Offline Library Adaptation Using Automatically Generated Heuristics

Frédéric de Mesmay, Yevgen Voronenko, and Markus Püschel
Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, USA
{fdemesma, yvoronen, pueschel}@ece.cmu.com

***Abstract*—Automatic tuning has emerged as a solution to provide high-performance libraries for fast changing, increasingly complex computer architectures. We distinguish *offline* adaptation (e.g., in ATLAS) that is performed during installation without the full problem description from *online* adaptation (e.g., in FFTW) that is performed at runtime. Offline adaptive libraries are simpler to use, but, unfortunately, writing the adaptation heuristics that power them is a daunting task. The overhead of online adaptive libraries, on the other hand, makes them unsuitable for a number of applications. In this paper, we propose to automatically generate heuristics in the form of decision trees using a statistical classifier, effectively converting an online adaptive library into an offline one. As testbed we use Spiral-generated adaptive transform libraries for current multicores with vector extensions. We show that replacing the online search with generated decision trees maintains a performance competitive with vendor libraries while allowing for a simpler interface and reduced computation overhead.**

***Keywords*-automatic performance tuning; library generation; high-performance computing; decision trees; statistical classifier; machine learning; fast Fourier transform; FFT;**

I. Introduction

As we approach the power wall, processor microarchitectures have become increasingly complex and diverse. Even a standard commodity platform offers now features such as multiple cores, short vector SIMD units, long pipelines, and a deep memory hierarchy. Optimizing for these features is very difficult and often out of reach for compilers, hence the task falls to the software developer.

One domain were this problem is apparent is high performance library development for computing functions. Common practice involves experts who write different optimized routines for the same function, one for each of the supported architectures (an example is Intel's IPP library [1]). This approach is costly: it requires continual development to efficiently support permanently changing computing platforms.

A better approach pursued by several research projects is to automate the process of optimization and porting using automatic program generation or *adaptive libraries*. Such libraries provide degrees of freedom in how to compute the exact same function and the fastest choice is automatically determined by empirical search. Adaptive libraries have proven successful in various domains including basic dense linear algebra (ATLAS [2]) and linear transforms (FFTW [3] and Spiral-generated libraries [4]).

This work was supported by NSF through awards 0325687, 0702386, by DARPA (DOI grant NBCH1050009), the ARO grant W911NF0710416, and by Intel.

In this paper, we distinguish between *online* (at runtime) and *offline* (at installation time) adaptation. Online adaptive libraries require search every time the input specification (typically the input size) changes. An example is FFTW where this process is called *planning*. In offline adaptive libraries, the search is done only at installation time. Subsequently, all problem sizes are supported, without rerunning search. An example of such a library is ATLAS: during installation, it automatically produces a tuned implementation of basic linear algebra subroutines (BLAS). A summary of the different properties of the libraries is presented in Table I using the discrete Fourier transform (DFT) as example. We note that FFTW also possesses a hand-written heuristic which provides mixed performance results as we will show later. and

Observe that the interface in all libraries is the same requiring two distinct calls in a way that is similar to currying (partial application). In the first phase, `d = dft(n)`, the libraries perform initialization, which includes precomputation of the trigonometric constants (called *twiddle factors*), and in the case of the FFTW, the library searches for the fastest way to compute the DFT of the given size n. The actual computation is only performed in the second phase when the data is provided. The time for search is only amortized if many computations of the same type (here size n) are performed. The advantage of non-adaptive and offline adaptive libraries is that the initialization step is asymptotically faster than the computation step ($O(n)$ vs. $O(n \log n)$), but in practice it is still slower than the computation step, until n is at least several thousand points. In an online adaptive library, the adaptation process usually must perform at least one timing of the entire computation (and in reality, many more timings), and thus is guaranteed to take longer than just the computation step. In practice, for larger problem sizes, it may take several hours, which is prohibitive for applications with permanently changing problem sizes.

