition to comparison with existing approaches, we wish to evaluate our model by assessing how close its performance is to the maximum achievable. However, it is not possible to determine the best, due to the combinatorially large number of partitions. Instead we randomly generated 3,000 *different* partitions for each program and select the best performing partition as an indication of the upper bound on performance that could be achieved if we had sufficient resources. We call this "Best-Found" performance.

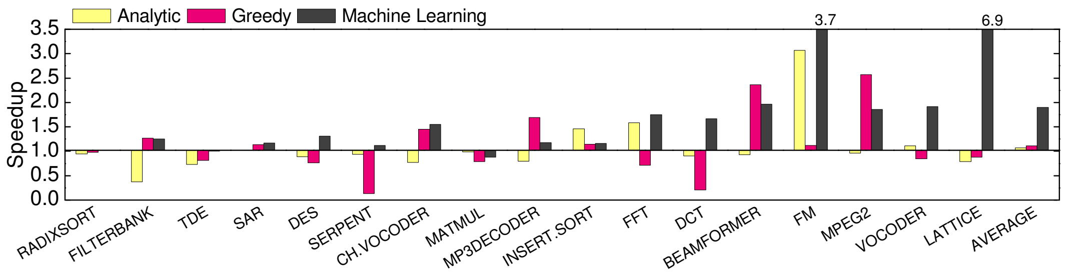


Figure 7: Performance comparison on the 4-core platform for the analytical model, the greedy partitioner and the ML-based model.

6. EXPERIMENTAL RESULTS

In this section we first report the performance of our approach against alternative approaches on the 4-core platform. This is followed by a short explanation of the results generated by different models. Next, we evaluate the accuracy of our model in predicting good structures and analyse what type of partitioning is important for each program. Finally, we extend our model to a 8-core platform and evaluate its performance.

6.1 Performance Comparisons

6.1.1 Comparison with other Techniques

Figure 7 shows the performance results for the 4-core platform. On average the analytical-based and the greedy-based partitioners do not significantly improve over the StreamIt default dynamic-programming-based partitioner. Our approach, however, is able to deliver significant improvement over the defaults scheme with a 1.90x average speedup.

Analytic. On average, the analytical-based model only achieves 1.07x speedup over the StreamIt default partitioner. This is not a surprising result because the StreamIt default partitioner is a strong baseline tuned by hand. It is only able to improve performance in 4 out of the 17 programs. It successfully partitions FM resulting in a 3.0x speedup by coarsening the pipeline. However, it fails to balance the pipeline of FILTERBANK due to its inability to capture the complicated communication pattern of the program. This leads to a greater than 2x *slowdown*.

Greedy. The greedy partitioner also fails to significantly improve over the StreamIt default partitioner. On average, it achieves a 10% performance improvement over the default partitioner. In approximately half of the programs, it gives a performance improvement. For example it is able to achieve an impressive 1.69x and 2.57x speedup on MP3DECODER and MPEG2 by reducing communication through aggressive fusion. However, it also slows down 9 applications, particularly in the case of SERPENT and DCT, which are up to 7.7x slower than the default dynamic programming approach. This result clearly shows the best partitioning heuristic varies from program to program.

Our approach. Our machine learning based approach, on the other hand, can greatly improve performance compared to the default partitioner and gives more stable results. It achieves better performance in most of the benchmark, up to 6.9x for LATTICE. In just one case, MATMUL, we perform worse than the default (as do the other 2 schemes). The backend `ICC` compiler aggressively auto-vectorizes the program which has not been captured by our model. This issue can be solved by adding additional features to the model and is the subject of future work.

6.1.2 Comparison vs Best-Found Performance

Although our scheme performs well compared to existing approaches, it is useful to know whether there is further room for improvement. In figure 8, we compare our scheme against an approximation to the best-found performance. For FM we reach this maximum, but for other programs such as MP3DECODER, INSERTIONSORT and VOCODER, there is significant room for improvement. So although our approach outperformed all prior techniques on VOCODER, it could have done better. Overall there is a 2.5x average maximum speedup available and we achieve 60% of that maximum performance.

6.2 Explanation

In order to get further insights into the different models, we investigate partitions generated by different approaches of three selected StreamIt benchmarks.

RADIXSORT. This application has a regular parallel structure: it is pure pipeline parallelism; 10 out of its 13 filters have exactly the same computation-communication ratio. For this program, both the dynamic-programming based and the greedy-based algorithms give a partition that has the Best-Found performance. Thus, our approach is not able to improve their results.

