2012 IEEE/ACM 45th Annual International Symposium on Microarchitecture

Kernel Partitioning of Streaming Applications: A Statistical Approach to an NP-complete Problem

Petar Radojković[1,2] Paul M. Carpenter[1,2] Miquel Moretó[1,2,4] Alex Ramirez[1,2] Francisco J. Cazorla[1,3]
[1]Barcelona Supercomputing Center [2]Universitat Politècnica de Catalunya - Barcelona TECH
[3]Spanish National Research Council (IIIA-CSIC) [4]International Computer Science Institute, Berkeley
{petar.radojkovic, paul.carpenter, miquel.moreto, alex.ramirez, francisco.cazorla}@bsc.es

Abstract

One of the greatest challenges in computer architecture is how to write efficient, portable, and correct software for multi-core processors. A promising approach is to expose more parallelism to the compiler, through the use of domain-specific languages. The compiler can then perform complex transformations that the programmer would otherwise have had to do. Many important applications related to audio and video encoding, software radio and signal processing have regular behavior that can be represented using a stream programming language. When written in such a language, a portable stream program can be automatically mapped by the stream compiler onto multicore hardware. One of the most difficult tasks of the stream compiler is partitioning the stream program into software threads. The choice of partition significantly affects performance, but finding the optimal partition is an NP-complete problem.

This paper presents a method, based on Extreme Value Theory (EVT), that statistically estimates the performance of the optimal partition. Knowing the optimal performance improves the evaluation of any partitioning algorithm, and it is the most important piece of information when deciding whether an existing algorithm should be enhanced. We use the method to evaluate a recently-published partitioning algorithm based on a heuristic. We further analyze how the statistical method is affected by the choice of sampling method, and we recommend how sampling should be done. Finally, since a heuristic-based algorithm may not always be available, the user may try to find a good partition by picking the best from a random sample. We analyze whether this approach is likely to find a good partition. To the best of our knowledge, this study is the first application of EVT to a graph partitioning problem.

1. Introduction

Stream programming is suitable for applications that process long sequences of data, such as voice, image, multimedia content, Internet and communication traffic. Stream programming languages such as StreamIt [8, 39], Brook [7] and SPM [2, 10] represent the program as concurrent kernels, which communicate only via point-to-point streams. A kernel is a basic computation block with a user-defined function that processes one or more input data streams into one or more output data streams. Dependencies between different kernels are described explicitly through the communication data channels. The whole application can be represented as a *stream graph*. The nodes of the stream graph correspond to the kernels, while the directed edges represent the communication data channels.

There are three main advantages of stream programming languages, compared with traditional languages such as *C*. First, a stream programming language is a domain-specific language, which provides a natural way to describe streaming applications. Second, when the program is described using a stream language, the compiler may perform complex optimizations over the stream graph, producing an efficient multithreaded program. Some optimizations involve major changes to the program's structure and data layout. Third, unlike some other parallel programming models, including multithreading, a stream program is deterministic, and therefore easier to debug.

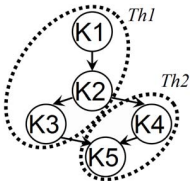


Figure 1: A kernel partition example

In order to take advantage of multiple processor cores the stream program is automatically compiled into a multithreaded executable by the stream compiler. One of the most important tasks of the stream compiler is to partition the kernels in the stream graph into software threads. In Figure 1, we illustrate a stream graph of a simple program, which is comprised of five kernels (K1 to K5). The figure also shows one possible kernel partition of the graph: kernels K1, K2, and K3 are to be compiled to software thread *Th1*, while kernels K4 and K5 are to be compiled into *Th2*.

Kernel partitioning can significantly affect the overall system performance. For example, for the benchmarks included in the StreamIt 2.1.1 suite, the relative performance difference between good and bad kernel partitions of the same benchmark mapped into four software threads ranges from 2.4× to 3.9×, and on average it is 3.5×.

The optimal kernel partition cannot be determined because the essence of the analysis is graph partitioning, which is an NP-complete problem [18, 20]. Due to the large exploration space, brute force exploration is impractical: as streaming applications comprise tens or hundreds of interconnected kernels (54 kernels on average in the StreamIt 2.1.1 suite), the number of possible kernel partitions is vast. For example, the *channelvocoder* benchmark has 55 kernels, and it can be distributed into 10^{22} partitions of exactly four software threads. The number of possible kernel partitions increases rapidly with the number of output software threads, so the number of partitions using eight threads is 10^{34}.

Several studies (see Section 6) propose heuristic-based algorithms to address the kernel partitioning problem. As kernel partitioning is an intractable problem, it is impossible in general to know the performance of the optimal partition, so the room for improvement is also unknown. It is hard to decide whether to invest additional effort to try to improve a given algorithm, since it may already be close to optimal.

In this paper, we present a statistical approach to the kernel partitioning problem. We present a method that predicts the performance of the optimal partition based on the observed performance of each kernel partition in a random sample. The method is based on Extreme Value Theory (EVT), a branch of statistics that analyzes extreme de-

1072-4451/12 $26.00 © 2012 IEEE
DOI 10.1109/MICRO.2012.44

viation from the population median. We also present several different approaches to generate the random samples. Finally, we show that the performance of the best observed kernel partition in a random sample is likely to be close to the optimal one. If a heuristic is not available for the user's problem, a good kernel partition can be found using random sampling on its own.

The main contributions of our study are:

- We present a method based on EVT that statistically estimates the performance of the optimal kernel partition. To the best of our knowledge, this is the first study that applies EVT to a graph partitioning problem.
- We show that the sampling method, used to generate random partitions, has a significant effect on the applicability of the statistical method. We analyze different sampling methods, and our results strongly recommend that the samples should be uniformly distributed.
- We use the estimates of the optimal performance to evaluate a state-of-the-art heuristic-based kernel partitioning algorithm.
- Finally, since a complex heuristic-based algorithm may not always be available, the user may pick the best from a random sample, and measure its quality using the estimates of the optimal performance. We analyze whether random sampling is likely to find a good kernel partition.

The presented analysis is evaluated for the benchmarks included in the StreamIt 2.1.1 suite. The method based on EVT successfully estimates the performance of the optimal kernel partitions for all the benchmarks under study. In all experiments, the two different kernel partitioning methods, the heuristics-based algorithm and the method based on random sampling, detected kernel partitions with practically the same performance.

The rest of the paper is organized as follows. Section 2 describes metrics that can be used to measure the performance of streaming applications. Section 3 discusses methods to generate random samples of kernel partitions. Section 4 presents the statistical analysis that we use to estimate the performance of the optimal kernel partition. In Section 5, we apply the presented analysis to the StreamIt 2.1.1 benchmark suite, and evaluate the results. Section 6 describes related work, and Section 7 presents the conclusions of the study.

2. Background

In this section, we describe different metrics that could be of interest when doing kernel partitioning. We also briefly describe related work with a focus on the conclusions that directly affect our study.

2.1. Target metric

There are several metrics that can describe the performance of streaming applications. In our study, we analyze the *cost* of streaming applications. The *cost* is proportional to the time needed to process a fixed amount of input data. This metric corresponds to execution time for non-streaming applications. Other metrics include energy or power, the hardware utilization of the target architecture, or some weighted sum of them.

The input to the statistical analysis is the cost of each kernel partition in a random sample. The costs are generated using metrics from the StreamIt 2.1.1 compiler. Instead of using the streaming compiler, the target metric could alternatively have been measured using real execution or simulation.

The proposed statistical approach and the general conclusions of this study are independent of both the target metric and the way in which the metric is evaluated: program compilation, execution or simulation. It is important to note, however, that the results from the statistical analysis are clearly dependent on the quality of the samples provided to it.

If the application behavior is sensitive to its input data, which is generally not the case for streaming applications, the user should consider the analysis for different input datasets that are representative for different application behavior.

If the user wants to use the statistical method for multiple objective functions separately, then it is only necessary to do a full set of compilations, executions or simulations once. After obtaining a complete set of metrics, the statistical analysis can be done multiple times using different metrics.

2.2. Convexity constraint

Carpenter et al. [9] present a partitioning and allocation algorithm for an iterative stream compiler. The algorithm produces kernel partitions that are easier to compile and that require short pipelines of software threads. The authors evaluate their proposal on the benchmarks included in the StreamIt 2.1.1 suite.

One of the conclusions in that paper is that the kernel partition should be convex. A kernel partition is convex if the dependencies between different software threads form an *acyclic* graph. This means that every directed path between two kernels in the same software thread is internal to that thread. The reason for the convexity constraint is that the choice of partition affects the length of the pipeline generated by the streaming compiler. The convexity constraint controls the length of that pipeline. The authors demonstrate that without the convexity constraint, the compiler may generate long pipelines of software threads, which increases memory use and latency of the inter-thread communication, significantly affecting the overall performance.