The interface used by the libraries in Table I conflicts with many legacy applications, which use a simpler one-

Table I
DIFFERENT LIBRARY TYPES AND THEIR PROPERTIES USING THE n POINT DFT AS AN EXAMPLE.

Library type	Non-adaptive	Online adaptive	Offline adaptive
Prototype	IPP [1]	FFTW [3]	*this paper*
Interface	`d = dft(n)` `d(X,Y)`	`d = dft(n)` `d(X,Y)`	`d = dft(n)` `d(X,Y)`
Initialization cost	$O(n)$	$> O(n \log n)$	$O(n)$
Computation cost	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Adaptation mechanism	none	online (planner at runtime)	offline (at installation time)
User view			
When problem changes	-	replan	-
When platform changes	rebuy	replan	reinstall

call interface. Non-adaptive and offline adaptive libraries can emulate the one-call interface without substantial overhead. In online adaptive libraries, a caching system could be used to hide online adaptation behind the legacy interface, but this complicates usage, potentially increases memory footprint, and does not resolve the problem of the high latency in applications with changing problem sizes.

On the other hand, if a library provides no adaptation mechanism (as IPP), it has to be reoptimized with every change in the microarchitecture or a performance loss is incurred.

Offline adaptation enables the best of both worlds: low initialization overhead and automatic adaptation. Alas, the development is difficult since it requires the developer to come up with a framework to automatically find suitable heuristics that have to work well for all problem sizes. In ATLAS this was done successfully but the search space (which includes different blocking and unrolling sizes) is BLAS-specific and cannot be used for other libraries, e.g., for the DFT.

Contributions. In this paper we propose a machine-learning approach to automatically convert an online adaptive library into an offline adaptive library. Specifically, we use at installation time (offline) a statistical classifier on a training set of problem sizes to automatically derive heuristics in the form of decision trees supporting *all* problem sizes. The decision trees are inserted into the library to remove search. Benefits include the following:

1) The ability of using knowledge about some sizes to infer a general behavior and hence the knowledge (represented as decision trees) of how to handle all sizes. The search is hence concentrated at installation time which simplifies shared usage (e.g., one time deployment on a supercomputer).
2) The ability to simplify the interface and match legacy interfaces, without substantial initialization overhead.
3) The obtained libraries have bounded initialization overhead (online adaptive libraries can have essentially unbounded initialization time).
4) The possibility of pruning complex adaptive libraries into simpler ones by cutting choices that are suboptimal and never used (not done in this paper).
5) Finally, the generated decision trees provide a mechanism for the developer obtain an understanding of when certain choices are taken.

We demonstrate the viability of our approach with Spiral-generated online adaptive DFT libraries. These are among the fastest DFT libraries available, and have a search space that includes decisions such as the radix, threading and number of threads, buffering and size of buffer, and several others. The heterogeneous nature of the search space and the nature of our approach should make it applicable for adaptive libraries outside the transform domain.

II. FORMAL PROBLEM STATEMENT

We consider an online adaptive performance library that provides a computing function parameterized by one or multiple positive integers, typically the input size or sizes. For simplicity, we assume in our problem formulation a single parameter n; the formulation for several parameters is analogous. One parameter is sufficient for the DFT (Table I) but not for all the subroutines needed to compute the DFT (as discussed later).

The library has degrees of freedom in the computation. We call every degree of freedom a *choice* and model it as a set of positive integers $C \subset \mathbb{N} = \{0, 1, 2, ...\}$. Examples include binary choices $\{0, 1\}$ such as "threading or not," the choice of the number of threads, and the choice of radix. In the ATLAS BLAS generator, choices include various tile sizes and the degree of unrolling. Making a *decision* for a given choice C means choosing $d \in C$. The complete computation is specified by a finite sequence (list) of decisions $D = \{d_1, \ldots, d_k\}$. We denote the set of possible decision lists (which may have different lengths) for input size n with $\mathcal{D}(n)$. Hence, in the library $\mathcal{D}(n)$ is the search space for size n that is available for adaptation.