LATTICE. This application contains 36 filters with both data (i.e., splitjoin) and pipeline parallelism as shown in figure 9 (a). Finding a good partition for it is certainly nontrivial and different approaches give different answers. Figures 9 (b) to (e) illustrate the partitions given by four partitioners: the StreamIt default, the greedy-based, the analytical-based, and our `ML`-based partitioners, respectively. The StreamIt default partitioner aims to form a balanced pipeline and generates a partition with four nodes. This partition outperforms the solutions given by the greedy-based and the analytical model-based partitioners, which generate partitions with more threads at the scheduling stage bringing extract runtime overhead. In contrast, our approach generates a coarse-grain stream graph, which has relatively fewer number of threads and lower communication cost. As a result, the `ML`-based approach achieves better performance than other techniques. By examining this application, we discover that the computation of LATTICE is relatively small compared to the communication cost on the 4-core platform. Therefore, a good partitioning strategy will try

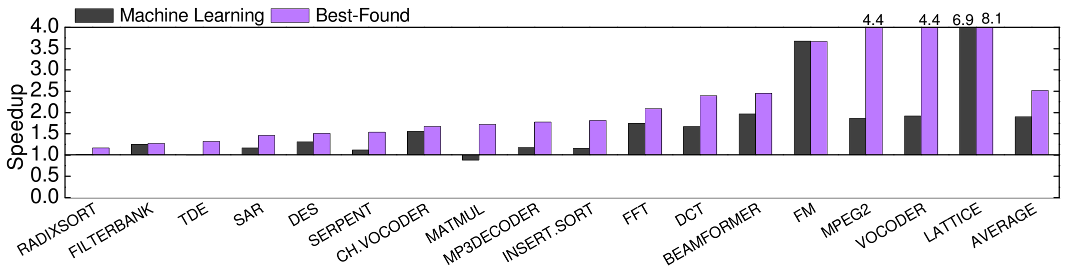


Figure 8: The performance of our approach vs the best performance found out of on average 3000 executions per program on the 4-core platform.

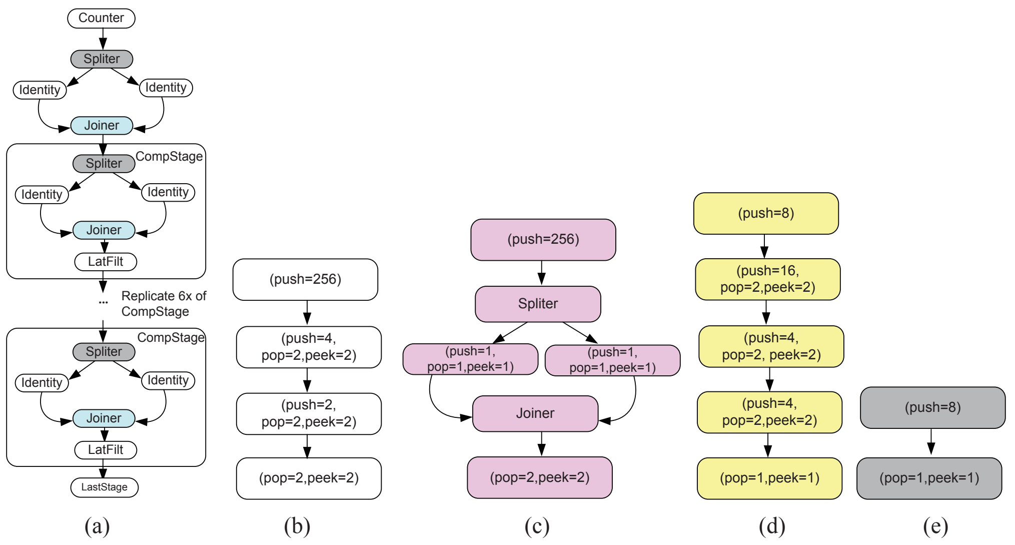


Figure 9: The stream graph (a) and partitions generated by different approaches for LATTICE. Each box represents a filter and the communication rate of each filter is denoted. The dynamic programming based partitioner gives (b), the greedy partitioner gives (c), the analytical model gives (d), and our ML based approach gives (e) by coarsening the stream graph to reduce communication overhead.

to coarsen the stream graph to reduce communication overhead. In this case, our approach identifies the program characteristics of LATTICE and applies a appropriate heuristic to aggressively coarsen the stream graph.