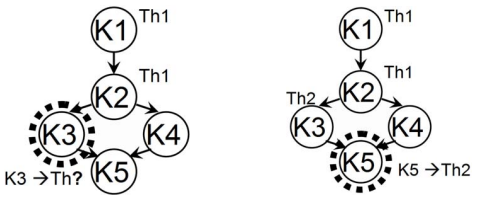
We follow these instructions and focus our study on the analysis of *convex* kernel partitions. We pay special attention to generating random samples comprised only of kernel partitions that satisfy the convexity constraint. It is important to notice, however, that convexity is not a requirement of the proposed statistical approach. The approach can be also used for the analysis of non-convex kernel partitions.

Although the convexity constraint significantly reduces the number of kernel partitions, their number is still vast, and brute force exploration is impractical. For example, the fm benchmark can be distributed into 10^{12} convex kernel partitions of exactly four software threads, radar into 10^{14} partitions, filterbank into 10^{20}, and vocoder into 10^{23}.

3. Sampling methods

In order to apply Extreme Value Theory (EVT) to the kernel partitioning problem for streaming applications, we need to generate random convex partitions of the stream graph that are independent and identically distributed (*i.i.d.*). Intuitively, random variables are independent if knowing the value of one of them gives no new information about the values of the others; they are identically distributed if they all have the same probability distribution, which does not have to be uniform. Uniform distribution would mean that each kernel partition would be selected with the same probability.

The different methods we used to select the random *i.i.d.* kernel partitions are described next. For each method, we describe how to select a single random kernel partition. To generate a sample of N *i.i.d.* kernel partitions, repeat the sampling method N times.



(a) Assigning kernel K3 (b) Assigning kernel K5

Figure 2: DFS sampling method

3.1. Depth-First Search (DFS)

The first sampling method generates a random kernel partition using a depth-first search (DFS) of the stream graph. The kernels are visited in sequence, each kernel being assigned with equal probability to any software thread that would not violate the convexity constraint. Thus, this method generates random kernel partitions in a single traversal of the stream graph.

We illustrate the sampling method with the example shown in Figure 2. The example stream graph contains five kernels (K1 to K5) that are to be compiled into two software threads (Th1 and Th2). For example, assume that kernels K1 and K2 are already assigned to Th1, and that the next kernel to be assigned is K3 (Figure 2(a)). K3 can be assigned with equal probability to Th1 or Th2. If K3 is assigned to Th2, kernel K5 has to be assigned to Th2 in order to generate a convex partition (see Figure 2(b)). Since the number of candidate threads that do not break the convexity constraint decreases rapidly, the DFS sampling method often generates kernel partitions with an unbalanced number of kernels per threads.

3.2. Edge Contraction (EC)

The second sampling method generates a random partition using edge contraction of the stream graph. Initially, each kernel is placed in its own cluster. Then, the edges of the stream graph are visited in random order. In each step the selected edge of the graph is contracted by fusing the clusters connected through this edge. If the resulting graph violates the convexity constraint, the contraction is undone. The process is continued, by moving to the next edge in random sequence, until the number of clusters equals the number of software threads. Finally, the clusters are randomly assigned to the threads.

Figure 3 illustrates the EC kernel partitioning of a simple stream graph into two software threads. For example, assume that K1→K2 edge is selected as the first edge to be contracted. In this case, kernels K1 and K2 are fused into a single cluster while the rest of the graph is not modified. Afterwards, we illustrate the contraction of edges K3→K5 and (K1&K2)→K4. Finally, the clusters are randomly distributed among software threads. In comparison with DFS, the EC sampling method generates random kernel partitions with a balanced number of kernels.

3.3. Edge Contraction with Filter (EC-F)

The third sampling method is an enhancement of the EC method, designed to bias the sampling towards kernel partitions with a low cost, which will lead to good application performance. The EC-F method selects the edges of the stream graph in random order, fuses the corresponding clusters, and checks whether the convexity constraint is violated, as for the EC sampling method. The only difference is that EC-F performs an additional check: if contracting the current edge generates a cluster with a high cost (i.e. that exceeds a given

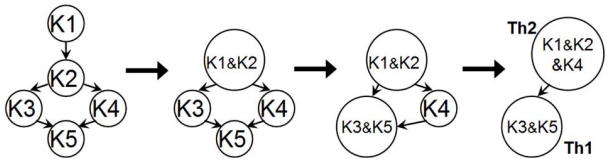


Figure 3: EC sampling method: Contracting K1→K2, K3→K5, and (K1&K2)→K4 edges, respectively.

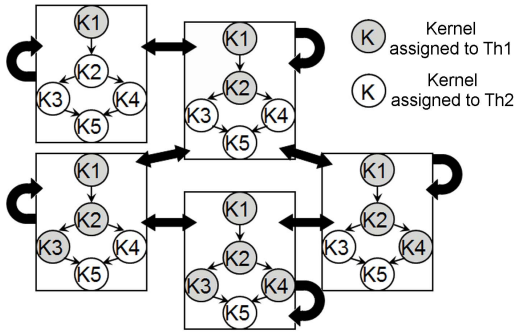


Figure 4: UD sampling method: Example partition graph for a small stream program

threshold), the contraction is undone and the process is repeated for a different edge. In the experiments presented in the paper, the threshold was the lowest cost detected in ten random kernel partitions generated using the EC sampling method.

It may happen that, although the number of clusters is still greater than the number of threads, none of the edges can be contracted without creating a cluster of cost exceeding the threshold. In this case, the remaining edges are visited in a new random order, and edges are contracted without checking whether the cost of the final clusters is over the threshold. Finally, the clusters are then randomly assigned to threads. In contrast with other presented sampling methods, this method does take into account the cost of each particular kernel when generating partitions. The main target of this algorithm is to generate random partitions with a balanced cost among clusters.

3.4. Uniformly Distributed (UD) sampling

The final sampling method generates a uniform sampling distribution of the kernel partitions. This means that each convex kernel partition is selected with the same probability. It is important to notice that the statistical method used in the study does not require the kernel partitions to be uniformly distributed, since it only requires their costs to be independent and identically distributed (*i.i.d.*). In general, previous sampling methods do not provide uniformly distributed kernel partitions.

This sampling method comprises three steps.

Step 1: We analyze different kernel partitions using the *partition graph*, the graph of all possible convex partitions of the stream graph under study. Each node of the partition graph is a different kernel partition, so the number of nodes is equal to the number of partitions. There is an undirected edge between two nodes of the partition graph if they differ in the assignment of exactly one kernel partition. Also, each node contains a self-loop edge, an edge that connects the node to itself. Due to its large size (this is an NP-complete problem), the partition graph is never actually constructed in its entirety. An example partition graph, for a small stream program that is to be assigned to two software threads, is shown in Figure 4.

Step 2: We perform a random walk on the partition graph. First, we have to choose an initial node of the partition graph to start the

random walk. This node can be selected by any method that generates random kernel partitions. In the experiments presented in our study, the initial kernel partition (initial node of the partition graph) is selected using the EC sampling method. The random walk starts from the initial node in the partition graph and calculates all its neighbors. Then it randomly chooses one of the neighbors (using a uniform distribution) to be the next node that is to be visited. The neighbor selection is repeated N times, with N large enough to potentially visit all the nodes of the partition graph of the benchmarks used in the study[1]. The last node of the partition graph that is visited is the outcome of the random walk. The probability that a given node is selected using the random walk is directly proportional to its degree (the number of its neighbors in the graph) [32]. As we know the probability of each visited node of the partition graph to be selected, the random walk generates samples with a known distribution.

Step 3: In the final step of this sampling method, we convert the output of the random walk from a known distribution to a uniform distribution. In order to do so, each kernel partition selected using the random walk is included in the outcome of this sampling method with a probability that is inversely proportional to its degree in the partition graph. This way, every convex partition has the same probability of being selected, i.e. the method provides uniformly distributed samples.

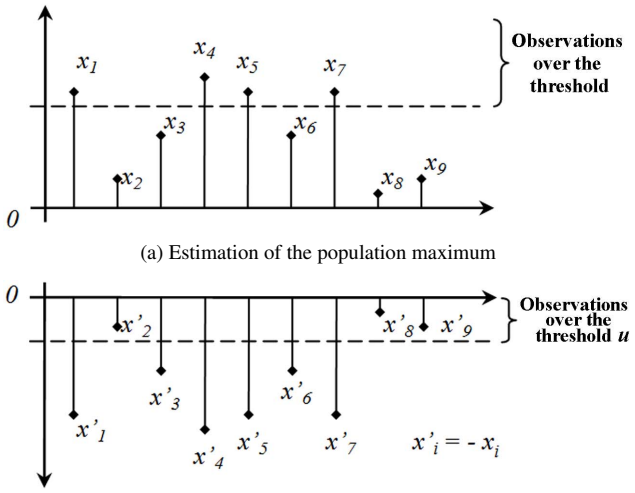
3.5. Statistical *i.i.d.* tests

The sampling methods described in the previous section are designed to generate random *i.i.d.* samples. After generating the samples, we perform statistical tests to confirm that they are indeed independent and identically distributed.