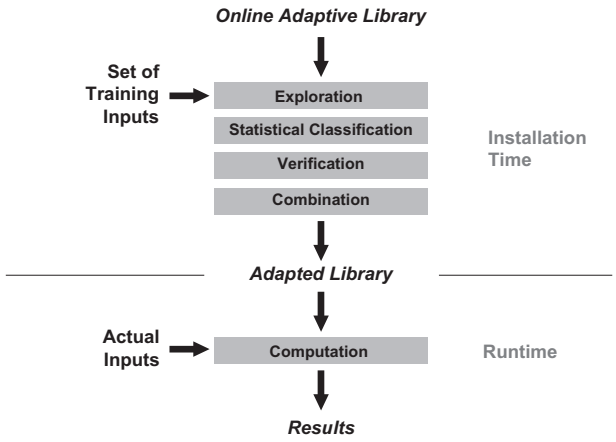


Figure 1. Our proposed offline adaptation process.

Problem statement. We now can formalize and distinguish the problems of online and offline adaptation for a given library. Given a list D of decisions, we assume $p(D)$ is the associated performance (higher is better) of the computation with the library.

Problem 1 (Online adaptation)**:** Given n, find

$$D_n = \operatorname*{argmax}_{D \in \mathcal{D}(n)} p(D).$$

Problem 2 (Offline adaptation)**:** Find a function

$$D: \ n \mapsto D_n,$$

where D_n is defined as in the online adaptation.

Note that in both problems, the optimum can only be guaranteed by exhaustive search which is not feasible. Hence D_n and D will be only approximations. For this reason, we will call the function D a *heuristic*.

For example, FFTW solves Problem 1 using a dynamic programming search and D_n can be stored as wisdom. FFTW also and alternatively provides a hand-written heuristic D. In ATLAS, the situation is simpler, since it generates a matrix multiplication for a fixed small size (the size is also determined by the search), which is then used as building block for all sizes. Hence, D is constant and can be inlined.

In this paper we use a machine learning technique to solve Problem 2 while achieving similar performance as search-based solutions to Problem 1. The function D will be generated as a set of decision trees.

Offline adaptation approach. Several efficient methods to solve Problem 1 have been proposed including dynamic programming [3], [4], and reinforcement learning based search [5]. Hence, we assume that Problem 1 is solved (even if time-consuming).

Our approach uses solutions D_n for several n as training set to solve Problem 2, i.e., to generate the heuristic D. The approach is summarized in Figure 1. It decomposes in four phases:

1) *Exploration:* For a training set of sizes n, D_n is computed.
2) *Statistical classification:* We use the classifier C4.5 [6] to generalize the knowledge learned during the exploration to produce D in the form of a set of decision trees. If needed, "hints" based on the library functionality are provided to help with the generalization.
3) *Verification:* The decision trees are verified and, if needed, corrected, to enforce the constraints imposed by the library.
4) *Combination:* Finally, the generated decision trees are inserted into the library as heuristics and replace the parts where the online decision code was called. The final result is a library adapted to the target platform.

III. Related Work

In the domain of high-performance library development, [7] is the only paper, to the best of our knowledge, that deals with automatically designing heuristics (which we denoted as Problem 2). The paper considers the Walsh-Hadamard transform, which has similar properties but is simpler than the DFT. The approach in [7] is to accurately predict the runtimes of the different algorithms by collecting WHT-specific features from the algorithms. The features are only related to the chosen radices; unrelated choices such as threading or buffering are not considered. For these reasons, the work is not easily extensible to other transforms or other domains since the feature selection is non-trivial and not automatic.

In the larger domain of optimizing compilation, two different subjects directly relate to this paper. First, *iterative compilation* (also called adaptive or feedback-driven) describes the process of successively compiling and executing the code in order to find the best optimizations for *one* given application. This topic is analogous, in a more general setting, to our Problem 1. Compilation time issues have led to smart timing frameworks [8] and various interesting search methods whether ad hoc [9] or rooting in machine learning [10] have been proposed.