VOCODER. The stream graph of VOCODER has over 120 filters containing large splitjoin sections and long-stage pipelines. Unlike LATTICE's partitioning strategy, merely considering coarsening the stream graph is not the right choice for this application. Figures 10 (a) and (b) correspond to the partitions given by the StreamIt default scheme and our approach respectively. This time, our ML-based approach takes a different strategy. In order to reduce communication overhead, it first coarsens those small computation kernels. At the same time, it exploits data parallelism in the critical path (which contributes to around 40% of the total computation) and generates a 9-node partition. When compared with the Best-Found solution, a 13-node partition, our ML-based model could be smarter by predicting a more aggressive partitioning goal.

As indicated by these examples, we can see that different partitioning strategies should be applied to applications with different program characteristics (section 6.3.3 gives detailed discussion about the program characteristics). Essentially, the analytical model and the two StreamIt partitioners are ”one-size-fits-all” strategies. They can be improved by a program-aware partitioning scheme. Developing such a scheme by hand is, however, extremely hard. Our approach, on the other hand, solves this problem by leveraging machine learning techniques. It uses prior knowledge to select and apply the program-specific partitioning strategy according to program characteristics of the target program, resulting in better performance than hardwired heuristics.

6.3 Analysis of Results

In this section we analyse the behaviour of our approach. We first evaluate how accurate the nearest neighbour model is. This is then followed by an evaluation of how useful our feature space is in distinguishing good partitions. Finally, we examine the structure of the best partitions found and examine what optimisation criteria are important in delivering performance.

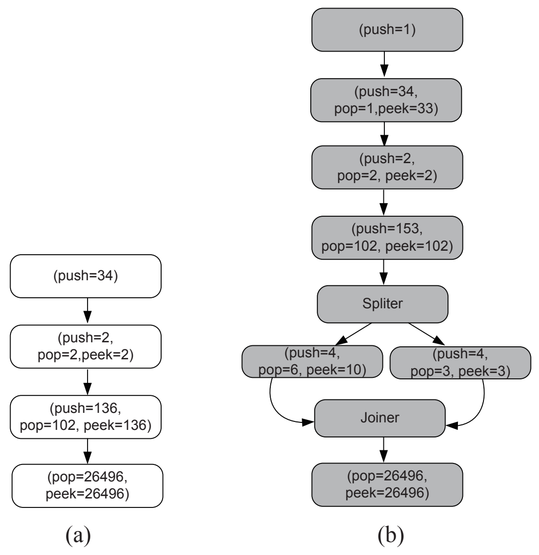


Figure 10: Partitions generated by the StreamIt default method (a) and our approach (b) for VOCODER. Each box represents a filter in which the communication rate is denoted. The StreamIt default scheme exploits purely pipeline parallelism. Our ML-based approach exploits both pipeline and data parallelism.

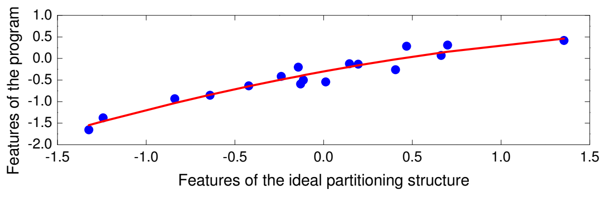


Figure 11: Correlation between program features and the ideal partitioning structure for each of the 17 benchmarks. This figure shows there is strong correlation between program features of a program and its ideal partitioning structure. Note that the feature vector of each program has been reduced into a single value to aid visualization.

6.3.1 *Correlation of Program Features*

The intuition behind our predictive model is that similar programs will have similar ideal partitioning structures as long as we are able to have features that capture similarity accurately. Figure 11 confirms this assumption. It shows the program features of the ideal partitioned program vs the features of the original program for each of the benchmark. The original multi-dimensional feature vectors have been projected into a single value for each program to aid clarity. This figure shows a strong correlation between the program features and the ideal partitioning structure. We can quantify this by using the *correlation coefficient* [4]. It takes a value between -1 and 1, the closer the coefficient is to $+/-1$, the stronger the correlation between the variables. It is 0.9 in our case , which indicates a high correlation between program features and the ideal partitioning structure. The means the premise for the nearest neighbour model is valid.