Wald–Wolfowitz test: The *Wald–Wolfowitz* test or *runs* test examines whether the observations in the sample are mutually independent [6, 16]. The test comprises two main steps. First, the costs of kernel partitions (non-negative real numbers) have to be converted into binary values. We converted the cost of a given kernel partition to *'0'* if its value was below the median cost in the sample, and converted it to *'1'* otherwise. This way, the sequence of non-negative real numbers was converted into a sequence of *0s* and *1s*, e.g. *000110000*. In the second step, the test analyzes the sub-sequences of consecutive identical values (*0s* or *1s*), which are referred to as *runs*. For example, the sequence *000110000* is composed of three runs: *000*, *11*, and *0000*. The Wald–Wolfowitz test validates that the observations in the sample under study are mutually independent if the lengths of the runs follow a Gaussian distribution [6]. The mutual independence hypothesis was tested at the 0.05 significance level. All the samples used in the study passed the test.

Kolmogorov–Smirnov test: In order to validate that selected kernel partitions in a given sample are identically distributed, we used a two-sample *Kolmogorov–Smirnov* test [16, 19]. The test compares the empirical cumulative distribution functions (ECDF) of two data sets and, based on the maximum distance between the two ECDFs, it confirms or rejects the hypothesis that the data sets correspond to the same distribution. The identically distributed test that we performed contains three steps. First, we generated a random sample of 20,000 kernel partitions and observed the cost of each partition. The costs of the kernel partitions in the sample followed the order in which the partitions were generated. Second, in each experiment, we observed two randomly-selected segments of m consecutive values from the original sample. Finally, we used a two-sample *Kolmogorov–Smirnov* test to check whether the randomly selected segments of the sample have the same probability distribution. If the kernel partition costs are indeed identically distributed, then all segments of consecutive values in the sample have the same distribution. For each sample used the study, we performed the test for segments of m =100, 500, 1,000 and 5,000 observations. All the samples used in the study passed the test at the 0.05 significance level.

[1] In our experiments, $N = 100$.



(a) Estimation of the population maximum

(b) Estimation of the population minimum

Figure 5: Exceedances over the threshold

4. A statistical approach to kernel partitioning of streaming applications

We estimate the minimal cost of kernel partitions (that lead to optimal performance) using Extreme Value Theory (EVT). EVT is a branch of statistics that studies extreme deviations from the median [5, 12]. One of the approaches in EVT is the *Peak Over Threshold* (POT) method. In its original form, the POT method takes into account only the distribution of the observations that exceed a given (high) threshold to estimate the population maximum [4, 35]. For example, in Figure 5(a), the observations x_1, x_4, x_5, and x_7 exceed the threshold and constitute extreme values, which can be used by POT analysis.

The POT method can also be used to estimate the population minimum [21, 28]. Estimation of the minimum requires the following five steps explained in detail in the next section:

- Obtain *i.i.d.* observations x_i of the cost of kernel partitions.
- Invert the sign of the observations: $x'_i = -x_i$.
- Determine the threshold u, shown in Figure 5(b).
- Use the values x'_i over the threshold u to estimate the maximum cost of the inverse population ($Max(Cost_{Inv})$).
- The minimum cost of the original population ($Min(Cost)$) corresponds to the negative value of the maximum of the inverse population: $Min(Cost) = -Max(Cost_{Inv})$;

The POT method can also be explained using cumulative distribution functions (CDF). For example, assume that F is the CDF of a random variable X. The POT method can be used to estimate the cumulative distribution function F_u of values of x above a certain threshold u. The function F_u is called the *conditional excess distribution function* and it is defined as

$$F_u(y) = P(X - u \leq y \mid X > u), \quad 0 \leq y \leq x_F - u,$$

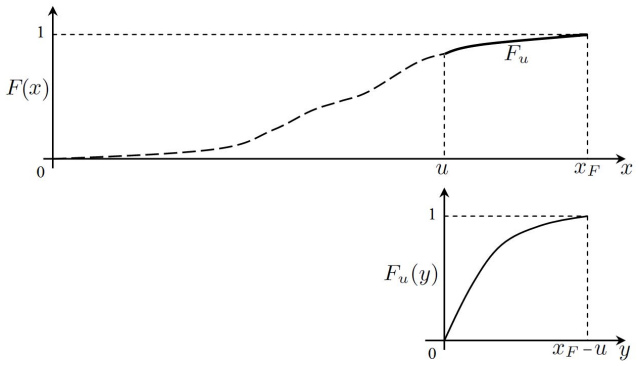


Figure 6: Cumulative distribution function $F(x)$ and matching conditional excess distribution function $F_u(y)$

where X is the observed random variable, u is the given threshold, $y = x - u$ are the exceedances over the threshold, and $x_F \leq \infty$ is the right endpoint of the cumulative distribution function F. Figure 6 shows a CDF of a random variable X (upper chart) and the corresponding conditional excess distribution function $F_u(y)$ (bottom chart).

The POT method is based on the following theorem [4, 35]:

Theorem 1 *For a large class of underlying distribution functions F, the conditional excess distribution function $F_u(y)$, for large threshold u, is well approximated by $F_u(y) \approx G_{\xi,\sigma}(y)$ where*

$$G_{\xi,\sigma}(y) = \begin{cases} 1 - (1 + \frac{\xi}{\sigma} y)^{-1/\xi} & for \quad \xi \neq 0 \\ 1 - e^{-y/\sigma} & for \quad \xi = 0 \end{cases}$$

for $y \in [0, (x_F - u)]$ if $\xi \geq 0$ and $y \in [0, -\frac{\sigma}{\xi}]$ if $\xi < 0$, where $G_{\xi,\sigma}$ is called Generalized Pareto Distribution (GPD).

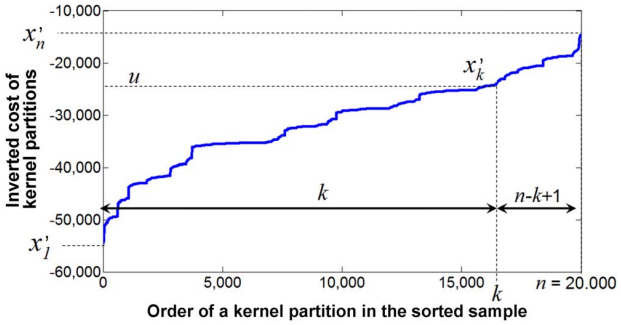
This means that for numerous distributions that present real-life problems, F_u can be approximated with a GPD. For each particular problem, the decision of whether GPD can be used to model the problem or not, is based on how well the sample of observations can be fitted to GPD. We describe the goodness of fit of observations to GPD in Steps 3 and 4 of Section 4.1. GPD is defined with two parameters: shape parameter ξ and scaling parameter σ. One of the characteristics of GPD is that for $\xi < 0$ the upper bound of the observed value equals $u - \frac{\sigma}{\xi}$, where σ and ξ are the GPD parameters and u is the selected threshold [21, 28].

In Theorem 1, the definition of $G_{\xi,\sigma}(y)$ for $\xi = 0$ can only be used to model problems with an infinite upper bound [21, 28]. In this study we use the GPD to estimate the minimal cost of kernel partitions for streaming applications. The value of this cost is always finite, and the estimated values of the parameter ξ are always $\hat{\xi} < 0$. Therefore, for the sake of simplicity of the presented mathematical formulas, in the rest of the paper we do not present $G_{\xi,\sigma}(y)$ formulas for $\xi = 0$.

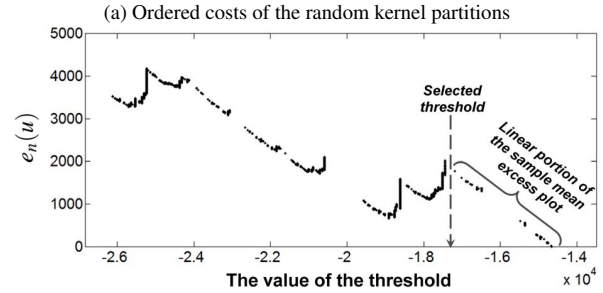
4.1. Application of Peak Over Threshold method

We use the POT method to estimate the minimal cost of kernel partitions for streaming benchmarks based on the cost of a sample of random partitions. Application of the POT method involves the following six steps:

Step 1: Generate the sample of random kernel partitions, and determine the cost of each partition in the sample (x_i). A requisite of the presented statistical analysis is that the selected kernel partitions must be independent and identically distributed (*i.i.d.*). The proposed methods to generate *i.i.d.* kernel partitions are described in Section 3. All of them passed the described *i.i.d.* tests.



(a) Ordered costs of the random kernel partitions



(b) Sample mean excess plot

Figure 7: Selection of the threshold for *mpeg2-subset*

Step 2: Invert the sign of the values of the observed costs. Originally, the POT method was used to estimate the maximum of a population based on a set of random *i.i.d.* observations. In order to estimate the *minimum* cost of kernel partitions, we invert the sign of the observed values ($x'_i = -x_i$) and estimate the *maximum* of the *inverse* population.

Step 3: Select the threshold u. The selection of the threshold u is an important step in POT analysis. Gilli and Këllezi [21, 28] propose using the *sample mean excess plot*, a graphical tool for threshold selection. This method first sorts all task assignments in the sample in non-decreasing cost order: $x'_1 \leq x'_2 \leq \cdots \leq x'_n$. Figure 7(a) shows the sorted cost of 20,000 uniformly distributed random kernel partitions of *mpeg2-subset* benchmark (see Section 3.4).