The second trend in compiler research that relates to our paper is the work that propose to automatically tune the heuristics themselves, which in turn allows the compiler to perform better on *all* programs (which is similar to our Problem 2). Early work on the subject [11] suggested to use reinforcement learning in order to improve the scheduling of straight-line code. Closer to our work, [12], [13], [14] have suggested to learn whether or not an optimization should be triggered by learning decision trees using algorithms derived from C4.5. In contrast to our work, legality is never an issue for them and they only focus on dealing with a

Outlook	Temperature	Humidity	Windy	Decision
sunny	85	85	false	don't play
sunny	80	90	true	don't play
overcast	83	78	false	play
rain	70	96	false	play
rain	68	80	false	play
rain	65	70	true	don't play
overcast	64	65	true	play
sunny	72	95	false	don't play
sunny	69	70	false	play
rain	75	80	false	play
sunny	75	70	true	play
overcast	72	90	true	play
overcast	81	75	false	play
rain	71	80	true	don't play

Figure 2. The "weather" machine learning data set.

single heuristic whereas we generate all the heuristics of the library with this method. Other researcher have pursued the same goal with different search methods such as genetic programming [15]. In this area, research essentially focuses on predicting whether an optimization should be triggered but leaving the actual parameter choices to an heuristic. Predicting the actual parameters is needed for our work, in the spirit of what is achieved by [16] exploring the model space or [17] exploring the feature space.

IV. Background: Inducing Classification Models

The goal of decision tree learning is to create a model that classifies records based on a training set of already classified records. It is built by recursively partitioning the training set using well chosen tests. The interest is two-fold: On the one hand, it helps *summarizing* and *understanding* the already known training data and on the other hand it gives a *simple* mechanism to *infer* a classification for new cases. The most famous algorithm for producing such decision trees is probably Quinlan's C4.5 [6], which is based on his earlier algorithm ID3 but supports numerical features. The algorithm is best understood by walking through an example and we will use for this purpose the famous "weather" data set.

Example: Golfing or not golfing. A golf manager has observed that the attendance varies greatly depending on the weather (Figure 2) and is trying to understand the pattern behind it in order to manage his staff better. Before explaining the algorithm any further, we invite the reader to directly look at Figure 3, which is the decision tree produced by C4.5 for such a data set. Observe how the tree concisely captures the decisions in the table and also enables generalization for cases not contained in the table. The main difficulty is to determine which question should be asked in order to partition the cases in the most meaningful way.

Entropy of an event. ID3 and C4.5 are both based on Occam's razor, preferring simpler explanations over complicated ones: the goal is to always maximize the information

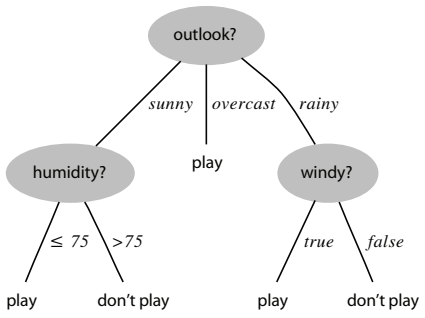


Figure 3. The decision tree generated by C4.5 for the "weather" dataset.

gain. To do this, the algorithms rely on the concept of *entropy*, which is a measure of the information content of a distribution, and was first introduced by Shannon [18],

Formally, suppose that the final decision can take any of the values $\{d_1, \ldots, d_n\}$ (here: golfing or not) and that the feature a (e.g., temperature) can take any of the values $\{a_1, \ldots, a_m\}$. We denote by $P(d_i|a = a_j)$ the conditional probability that decision d_i is made given that a takes the value a_j. By definition, the quantity

$$H(a = a_j) = -\sum_{i=1}^{n} P(d_i|a = a_j) \log_2 P(d_i|a = a_j)$$

is called the *entropy* of the event $(a = a_j)$ and measures the level of uncertainty that remains about the final decision if the said event happens. It is computed in *bits* (a fair coin has an entropy of one bit).