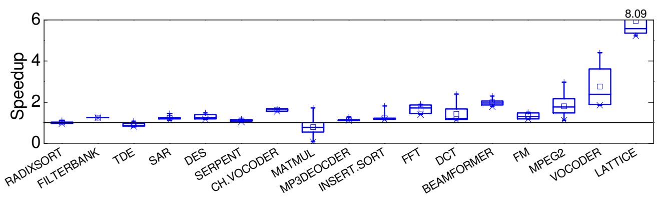


Figure 12: Performance of mappings around the predicted ideal partitioning structure. The distance of each program to the predictive ideal partitioning structure is evaluated with weighted Euclidean distance. The x-axis represents the program and the y-axis represents the speedup relative to the StreamIt default mapping. The central box denotes the mean speedup and the top and bottom of each box represent the highest and the lowest speedup using our predictive model.

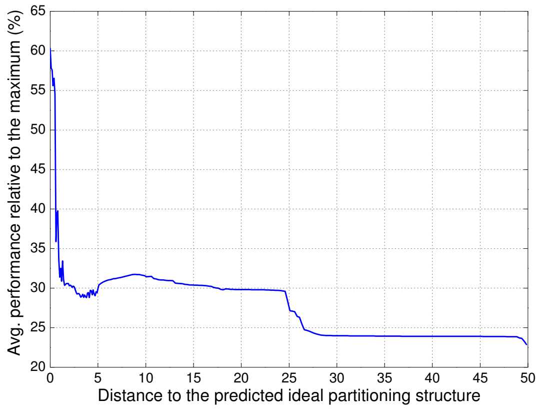


Figure 13: Average performance relative to upper bound vs distance from ideal partition. This figure shows the average performance of a partition as a function of its distance from the predicted ideal structure. As the distance decreases, performance improves.

6.3.2 *Distance-based Mapping Selection*

The box plots in figure 12 summarises the performance of partitions around a predicted ideal partitioning structure. It shows the performance of partitions with a normalised distance of less than 0.5 to the predicted ideal partitioning structure, as used by our scheme. The diagram shows that regions around the predicted ideal result in good performance. The top and the bottom of the "whisker" of each program represent the highest and the lowest speedup found in the region around the predicted ideal. For the majority of programs we obtain significant performance improvement if we can generate a mapping that is closer to the predicted ideal partitioning structure. The one exception is MATMUL, as seen in figure 8. If we zoom in on VOCODER, we see that the average speedup obtained in this region is 2.7. The lower value is 1.9 and the upper is 4.4. If we look at figure 8, we see that our scheme selects a partition that achieves just a 1.9 speedup - the lower bound, while the best performance is 4.4 - the upper bound. This shows that our

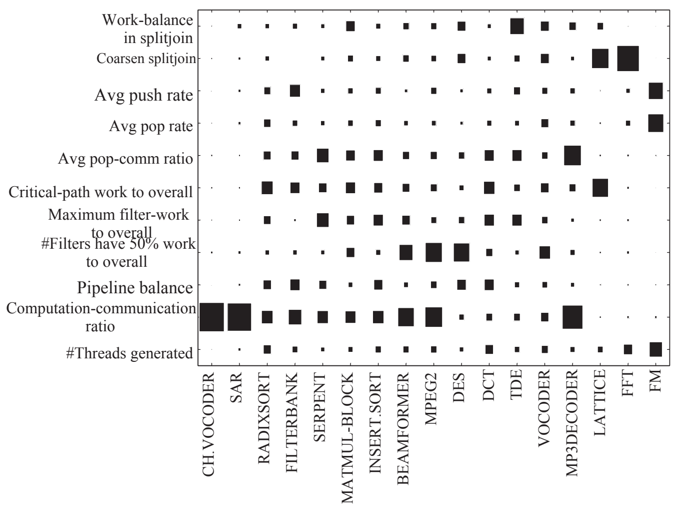


Figure 14: A Hinton diagram showing the partitioning objectives that are likely to impact performance of each benchmark. The larger the box, the more likely an partitioning objective affects the performance of the respective program.

scheme could improve if was smarter in choosing the ideal structure within a good cluster for this program.

Figure 13 shows how the performance of a partitioning structure varies as a function of its distance from the predicted ideal partitioning structure. This diagram averages the results across all benchmarks and shows that partitions near the predicted ideal give the best average performance. The figure demonstrates that Euclidean distance from the predicted ideal structure is a useful means of discriminating good partitions from poor.