Then, the possible threshold u takes the values from x'_1 to x'_n ($x'_1 \leq u \leq x'_n$), and for each value we compute the sample mean excess function $e_n(u)$:

$e_n(u) = \frac{\sum_{i=k}^{n}(x'_i - u)}{n-k+1}$, where $k = min\{ \ i \ | \ x'_i > u\}$.

In this formula, the factor $n-k+1$ is the number of observations that exceed the threshold. Finally, the sample mean excess plot is defined by the points $(u, e_n(u))$ for $x'_1 \leq u \leq x'_n$. Figure 7(b) shows the example of the sample mean excess plot for the *mpeg2-subset* benchmark.

As commented before, the estimated parameter ξ of GPD must be negative ($\hat{\xi} < 0$) to obtain the upper bound of the $Max(Cost_{Inv})$. A characteristic of the GPD with parameter $\xi < 0$ is that it has a linear mean excess function plot. In order to have a good fit of the conditional distribution function F_u to GPD, the threshold should be selected so that the observations that exceed the threshold have a roughly linear sample mean excess plot. As an example, for the data presented in Figure 7, the threshold should be selected

to be $u = -17,500$. The sample mean excess plot is also a good tool to test whether GPD can be used to model a particular set of observations. If the right portion of the mean excess plot for the sample of measured task assignments performance is not roughly linear, that particular problem cannot be modeled using GPD.

Another important tool that can be used to understand if a given sample of observations can be modeled with a GPD is a *quantile plot* [5, 28]. In a quantile plot, the sample quantiles x'_i are plotted against the quantiles of a target distribution $F^{-1}(q_i)$ for $i = 1,\ldots,n$. If the sample data originates from the family of distributions F, the plot is close to a straight line.

The linear sample mean excess plot and the quantile plot are not the only constraints that should be considered when selecting the threshold. If the threshold is too low, the estimated parameters of GPD may be biased to the median values of the cumulative distribution function instead of to the maximum values. In order to avoid this bias, when selecting a threshold we have to ensure that the number of observations that exceed the selected threshold is not higher than 5% of the task assignments in the whole sample. This is a commonly used limit in studies that use POT analysis [21, 28, 36].

Step 4: Fit the GPD function to the observations that exceed the threshold and estimate parameters ξ and σ. Once the threshold u is selected, the observations over the threshold can be fitted to GPD, and the parameters of the distribution can be estimated. For the sake of simplicity, we assume that observations from x'_k to x'_n in the sorted sample presented in Figure 7(a) exceed the threshold. We rename the exceedances $y_{i-k+1} = x'_i - u$ for $k \le i \le n$ and use the set of elements $\{y_1, y_2, ..., y_m\}$ to estimate the parameters of GPD. The number of elements in the set, $m = n-k+1$, is the number of exceedances over the threshold.

Different methods can be used to estimate the parameters of GPD from a sample of observations [11, 24, 26, 38]. In our study, we used estimation based on the *likelihood* function [3]. The GPD has parameters ξ and σ. The likelihood that a set of observations $Y = \{y_1, y_2, ..., y_m\}$ is the outcome of a GPD with parameters $\xi = \xi_0$ and $\sigma = \sigma_0$ is defined to be the probability that GPD with parameters ξ_0 and σ_0 has outcome Y.

We make use of the likelihood function to compute the probability of different values of GPD parameters for a given set of observations $\{y_1, y_2, ..., y_m\}$. As the logarithm is a monotonically increasing function, the logarithm of a positive function achieves the maximum value at the same point as the function itself. This means that instead of finding the maximum of a likelihood function, we can determine the maximum of the logarithm of the likelihood function, the *log-likelihood* function. In statistics, log-likelihood is frequently used instead of the likelihood function because it simplifies computations. The estimation of parameters ξ and σ of $G_{\xi,\sigma}(y)$ involves the following steps:

(i) Determine the corresponding probability density function as a partial derivative of $G_{\xi,\sigma}(y)$ with respect to y:

$$g_{\xi,\sigma}(y) = \frac{\partial G_{\xi,\sigma}(y)}{\partial y} = \frac{1}{\sigma}(1+\frac{\xi}{\sigma}y)^{-\frac{1}{\xi}-1}$$

(ii) Find the logarithm of $g_{\xi,\sigma}(y)$:

$$log\ (g_{\xi,\sigma}(y)) = -log\ \sigma - (\frac{1}{\xi}+1)\ log(1+\frac{\xi}{\sigma}y)$$

(iii) Compute the log-likelihood function $L(\xi,\sigma|y)$ for the GPD as the logarithm of the joint density of the observations $\{y_1, y_2, ..., y_m\}$:

$$L(\xi,\sigma|y) = \sum_{i=1}^{m} log\ g_{\xi,\sigma}(y_i)$$

$$L(\xi,\sigma|y) = -m\ log\ \sigma - (\tfrac{1}{\xi}+1)\sum_{i=1}^{m} log(1+\frac{\xi}{\sigma}y_i)$$

We compute estimated values of parameters $\hat{\xi}$ and $\hat{\sigma}$, to *maximize* the value of the log-likelihood function $L(\xi,\sigma|y)$ for observations $\{y_1, y_2, ..., y_m\}$:

$$L(\hat{\xi},\hat{\sigma}|y) = \max_{\xi,\sigma}(L(\xi,\sigma|y))$$

$$L(\hat{\xi},\hat{\sigma}|y) = \max(-m\ log\ \sigma - (\frac{1}{\xi}+1)\sum_{i=1}^{m} log(1+\frac{\xi}{\sigma}y_i))$$

In order to determine the parameters $\hat{\xi}$ and $\hat{\sigma}$, we find the minimum of the negative log-likelihood function, $\min_{\xi,\sigma}(-L(\xi,\sigma|y))$, using the procedure *fminsearch()* included in Matlab® R2007a [13]. The values $\hat{\xi}$ and $\hat{\sigma}$ are called the point estimates of the parameters ξ and σ, respectively.

Step 5: Estimate the maximum of the inversed costs. The maximum of the inverse cost can be determined only for $\hat{\xi} < 0$ which is satisfied for all data sets that are presented in this paper. The point estimate of $Max(Cost_{Inv})$ is computed as $\widehat{Max(Cost_{Inv})} = u - \hat{\sigma}/\hat{\xi}$.

In order to indicate the confidence of the estimate, we compute the confidence intervals of the estimated $\widehat{Max(Cost_{Inv})}$. The confidence intervals is computed using the *likelihood ratio test* [3], which consists of the following steps:

(i) Define GPD as a function of ξ and $Max(Cost_{Inv})$:

$$G_{\xi,Max(Cost_{Inv})}(y) = 1 - (1 - \frac{1}{Max(Cost_{Inv})-u}y)^{-1/\xi}$$

(ii) Determine the corresponding probability density function:

$$g_{\xi,Max(Cost_{Inv})}(y) = \frac{\partial G_{\xi,Max(Cost_{Inv})}(y)}{\partial y} = \\ = -\frac{1}{\xi(Max(Cost_{Inv})-u)}(1-\frac{1}{Max(Cost_{Inv})-u}y)^{-\frac{1}{\xi}-1}$$

(iii) Compute the joint log-likelihood function for observations $\{y_1, ..., y_m\}$:

$$L(\xi,Max(Cost_{Inv})|y) = \sum_{i=1}^{m} log\ g_{\xi,Max(Cost_{Inv})}(y_i)$$

$$L(\xi,Max(Cost_{Inv})|y) = -n\ log(-\xi(Max(Cost_{Inv})-u)) - \\ -(1+\tfrac{1}{\xi})\sum_{i=1}^{n} log(1-\frac{1}{Max(Cost_{Inv})-u}y_i)$$

(iv) Find the $Max(Cost_{Inv})$ confidence interval. We determine the confidence interval for $Max(Cost_{Inv})$ using the likelihood ratio test [3] and Wilks's theorem [14, 41, 42]. The maximum log-likelihood function is determined as:

$$L(\hat{\xi},\widehat{Max(Cost_{Inv})}|y) = \max_{\xi,Max(Cost_{Inv})}(L(\xi,Max(Cost_{Inv}))).$$

The function $L(\hat{\xi},\widehat{Max(Cost_{Inv})}|y)$ has two parameters that are free to vary (ξ and $Max(Cost_{Inv})$), hence it has two degrees of freedom ($df_1 = 2$). As $Max(Cost_{Inv})$ is our parameter of interest, the profile log-likelihood function is defined as:

$$L^*(Max(Cost_{Inv})) = \max_{\xi} L(\xi,Max(Cost_{Inv})).$$

The function $L^*(Max(Cost_{Inv}))$ has one parameter that is free to vary, *i.e.* one degree of freedom ($df_2 = 1$). Wilks's theorem applied to the problem that we are addressing claims that, for a large number of exceedances over the threshold, the distribution of $2(L(\hat{\xi},\widehat{Max(Cost_{Inv})}) - L^*(Max(Cost_{Inv})))$ converges to a χ^2 distribution with $df_1 - df_2$ degrees of freedom. Therefore, the confidence interval of $Max(Cost_{Inv})$ includes all values of $Max(Cost_{Inv})$ that satisfy the following condition:

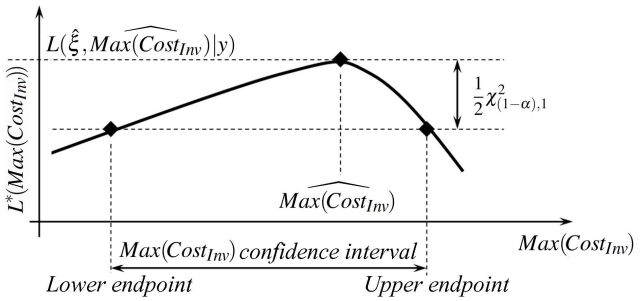


Figure 8: $Max(Cost_{Inv})$ confidence interval

$$L(\hat{\xi}, \widehat{Max(Cost_{Inv})}) - L^*(Max(Cost_{Inv})) < \frac{1}{2}\chi^2_{(1-\alpha),1} \quad (1)$$

$\chi^2_{(1-\alpha),1}$ is the $(1-\alpha)$-level quantile of the χ^2 distribution with one degree of freedom ($df_1 - df_2 = 1$). α is the confidence level for which we compute $Max(Cost_{Inv})$ confidence intervals. We illustrate the computation of the $Max(Cost_{Inv})$ confidence interval in Figure 8. The figure plots $L^*(Max(Cost_{Inv}))$ for different values of $Max(Cost_{Inv})$. For $Max(Cost_{Inv}) = \widehat{Max(Cost_{Inv})}$, L^* reaches its maximum. The confidence interval of $Max(Cost_{Inv})$ includes all values of $Max(Cost_{Inv})$ that satisfy the condition $L^*(Max(Cost_{Inv})) > L(\hat{\xi}, \widehat{Max(Cost_{Inv})}) - \frac{1}{2}\chi^2_{(1-\alpha),1}$, which corresponds to Equation 1. We computed the $Max(Cost_{Inv})$ confidence interval using an iterative method based on the *fminsearch()* function included in Matlab® R2007a.

Step 6: Estimate the minimum cost of the kernel partitions. The minimum cost of the kernel partitions corresponds to the estimated maximum of the inverse cost: $Min(Cost) = -Max(Cost_{Inv})$. Also, the lower and upper endpoints of the $Min(Cost)$ confidence interval correspond to the inverse upper and lower endpoints of the $Max(Cost_{Inv})$ confidence interval, respectively.

The code that performs the statistical *i.i.d.* test, generates the sample mean excess plots, infers the parameters of the GPD distribution, and estimates the minimum cost of kernel partitions was developed in Matlab® R2007a.

5. Results

In this section, we use the POT method to estimate, for each of the StreamIt 2.1.1 benchmarks, the cost of the optimal kernel partition. We compare the four sampling methods described in Section 3. We also evaluate the number of random kernel partitions that are required by the presented statistical approach. Finally, we analyze whether a good kernel partition would be found using random sampling on its own.

Before using any heuristics-based algorithm for the concrete application under study, the user should check whether exhaustive search would be impractical. In general, the number of valid kernel partitions is vast (e.g. 10^{20}). It is possible, however, that a particular benchmark has a small stream graph, so that exhaustive search would work. For example, the *dct* benchmark included in the StreamIt 2.1.1 suite contains only eight kernels, linked in a simple stream graph. The number of partitions of this benchmark, onto four threads, is just 32. In this case, exhaustive search is the simplest and fastest way to find the optimal kernel partition.

Table 1: Applicability of the POT method

Benchmark	Sampling method			
	DFS	EC	EC-F	UD
bitonic-sort	*NA*	✔	*NA*	✔
channelvocoder	✔	*NA*	*NA*	✔
des	*NA*	✔	✔	✔
fft	*NA*	✔	*NA*	✔
filterbank	*NA*	*NA*	*NA*	✔
fm	✔	*NA*	*NA*	✔
mpeg2-subset	*NA*	✔	*NA*	✔
radar	✔	*NA*	*NA*	✔
serpent_full	*NA*	✔	*NA*	✔
tde_pp	*NA*	✔	*NA*	✔
vocoder	*NA*	✔	*NA*	✔

5.1. Estimation of the minimal cost using the POT method

We apply our technique to estimate the performance of the optimal kernel partition on four threads, for each of the benchmarks in the StreamIt 2.1.1 suite. As discussed in the previous section, the *dct* benchmark can be solved using exhaustive search, leaving eleven benchmarks to be analyzed using the POT statistical method.

For each benchmark, we use the four sampling methods described in Section 3 to generate random samples, and use these samples as the input to the statistical analysis. Each sample contains 20,000 random kernel partitions. Table 1 shows whether or not the statistical method could produce an estimate of the optimal performance. Each row in the table corresponds to one of the benchmarks, and each column corresponds to a different sampling method. A tick sign (✔) means that the POT method did generate an estimate. An *NA* (Not Applicable) entry means that the statistical method failed to produce any estimate.

There are two reasons why the POT method is sometimes unable to produce an estimate. First, the lower bound of the estimated minimal cost may diverge to minus infinity. Second, the iterative method that determines the confidence bounds of the estimated minimal cost (see Step 5 in Section 4.1) may not converge to a solution. In all experiments in which the POT method was not applicable, the sample mean excess plot and the quantile plot strongly suggested that the POT method could not be applied to that dataset (see Step 3 in Section 4.1).

From the results in the table, we see that the POT method using the Depth First Search (DFS) sampling method was successfully applied for only three out of eleven benchmarks. The results for the Edge Contraction (EC) method are better, but still moderate: the POT method estimated the minimum cost for seven out of eleven benchmarks. The Edge Contraction with Filter (EC-F) method was an attempt to improve load-balance over EC. However, the POT analysis could now only be applied to one of the benchmarks. We compared the costs of the random kernel partitions sampled by the EC and EC-F methods, and confirmed that the EC-F method did indeed select kernel partitions with lower cost. This was, however, not sufficient to make the samples appropriate for POT analysis. In future work, we plan to analyze this phenomenon in detail. Finally, when the POT method was applied to the uniformly distributed random samples (UD column of the table), a minimum cost was generated for all eleven benchmarks under study.

From the results presented in Table 1, we conclude that the sampling method is an important step in the analysis. All presented sampling methods select *i.i.d.* samples and fulfill the requirements of the POT statistical analysis. However, only the uniformly distributed

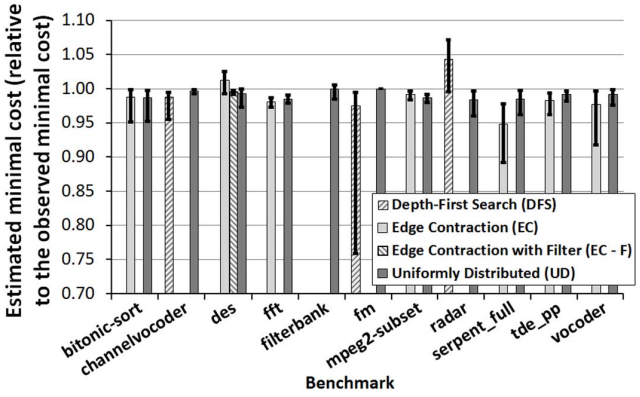


Figure 9: Estimated minimal cost

samples always led to an estimate of the cost of the optimal kernel partition. Other sampling method used to address *the same problem, for the same benchmarks, using the same statistical analysis* provided moderate (EC method) or low performance (DFS and EC-F methods).

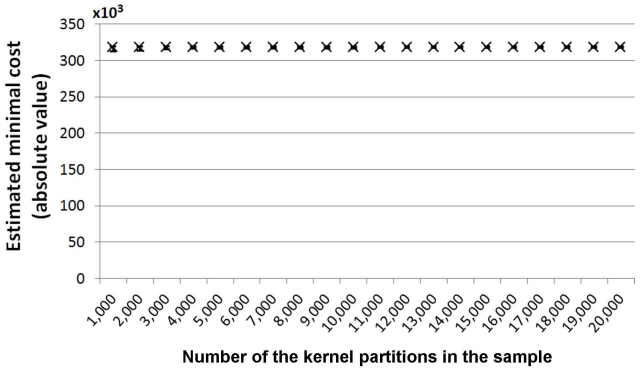
5.2. Precision of the estimation

The results that we use to analyze the precision of the estimated values are presented in Figure 9. The X-axis of the figure lists the benchmarks, while the Y-axis shows the estimated minimal cost, i.e. the estimated cost of the optimal kernel partition. The results are presented relative to the cost of the best kernel partition captured in 80,000 random kernel partitions from all four sampling methods. Kernel partitions were generated using four different sampling methods (DFS, EC, EC-F, and UD), and each method generated 20,000 random partitions.