In our example, we can use Table 2 to first compute all conditional probabilities and then the entropy of events such as

$$\begin{aligned} P(\text{play}|\text{windy=false}) &= 6/8, \\ P(\text{don't play}|\text{windy=false}) &= 2/8, \end{aligned}$$

which yields

$$H(\text{windy=false}) = -(6/8 \log_2(6/8) + 2/8 \log_2(2/8)) = 0.81.$$

We see that, once golfers know that the outlook is overcast, their decision is already taken (they will definitely play). If it is not windy, uncertainty remains.

Entropy of a feature. Computing the weighted sum of the entropies over all the possible values a feature may take yields the entropy of the feature:

$$H(a) = \sum_{j=1}^{m} P(a = a_j) H(a = a_j).$$

The feature with the smallest entropy is the one that best partitions the training data set and is therefore the one that should be placed at the root of the decision tree. Recursively applying this process yields a full decision tree.

In our example, we observe that the "outlook" feature discriminates better than the "windy" feature, hence it becomes the root of the decision tree in Table 3:

$$H(\text{outlook}) = 5/14 \cdot 0.97 + 4/4 \cdot 0 + 5/14 \cdot 0.97 = 0.69,$$
$$H(\text{windy}) = 6/14 \cdot 1 + 8/14 \cdot 0.81 = 0.89.$$

Numerical features. In our example, "temperature" and "humidity" are both described by continuous ranges rather than by discrete classes and therefore, cannot be directly taken into account by the above algorithm. It is clear, however, that any numerical range can be split into two classes using a threshold: one containing values that are bigger and the other containing values that are lower or equal. This process is called discretization and C4.5 automatically selects the threshold that provides the biggest information gain.

Shortcomings. C4.5 is limited to classifying rectangular regions in the feature space. This is due to the conjunctive partitioning system that can only produce expressions of the type $(x > 16) \wedge (y > 3) \wedge (x \leq 70)$. Therefore, it fails at properly handling xor and parity problems.

In the context of adaptive numerical libraries, C4.5 will therefore not be able, unless it is "hinted", to recognize a decision that is beneficial only if two features are equal (e.g., the input and output strides). Also it will not be able to recognize number theoretic properties that may be significant (e.g., divisibility for the radix choice).

V. Background: Structure of a Recursive DFT Library

In this section we explain the structure of the search space in recursive state-of-the-art adaptive libraries computing the discrete Fourier transform (DFT). The libraries considered are generated by Spiral; FFTW offers a similar structure. Most importantly, we will see that the search space can be represented as a graph involving a set of heterogeneous decisions.

DFT. The DFT is the matrix-vector product $y = \mathbf{DFT}_n\, x$, where $x, y \in \mathbb{C}^n$ are the complex input and output vector, and

$$\mathbf{DFT}_n = [e^{-2\pi i k\ell/n}]_{0 \leq k,\ell < n}, \quad i = \sqrt{-1}.$$

FFT. Fast Fourier transform algorithms (FFTs) reduce the complexity of the DFT from $O(n^2)$ to $O(n \log(n))$ and can be written as matrix factorizations of $\mathbf{DFT}_n$. For example, the famous Cooley-Tukey FFT can be written as

$$\mathbf{DFT}_n = (\mathbf{DFT}_k \otimes I_m) T^n_m (I_k \otimes \mathbf{DFT}_m) L^n_k, \quad (1)$$

where $n = km$. Here, I_n is the identity matrix of size n; T^n_m is a diagonal matrix and L^n_k a stride permutation matrix that maps the vector elements as $i(n/k)+j \mapsto jk+i$. Finally, the tensor (or Kronecker) product $\otimes$ of two matrices is defined as