6.3.3 *Importance of Partitioning Choices on Performance*

Section 2 has shown that the best heuristic varies across programs. We now consider the importance of specific partitioning characteristics for each program on the 4-core platform. We have considered a number of characteristics that a partitioning algorithm may wish to consider in making partitioning decisions e.g. communication computation ratio, average push rate etc. Figure 14 shows a Hinton diagram illustrating the importance of a number of different partitioning objectives on the performance of each program. Intuitively, this information gives us an indication of those characteristics on which an optimising heuristic should focus. The larger the box, the more significant the issue for that program. The x-axis denotes the programs, the y-axis denotes partitioning criteria. Figure 14 shows that each of these objectives has an impact on each program. The computation to communication ratio is important for all programs and extremely important for the CHANNELVOCODER and SAR. Having a balanced pipeline, however, is less important overall. Some programs are sensitive to all of these objectives, e.g. RADIXSORT while for some program e.g. FFT, one issue, coarsen the splitjoin sections, is of overwhelming importance. This diagram illustrates just how hard it is for a heuristic which typically focuses on one or two objectives, to find the best partitioning for all programs.

6.4 Adapting to a New Platform

In order to evaluate the portability of our model, we evaluated it on a 2x Quad-core Intel Xeon (8-core) platform. Training data was collected from the new platform and used to train the model. Note that we used the same program features and methodology to train the model as used on the 4-core platform. Due to time constraints we had to use relatively fewer partitions for each training program. This will affect the performance of our model.

Figure 15 shows the performance of the different approaches on the 8-core platform. The most striking result is that for some applications, the greedy partitioner does better than it does on the 4-core platform. For DES and FFT, the greedy partitioner achieves 1.28x and 1.78x speedup respectively compared to the default dynamic programming partitioner when on the 4-core platform, it slowed down these programs to 70% of the partition generated by the StreamIt default. The greedy partitioner improves the performance of 10 benchmarks on this platform, up to 4.9x for VOCODER, with an average 1.2x speedup but gives significant performance slowdowns for 5 programs. The analytical-based model also gives unstable results. It gives an average speedup but only gives noticeable improvement on 4 programs. It, however, gives a significant slow down on FILTERBANK by more than a factor of 2.

In contrast to those two approaches, our machine learning approach is more stable across programs, with just one small slowdown in the case of RADIXSORT. On average we achieve 1.8x improvement across the benchmarks. With the correct amount of training data, this is certain to improve even further. Compared to the analytical model and the greedy heuristic, our model is stable not only across programs but also across platforms. This example demonstrates the portability of our machine learning approach.

7. RELATED WORK

Stream Graph Partitioning. There is a significant volume of prior work dealing with stream graph partitioning. Liao *et al.* build an affine partitioning framework to map streaming parallelism onto multi-core processors [20]. The ACTOES compiler [24] uses a static graph partitioning algorithm to map stream programs onto streaming processors. Navarro *et al.* [25] construct an analytical model to determine parallelism configurations for pipeline parallelism which contains data parallel pipeline stages. Stream graph modulo scheduling (`SGMS`) orchestrates execution of StreamIt applications for the IBM Cell processor [16]. `SGMS` uses integer linear programming (`ILP`) formulations to perform partition by aiming to overlap the communication and computation. Similar `ILP` solver-based approach has also been used to generate partitions for StreamIt applications targeting on GPUs [39]. Formulating `ILP` models, however, requires expert knowledge on the underlying architecture. Finding a solution under certain constraints for the `ILP` formula can be time-consuming too. Recently, MacroSS [13] has been proposed to exploit macro-level SIMD for streaming applications on SIMD processors. MacroSS groups filters together to utilize the SIMD engine, which achieves good performance on Intel Core i7 processor. Rather than developing a hard-wired heuristic by hand, we use machine learning techniques to build a portable predictive model to perform stream graph partitioning based on prior knowledge.