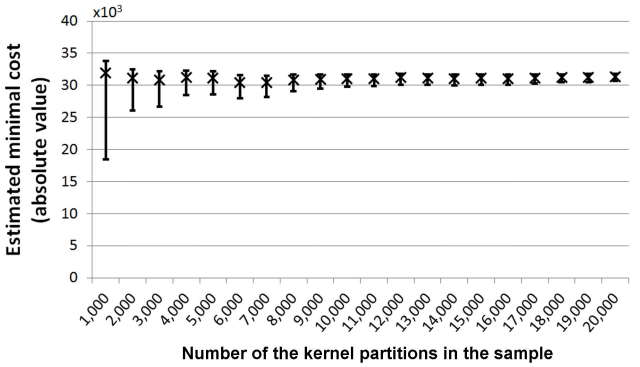
Different bars of the chart correspond to the different sampling methods used: DFS, EC, EC-F, and UD. If the POT method could not be used to estimate the minimal cost for a given benchmark and sampling method, the corresponding bar is not plotted. The height of the solid bars correspond to the point estimation of the minimal cost, while the error bars correspond to the confidence bounds for 0.95 confidence level.

High precision of the estimated minimal cost is indicated by tight confidence bounds. The width of the confidence bounds is below 10% for all bars except one (DFS sample for the *fm* benchmark). For 18 out of 22 cases, the width of the confidence bounds is below 5%.

Required number of random kernel partitions: UD method is the only sampling method that provided samples appropriate for the POT statistical analysis for all the benchmarks under study. Therefore, from this point on, we analyze only the samples that are generated with this method. In order to understand the impact of the sample size on the estimated minimal cost, we generated samples that contain between 1,000 and 20,000 random kernel partitions. For each sample, we used the POT method to estimate the minimal kernel cost. Intuitively, we expect that the POT method provides more precise estimation as the number of kernel partitions in the sample increases. In general, larger samples contain more kernel partitions in the tail that are fitted to the Generalized Pareto Distribution (GPD), and therefore the estimated GPD parameters and the minimal cost are more precise. Figure 10 shows the results for the *channelvocoder* and *serpent_full* benchmarks. In each figure, X-axis lists the number of random kernel partitions in the sample, while the Y-axis shows the



(a) channelvocoder benchmark (fastest convergence)



(b) serpent_full benchmark (slowest convergence)

Figure 10: The impact of the sample size on the estimation of the minimal cost (UD sampling method)

estimated minimal cost. The cross markers show the point estimation of the minimal cost, and the error bars correspond to the confidence bounds for the 0.95 confidence level.

For the *channelvocoder* benchmark, 1,000 random kernel partitions are sufficient to estimate the minimal cost with a high precision (see Figure 10(a)). We detect similar results for *fft, filterbank, fm, mpeg2-subset, tde-pp,* and *vocoder*. On the other hand, for the *serpent_full* benchmark, estimation based on 1,000 random kernel partitions has wide confidence bounds (see Figure 10(b)). Precise estimation of the minimal cost requires more than 8,000 random kernel partitions. The width of the confidence bounds reduces significantly as the sample increases from 1,000 to 8,000 kernel partitions. Further increment in the sample size only slightly improves the precision of the estimation. From the results shown in Figure 10(b), we also see that, as the sample size increases, the point estimation remains roughly the same and the confidence bounds converge to this value. Results for the benchmarks *bitonic-sort, des,* and *radar* follow the same trend.

Based on the presented analysis, we see that the sample size required for the precise estimation of the minimal cost significantly depends on the benchmark under study. If a user requires a minimal cost to be estimated with a given precision, we propose the following iterative method. The user can generate a small sample of random kernel partitions and estimate the minimal cost using the POT method. As long as the estimated cost does not fulfill the user's requirements, the user can increase the sample size and repeat the analysis.

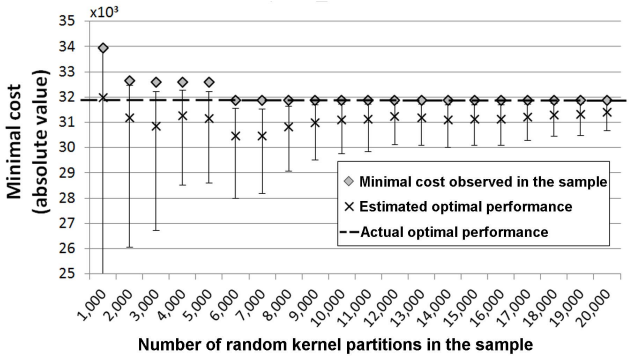


Figure 11: Comparison between the actual and the estimated kernel partition costs (serpent_full benchmark, UD sampling method)

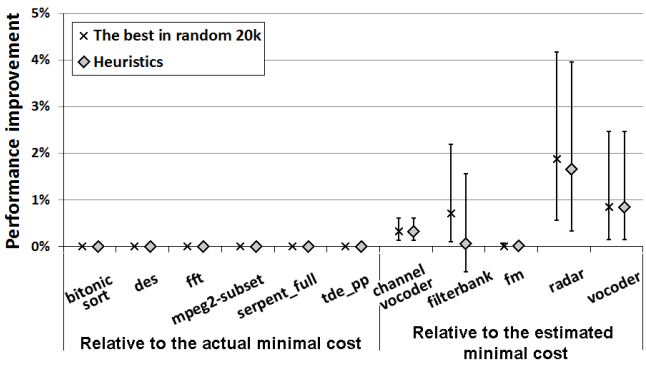


Figure 12: Comparison of random sampling (UD method) and heuristics-based algorithm

5.3. Accuracy of the estimation

In general, the kernel partitioning problem is an intractable problem. However, for *bitonic-sort, des, fft, mpeg2-subset, serpent_full,* and *tde_pp* benchmarks partitioned into exactly four software threads, brute force exploration is time consuming, but feasible. Therefore, we were able to determine the cost of all the kernel partitions, and to compare the actual and the estimated best kernel partition costs for these benchmarks. The results for the *serpent_full* benchmark are shown in Figure 11. We also analyze the estimation accuracy for different numbers of uniformly distributed random kernel partitions in the sample. The X-axis of the figure lists the size of the sample, while the Y-axis shows the absolute value of the kernel partition cost. The cross data markers show the point estimation of the minimal cost, and the error bars correspond to the confidence bounds for the 0.95 confidence level. The actual best kernel partition cost is marked with the horizontal dashed line. Finally, we also plot the minimal kernel cost observed in each random sample (diamond data markers).

First, we observe that the estimated best kernel partition cost (with confidence bounds included) is always lower than the minimal kernel cost detected in the corresponding random sample. Intuitively, this is because the statistical method estimates that the best kernel partition cost in the population (out of all possible partitions) is not higher than the minimal cost observed in the sample. We also detected that the upper confidence bound of the estimated best kernel partition cost asymptotically approaches the minimal kernel partition cost observed in the sample, as the confidence level of the estimation increases.

The estimation of the best kernel partition cost is accurate if its confidence bounds include the actual best (minimal) cost, which is satisfied for the samples that contain from 1,000 to 5,000 random kernel partitions in the Figure 11. For the samples that contain more than 6,000 kernel partitions, the presented statistical method slightly underestimates the minimal cost. This is because these samples capture a kernel partition with the best actual cost, as we explain in the previous paragraph. The underestimation is very low and it decreases with the number of kernel partitions in the sample, from 0.9% (6,000 kernel partitions) to 0.3% (20,000 partitions). The results for *bitonic-sort, des, fft, mpeg2-subset,* and *tde_pp* benchmarks follow the same trend.

Brute force exploration of the kernel partitioning problem for *channelvocoder, filterbank, fm, radar,* and *vocoder* benchmarks is infeasible. Therefore, for these benchmarks, we cannot determine the optimal kernel partition and its cost, and we cannot validate that the estimated values of the POT method were correct. However, from the results presented in Figure 9, we can detect that the estimation is incorrect if:

- The confidence bounds of different bars that correspond to the same benchmark do not overlap. This means that the POT method applied to different samples of the same benchmark estimated different minimal cost.
- The ratio between the estimated minimal cost (with confidence bounds included) and the minimal cost detected in random samples is higher than 1. This means that we detected a kernel partition with the cost that is lower than the estimated minimal cost.

From the results presented in Figure 9, we did not detect a single mispredicted cost of the optimal kernel partition.

5.4. Random sampling approach to a kernel partitioning

Our previous study [36] addresses the problem of process scheduling for modern multicore/multithreaded processors. The results presented in the study demonstrate that a random sample of several thousand random process schedules likely captures a schedule with a good performance. The study analyzes the probability that a uniformly distributed random sample of N observations contains at least one observation from the best-performing $P\%$ of the population (e.g. the best 1% of the population). This probability can be computed using the following formula: $Prob = 1 - \left(\frac{100\text{-}P}{100}\right)^N$. As P is a small positive number, the value of the fraction $\frac{100\text{-}P}{100}$ is always between 0 and 1. Therefore, for large N, the factor $\left(\frac{100\text{-}P}{100}\right)^N$ converges to 0, and the observed probability converges to 1. For example, the probability that a uniformly distributed random sample of 1,000 observations contains at least one element from the best 1% of the population exceeds 99.99%.

In order to analyze whether random sampling can be used to select a good kernel partition, we compare the cost provided by the fairly-complex heuristics-based kernel partitioning algorithm proposed by Carpenter et al. [9] with the minimal cost observed in the random sample. The sample was comprised of 20,000 kernel partitions generated using the UD sampling method. The results are shown in Figure 12. The X-axis of the figure lists different benchmarks, while the Y-axis shows the possible performance improvement of the kernel partitioning approaches. For *bitonic-sort, des, fft, mpeg2-subset, serpent_full,* and *tde_pp* benchmarks, brute force exploration is feasible, so we compare the kernel partition costs provided by the random sampling and heuristics-based algorithm with the actual minimal costs. For the remaining five benchmarks, *channelvocoder, filterbank, fm, radar,*

and *vocoder*, the results are plotted relative to the estimated minimal costs. The minimal cost is estimated using the POT method on 20,000 uniformly distributed random kernel partitions. The markers correspond to the point estimation of the minimal cost, while the error bars correspond to the estimated confidence bounds for a 0.95 confidence level.

For *bitonic-sort, des, fft, mpeg2-subset, serpent_full,* and *tde_pp* benchmarks, the best costs observed by the random sampling and the heuristics-based algorithm match the actual best costs of kernel partitions. For the *channelvocoder, filterbank, fm, radar,* and *vocoder* benchmarks, random sampling and heuristics-based algorithm, detect kernel partitions with a cost that is close to the estimated optimal one. For four out of five benchmarks (all except *radar*), the possible improvement of both approaches is below 3% (confidence bounds included). For *radar* benchmark, the estimated improvement ranges up to 4% and 4.2% for random sampling and heuristics-based algorithm, respectively. If we consider the point estimation, the estimated performance improvement is below 2% for all the benchmarks, and below 1% for four out of five benchmarks. For *channelvocoder, fm,* and *vocoder* benchmarks, the performance of the best kernel partitions in the random sample match the performance of the heuristics-based algorithm. For *filterbank* and *radar* benchmarks, the heuristics-based algorithm selected kernel partitions with 0.6% and 0.2% lower cost, respectively, which is a negligible difference. If a good heuristics-based approach is available for the applications, hardware, and metric under study, the user can choose whether to use the heuristics or the random sampling. However, it is common that heuristics-based approaches are not directly applicable to the exact situation under study. It is often difficult and time-consuming to adapt a heuristic to a particular target scenario. On the other hand, the random sampling approach is simple and easy to apply.

Required number of random kernel partitions: The formula $Prob = 1 - \left(\frac{100\text{-}P}{100}\right)^N$ can be used to compute the probability that an uniformly distributed random sample of N observations captures at least one out of P% of the kernel partitions with the lowest cost. However, as we do not know the difference in the cost in the best P% of all kernel partitions, the formula cannot be used to compute the difference between the minimal cost captured in a random sample and the actual optimal cost.

In order to analyze whether a sample of N randomly selected kernel partitions captures a good partition, we observe the minimal cost detected in the random sample and compare it with the minimal cost determined by the statistical estimation or brute force exploration, when feasible. The random samples are generated with the UD sampling method. Figure 13 shows the results of the experiments for *serpent_full* benchmark. We repeat the experiment for different sample sizes that are listed along the X-axis of the figure. Dashed vertical lines separate the results for tens (from 10 to 90), hundreds (from 100 to 900), and thousands (from 1,000 to 20,000) of random kernel partitions in the sample. The Y-axis shows the relative difference between the minimal cost captured in the random sample and the actual minimal cost determined by brute force exploration. In order to present statistically significant results, for each sample size, we randomly generate the sample 100 times and report the mean (cross marker) and the standard deviation (error bars) of the minimal cost detected in different runs.

We see that tens of random kernel partitions in the sample are unlikely to capture a good partition. We also detect a high standard deviation, which means that the cost of the best-captured kernel

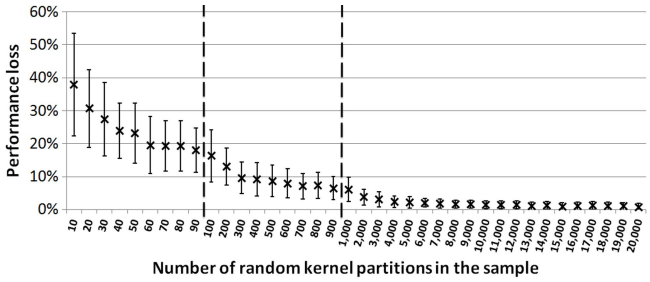


Figure 13: The impact of the sample size on the performance of the random sampling (serpent_full benchmark, UD sampling method)

partition is significantly different for different samples of the same size. For hundreds of kernel partitions in the sample, the best captured cost slowly converges to the estimated optimal one. The standard deviation decreases, which means that different samples of the same size provide similar performance. Finally, several thousand random kernel partitions capture a cost that is very close to the optimal one. Also, the detected standard deviation is low (1-2%).

We detect the same trend for all eleven benchmarks under study. Therefore, we conclude that uniformly distributed random sampling can be used to find a good kernel partition. However, we recommend this method only when the random sample contains at least several thousand kernel partitions.

In order to determine the sample size that captures a good kernel partition, a user could also observe the convergence rate of the best observed kernel partition performance in the sample as the sample increases. For example, from the Figure 13, we could observe that increasing the random sample over several thousand kernel partitions insignificantly improves the observed performance, and stop the random sampling at that point without estimation of the optimal performance. Although this approach may provide good results, it is not clear how it would avoid convergence to a local minimum of the population (e.g. see the results for 60, 70, and 80 kernel partitions in Figure 13). Also, without estimated optimal performance, we cannot determine the quality of the delivered kernel partition, i.e. we cannot provide the confidence bounds of the estimated performance improvement.

5.5. Other considerations

There are several additional aspects to consider regarding the presented statistical approach.

Experimentation time: The presented approach requires thousands of random kernel partitions to be generated and evaluated. The time to generate and evaluate 1,000 uniformly distributed random kernel partitions, for one of the StreamIt 2.1.1 benchmarks, was on average 28 minutes, using a single core of an Intel Xeon E5649 processor at 2.5GHz with 4GB memory. Running the POT method takes less than a minute. The program that generates the uniformly distributed samples is not optimized and is implemented in the Python programming language. An implementation in *C* would be much faster. Also, since different kernel partitions can be generated independently, the time to generate the sample would decrease linearly with the number of cores. The experimentation time is acceptable considering that the selected kernel partition will be compiled to the executable that can be used on numerous systems based on the same hardware during the lifetime of the system.

Scalability: The number of cores and the number of hardware threads increase in each processor generation [34]. In order to optimally use future multicore processors, kernel partitioning algorithms will have to generate a significantly larger number of threads. It is important, therefore, to analyze how these algorithms scale with the number of software threads. On the other hand, at the application level, it is reasonable to expect that the complexity of streaming applications increases leading to more complex stream graphs that comprise a larger number of kernels.

The statistical analysis that estimates the optimal kernel cost is based on the values of the performance metric, so its cost is independent of the number of threads and the complexity of the stream graph. The cost of the sampling method that generates the uniformly distributed random kernel partitions scales linearly with the number of kernels in the stream graph and with the number of output software threads. It also scales linearly with the mixing time of the partition graph [30].

Compiler optimizations and system constraints: When the program is described using a stream language, the compiler may perform complex optimizations over the stream graph; it can combine adjacent filters, split computationally intensive filters into multiple parts, or duplicate filters to have more parallel computation [8]. In order to find the set of optimizations that provides the best performance, it is important to determine good kernel partitions for different optimization sets. In this case, the proposed kernel partitioning approach does not change, but the random sampling and the presented statistical analysis are simply repeated for different optimization sets, i.e. for different stream graphs of the same program. Kernel partitioning that satisfies different system constraints, such as optimizing performance subject to memory limits, is an interesting avenue for future work.

Evaluation on real hardware: In this paper, the presented statistical approach was evaluated based on the estimates of the kernel partitions' costs provided by the StreamIt compiler. The approach was not evaluated on real hardware because of limitations in the experimental environment. The back-end of the StreamIt compiler, that we used in the study, was not capable of generating working code for different user-defined kernel partitions. As a part of future work, we plan to evaluate the presented statistical approach on real hardware. In order to do so, we intend to modify the StreamIt compiler, so it can generate the executables that correspond to any given kernel partition.

6. Related work

Several projects and studies propose different tools for compiling of streaming-like applications and their mapping onto multicore architectures.

StreamIt is a project with publicly available compiler and benchmark suite [1] . The StreamIt source language imposes a structure on the stream program graph to the compiler. The StreamIt compiler performs fully automated load balancing, communication scheduling, routing, and a set of cache optimizations [22, 23, 37]. The StreamIt compiler targets the Raw Microprocessor [40], symmetric multicore architectures, and clusters of workstations.

The Stream Graph Modulo Scheduling (SGMS) algorithm is part of StreamRoller [29], a StreamIt compiler for the Cell Architecture. This algorithm splits stateless kernels, partitions the graph, and statically schedules the software threads onto the Cell architecture. The splitting and partitioning problem is translated into an integer linear programming problem, which is solved using CPLEX Optimization Studio [27], an software package for mathematical programming.