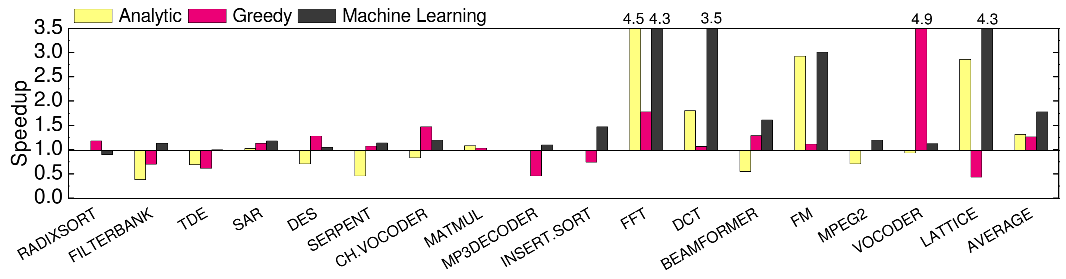


Figure 15: Performance comparison on the 8-core platform for the analytical model, the greedy partitioner and the ML-based model.

Runtime Scheduling for Streaming Parallelism. A Runtime scheduler uses dynamic information to execute a partitioned stream application (graph). Gordon *et al.* exploited coarse-grain task, data and pipeline-level parallelism incorporating with a dynamic task scheduler. Their approach firstly uses a greedy partitioner to partition a stream graph, then uses a runtime scheduler to execute the partitioned streaming graph onto multi-cores [11]. FlexStream [14] is a runtime adaptation system that dynamically re-maps an already partitioned stream graph according to the number of processors available for heterogeneous multi-core systems. Aleen *et al.* [1] combine profiling information and execution time estimation to predict the dynamic behavior of a stream application. This profiling information then is used to dynamically adjust the pipeline to achieve load balance. Our machine learning based approach automatically adapts to the behavior of the runtime scheduler through training. Our approach is orthogonal to existing and future scheduling approaches, which significantly reduces the efforts of porting a stream graph partitioner to new architectures or runtime systems.

Machine Learning Based Compilation. Machine learning has been successfully applied to optimising both sequential and data parallel programs. Stephenson *et al.* [34] have used genetic algorithms to tune the compiler heuristics for sequential programs. Similar approaches also have been used to find either good loop transformations [27] or compiler flags [15]. In contrast to these online-learning algorithms, our predictor learns from prior knowledge obtained by *off-line* training, hence, it has a much lower overhead in deployment. McGovern and Moss [23] use supervised learning to decide the scheduling order of instruction pairs. Stephenson and Amarasinghe [22] used supervised classification techniques such as *K-Nearest Neighbour* and *Support Vector Machine* to find the loop unroll factor. Both approaches use off-line training but only target on sequential programs. In our previous work [41], we have built machine learning models to map data parallelism onto multi-cores. The Qilin compiler [21] divides parallel loops between CPUs and GPUs using regression-based prediction. All approaches focus on *fixed* targets, such as per loop or a fixed number of tasks. This work is distinct from previous ones in that we target on streaming parallelism which contains both data and pipeline parallelism and the prediction target (e.g., the transformation sequence) is unbounded.

Automatic Program Generation. Automatic program generators have demonstrated their success either in generating ”similar” sequential programs from existing benchmark suites [40] or in optimising domain-specific applications [6]. Our approach, on the other hand, aims at generating small ”different” streaming parallel programs as training examples for machine learning models.

8. CONCLUSIONS

This paper has presented an automatic and portable compiler based approach to partitioning streaming programs for multi-cores, providing a significant performance improvement over hardwired heuristics. Using machine learning techniques, our compiler predicts the ideal partition structure of a streaming application, allowing us to quickly search the transformation space without running the code. In addition to the predictive model, we have developed a micro-kernel streaming program generator which automatically generates small training examples for the predictive model. We demonstrated our approach by mapping StreamIt applications onto two multi-core platforms. On average, we achieve a 1.90x speedup over the StreamIt default scheme on a 4-core platform. Compared to a recently proposed analytical-based model, our approach achieves on average a 1.79x performance improvement. When our approach was ported to an 8-core machine, we were able to achieve a 1.80x improvement over the StreamIt default. Future work will consider incorporating our compiler framework with runtime task scheduler to dynamically exploit task, data and pipeline parallelism.

9. ACKNOWLEDGMENTS

We would like to thank members of Compiler and Architecture Design Group at Edinburgh University, especially, Björn Franke, Alastair Murray, Christian Fensch and Georgios E. Tournavitis, for their kind help and constructive feedback. We gratefully thank the anonymous reviewers for their very useful comments that help to improve the quality of this paper. We acknowledge the use of the StreamIt compiler (version 2.1.1) for experiments. This work has made use of the resources provided by the Edinburgh Compute and Data Facility (`ECDF`). (http://www.ecdf.ed.ac.uk/). The `ECDF` is partially supported by the eDIKT initiative (http://www.edikt.org.uk).