Gedae Graph Language [33] is a proprietary GUI tool that supports the hierarchical development of data flow graphs. Gedae allows the user to specify different graph partitions and automatically maintains the data flow and connectivity of the graph. However, all the graph partition is done under user control, not by the compiler.

The Ptolemy II software environment [17] is designed to model heterogeneous embedded computing systems. Ptolemy views computing systems as a set of basic processing blocks (*actors*) that are connected using explicitly-defined communication channels. This view is very similar to the state-of-the-art interpretation of streaming-like applications. Related work from the Ptolemy project explores the more theoretical aspects of partitioning and scheduling data flow graphs for multiprocessors [25].

Liao et al. [31] present a parallel compiler for the Brook streaming language [7] with aggressive data and computation transformations. The compiler models each streaming kernel as an implicit loop nest over stream elements and uses affine partitioning to map regular programs onto multicore processors.

Farhad et al. [18] show that state-of-the-art linear programming approaches are impractical for transformations of large stream graphs to be executed on a large number of processor cores. The authors also propose an approximation algorithm for deploying stream graphs on multicore processors.

Our study shows a different approach to the kernel partitioning problem. Instead of using complex heuristics-based algorithms, we address the problem using random sampling and statistical inference. We present a statistical method that estimates performance of the optimal kernel partition based on measured performance of a sample of random partitions. We also demonstrate that random sampling can be used to find a kernel partition with performance close to the optimal one.

In our previous study [36], we use random sampling and statistical inference to analyze the optimal assignment of existing software threads onto different processor cores. There are two main contributions of this article, beyond our previous work: (1) In this article, we apply EVT to a different domain. Kernel partitioning and thread assignment are fundamentally different problems. In its essence, kernel partitioning is a graph partitioning problem, and the thread assignment problem addressed in our previous study is a multiprocessor scheduling problem [20]. (2) In this article, we also show that the sampling method has a significant effect on the applicability of the statistical method. We analyze different sampling methods, and our results strongly recommend that the samples should be uniformly distributed.

7. Conclusions

One of the greatest difficulties in using modern computing systems is how to write efficient, portable, correct software for multicore processors. A promising approach is to expose more parallelism to the compiler through domain-specific languages, enabling the compiler to perform complex high-level transformations. An important application domain comprises stream programs. A prominent step in compiling a stream program to multiple processors is kernel partitioning, which significantly affects application performance. Finding an optimal kernel partition is, however, an intractable problem.

In this paper, we proposed a statistical approach to the kernel partitioning problem. We described a method that statistically estimates, with a given confidence level, the performance of the optimal kernel partition. Knowing the optimal performance improves the evaluation of any kernel partitioning algorithm, and it is the most important piece of information for the system designer when

deciding whether an existing algorithm should be enhanced. We demonstrated that the sampling method is an important part of the analysis, and that not all methods that generate $i.i.d.$ samples provide good results. We also showed that random sampling on its own can be used to find a good kernel partition, and that it could be an alternative to heuristics-based approaches.

The presented statistical method does not depend on the application. It does not require any application profiling nor does it require the understanding of the application stream graph. The method can be applied to streaming applications with any number of kernels, and it can target any number of software threads. The presented method can analyze different metrics such as throughput, maximum hardware utilization, and minimum energy or power consumption.

We successfully applied the presented statistical analysis to the benchmarks included in the StreamIt 2.1.1 suite. The method precisely estimated the optimal kernel partition performance for all the benchmarks under study. Also, in our experiments, several hundred or several thousand random kernel partitions were enough to find a partition with close to optimal performance. The performance of the kernel partitions that were selected using random sampling matched the performance provided by the complex heuristic-based approach.

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under the contract TIN-2007-60625, and by the European HiPEAC-3 Network of Excellence. Also, this work has been partially supported by the Department of Universities, Research and Information Society (DURSI) of the Catalan Government (grant 2010-BE-00352). Petar Radojković holds the FPU grant (Programa Nacional de Formación de Profesorado Universitario) under contract AP2008-02370, of the Ministry of Education of Spain. Miquel Moretó is supported by an MEC/Fulbright Fellowship. The authors wish to thank to Liliana Cucu-Grosjean and Luca Santinelli from INRIA, and Jaume Abella from Barcelona Supercomputing Center for their technical support.

References

[1] "StreamIt project," http://groups.csail.mit.edu/cag/streamit/.

[2] ACOTES, "IST ACOTES Project Deliverable D2.2 Report on Streaming Programming Model and Abstract Streaming Machine Description Final Version," 2008.

[3] A. Azzalini, *Statistical Inference Based on the Likelihood*. Chapman and Hall, 1996.

[4] A. A. Balkema and L. de Haan, "Residual life time at great age," *Annals of Probability*, vol. 2, 1974.

[5] J. Beirlant *et al.*, *Statistics of Extremes: Theory and Applications*. John Wiley and Sons, Ltd, 2004.

[6] J. V. Bradley, *Distribution-Free Statistical Tests*. Prentice-Hall, 1968.

[7] I. Buck, "Brook Spec v0.2," 2003.

[8] CAG MIT, *StreamIt Language Specification, Version 2.1*, 2006.

[9] P. M. Carpenter, A. Ramirez, and E. Ayguade, "Mapping stream programs onto heterogeneous multiprocessor systems," in *Proceedings of the international conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2009.

[10] P. M. Carpenter *et al.*, "A streaming machine description and programming model," *Proceedings of the International Symposium on Systems, Architectures, Modeling and Simulation*, 2007.

[11] E. Castillo and A. Hadi, "Fitting the Generalized Pareto Distribution to data," *Journal of the American Statistical Association*, vol. 92, 1997.

[12] E. Castillo, *Extreme value theory in engineering*. Academic Press, Inc., 1988.

[13] S. J. Chapman, *Essentials of MATLAB Programming*. Cengage Learning, 2009.

[14] H. Chernoff, "On the distribution of the likelihood ratio," *Annals of Mathematical Statistics*, vol. 25, 1954.

[15] W. G. Cochran, *Sampling Techniques, 3rd edition*. Wiley-India, 2007.

[16] L. Cucu-Grosjean *et al.*, "Measurement-based probabilistic timing analysis for multi-path programs," in *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, 2012.

[17] J. Eker *et al.*, "Taming heterogeneity–the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, 2003.

[18] S. M. Farhad *et al.*, "Orchestration by approximation: mapping stream programs onto multicore architectures," in *Proceedings of the sixteenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[19] W. Feller, *An introduction to Probability Theory and Its Applications*. John Wiley & Sons, Inc., 1971.

[20] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[21] M. Gilli and E. Këllezi, "An application of extreme value theory for measuring financial risk," *Computational Economics*, vol. 27, 2006.

[22] M. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[23] M. Gordon *et al.*, "A Stream Compiler for Communication-Exposed Architectures," in *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[24] S. Grimshaw, "Computing the maximum likelihood estimates for the Generalized Pareto Distribution to data," *Technometrics*, vol. 35, 1993.

[25] S. Ha and E. Lee, "Compile-time scheduling and assignment of data-flow program graphs with data-dependent iteration," *IEEE Transactions on Computers*, vol. 40, no. 11, 1991.

[26] J. R. M. Hosking and J. R. Wallis, "Parameter and quantile estimation for the generalised pareto distribution," *Technometrics*, vol. 29, 1987.

[27] ILOG, "CPLEX Math Programming Engine," http://www.ilog.com/products/cplex/.

[28] E. Këllezi and M. Gilli, "Extreme value theory for tail-related risk measures," International Center for Financial Asset Management and Engineering, FAME Research Paper Series, 2000.

[29] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design & Impl.*, 2008.

[30] D. Levin, Y. Peres, and E. Wilmer, *Markov chains and mixing times*. American Mathematical Society, 2009.

[31] S. Liao *et al.*, "Data and Computation Transformations for Brook Streaming Applications on Multiprocessors," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[32] L. Lovász, "Random walks on graphs: A survey," *Combinatorics, Paul Erdos is Eighty*, vol. 2, no. 1, 1993.

[33] W. Lundgren, K. Barnes, and J. Steed, "Gedae: Auto Coding to a Virtual Machine," in *8th High Performance Embedded Computing Workshop*, 2004.

[34] K. Olukotun and L. Hammond, "The future of microprocessors," *Queue*, vol. 3, no. 7, Sep. 2005.

[35] J. I. Pickands, "Statistical inference using extreme value order statistics," *Annals of Statististics*, vol. 3, 1975.

[36] P. Radojković *et al.*, "Optimal task assignment in multithreaded processors: A statistical approach," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[37] J. Sermulins *et al.*, "Cache aware optimization of stream programs," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.

[38] N. Tajvidi, "Design and implementation of statistical computations for Generalized Pareto Distributions," *Technical Report, Chalmers University of Technology*, 1996.

[39] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," *International Conference on Compiler Construction*, vol. 4, 2002.

[40] E. Waingold *et al.*, "Baring It All to Software: Raw Machines," *Computer*, 1997.

[41] S. S. Wilks, "The large-sample distribution of the likelihood ratio for testing composite hypotheses," *Annals of Mathematical Statistics*, vol. 9, 1938.

[42] S. S. Wilks, *Mathematical Statistics*. Princeton University, 1943.