10. REFERENCES

[1] F. Aleen, M. Sharif, and S. Pande. Input-driven dynamic execution prediction of streaming applications. In *PPoPP*, 2010.

[2] K. Asanovic, R. Bodik, and J. Demmel et al. A view of the parallel computing landscape. *Commun. ACM*, 52(10), 2009.

[3] G. Bikshandi, J. Guo, and D. Hoeflinger et al. Programming for parallelism and locality with hierarchically tiled arrays. In *PPoPP*, 2006.

[4] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[5] T. N. Bui and C. Jones. Finding good approximate vertex and edge partitions is NP-hard. *Inf. Process. Lett.*, 42(3), 1992.

[6] S. Chellappa, F. Franchetti, and M. Püeschel. Computer generation of fast fourier transforms for the cell broadband engine. In *ICS*, 2009.

[7] D. E. Culler, R. M. Karp, and D. Patterson et al. LogP: a practical model of parallel computation. *Commun. ACM*, 39(11), 1996.

[8] W. C. Dan, W. yu Chen, and Parry Husbands et al. A performance analysis of the berkeley upc compiler. In *ICS*, 2003.

[9] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[10] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, 2006.

[11] M. I. Gordon, W. Thies, and M. Karczmarek et al. A stream compiler for communication-exposed architectures. In *ASPLOS*, 2002.

[12] H. Hofstee. Future microprocessors and off-chip sop interconnect. *Advanced Packaging, IEEE Transactions on*, 2004.

[13] A. Hormati, Y. Choi, and M. Woh et al. MacroSS: Macro-simdization of streaming applications. In *ASPLOS*, 2010.

[14] A. H. Hormati, Y. Choi, and M. Kudlur et al. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *PACT*, 2009.

[15] K. Hoste and L. Eeckhout. COLE: compiler optimization level exploration. In *CGO*, 2008.

[16] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI*, 2008.

[17] M. Kulkarni, K. Pingali, and B. Walter et al. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[18] Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4), 1999.

[19] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proc. IEEE*, 75(9), 1987.

[20] S. Liao, Z. Du, and G. Wu et al. Data and computation transformations for brook streaming applications on multiprocessors. In *CGO*, 2006.

[21] C. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, 2009.

[22] S. Mark and A. Saman. Predicting unroll factors using supervised classification. In *CGO*, 2005.

[23] E. Moss, P. Utgoff, and J. Cavazos et al. Learning to schedule straight-line code. In *NIPS*, 1997.

[24] H. Munk, E. Ayguadé, and C. Bastoul et al. ACOTES Project: Advanced Compiler Technologies for Embedded Streaming. *International Journal of Parallel Programming*, 2010.

[25] A. Navarro, R. Asenjo, and S. Tabik et al. Analytical modeling of pipeline parallelism. In *PACT*, 2009.

[26] D. Pelleg and A. W. Moore. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *ICML*, 2000.

[27] L. Pouchet and C. Bastoul et al. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In *PLDI*, 2008.

[28] K. Ramamritham and J. A. Stankovic. Dynamic task scheduling in hard real-time distributed systems. *IEEE Softw.*, 1(3), 1984.

[29] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: concurrent programming for modern architectures. In *PPoPP*, 2007.

[30] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM J. Res. Dev.*, 35(5-6), 1991.

[31] G. Schwarz. Estimating the dimension of a model. *Ann. Statist*, 6(2), 1978.

[32] T. Sherwood, E. Perelman, and G. Hamerly et al. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.

[33] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[34] M. Stephenson, S. Amarasinghe, and M. Martin et al. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Not.*, 2003.

[35] J. Subhlok, J. M. Stichnoth, and David R. O'hallaron et al. Exploiting task and data parallelism on a multicomputer. In *PPoPP*, 1993.

[36] B. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *CC*, 2001.

[37] W. Thies. *Language and Compiler Support for Stream Programs*. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 2009.

[38] G. Tournavitis, Z. Wang, and B. Franke et al. Towards a holistic approach to auto-parallelization - integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI*, 2009.

[39] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software Pipelined Execution of Stream Programs on GPUs. In *CGO*, 2009.

[40] L. Van Ertvelde and L. Eeckhout. Dispersing proprietary applications as benchmarks through code mutation. In *ASPLOS*, 2008.

[41] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *PPoPP*, 2009.