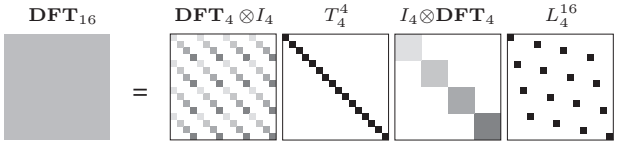
$$A \otimes B = [a_{k,l} B], \quad \text{where } A = [a_{k,l}].$$

```
void dft(int n, cpx *y, cpx *x) {
    if (use_dft_base_case(n))
        dft_bc(n, y, x);
    else {
        int k = choose_dft_radix(n);
        for (int i=0; i < k; ++i)
            dft_strided(m, k, t + m*i, x + m*i);
        for (int i=0; i < m; ++i)
            dft_scaled(k, m, precomp_d[i], y + i,
                       t + i);
    }
}

void dft_strided(int n, int istr, cpx *y, cpx *x)
                 { ... }
void dft_scaled(int n, int str, cpx *d, cpx *y,
                cpx *x) { ... }
```

Figure 4. FFTW 2.x-like implementation of the DFT. In this code, degrees of freedom are set by heuristics (instead of being searched online by the planning system). For brevity, auxiliary recursion steps `dft_strided` and `dft_scaled` are not detailed entirely.

We show below a visualization of the non-zero values in the matrices in (1) for $k = m = 4$.



In both tensor products, all parts of equal gray shade constitute a single $\mathbf{DFT}_4$. We observe that all four matrices are sparse, that the computation uses a divide-and-conquer approach, and that there is a degree of freedom (choice of $k|n$).

Mutually recursive implementation. The above FFT suggests an implementation in four passes corresponding to the four matrix factors. Both tensor products become loops which call smaller DFTs recursively. However, a better approach (taken, e.g., in FFTW 2.x) uses only two passes hence improving locality.

Namely, the explicit (and expensive) permutation L^n_k is replaced with a readdressing in the subsequent smaller DFTs. Similarly, scaling by T^n_m is fused with the subsequent DFTs. However, this creates the need for auxiliary functions, variants of the DFT with modified interfaces:

$$\underbrace{\mathbf{DFT}_n}_{\texttt{dft}} = \underbrace{\left((\mathbf{DFT}_k \otimes I_m) T^{km}_m\right)}_{\texttt{dft\_scaled}} \underbrace{\left((I_k \otimes \mathbf{DFT}_m) L^{km}_k\right)}_{\texttt{dft\_strided}}$$

We call the recursive functions needed for the computation *recursion steps* following [4].

The pseudo-code for such an implementation is displayed on Figure 4. Since the recursion must eventually terminate, base cases are provided for each recursion step in the form of unrolled codelets of fixed size (denoted with a `bc` for base case in the code). Following FFTW, we assume that `dft_scaled` is always a codelet.

Search space. The implementation in Figure 4 does not fully specify the computation but leaves choices that can be used for online adaptation (as was done in FFTW 2.x). The choices are the radix and whether to use a codelet or not. Further, note that the auxiliary recursion steps whose implementation is not displayed may possess their own choices. Also, they have the stride as additional parameter, which can impact the decision.

Figure 5 shows a representation of this search space as *decision graph* involving recursion steps (gray boxes) and choices (white boxes). It is a classic static closure graph augmented with the choices in the library. The outgoing edges of choices are labeled with decisions and connect to spawned recursion steps. For example, a decision on the radix recurses as shown in Figure 4. The outgoing edge of a recursion step may connect to a choice or to another recursion step that it calls without choice.

We make further observations:

1) Choices of the same type (e.g., "base case?") may occur multiple times inside the graph since they correspond to different recursion steps.
2) The decision graph contains cycles due to the recursive nature of the library and hence the search space. During the decision process, the same choice box can hence be encountered several times. For example choice of radix for a strided DFT of size 128 and later choice of radix for a strided DFT of size 16. What cannot be seen in the decision graph is that the decision procedure eventually terminates.
3) A given recursion step is implemented using zero, one, or more recursion steps. Therefore, open choices are of different nature: a recursion step is *either* implemented as a base case or not, but an actual recursion always uses *two* recursion steps.

Advanced implementations. The search space gets considerably more complicated with state-of-the-art libraries. The reason is in the support for vectorization, multithreading, and advanced memory hierarchy optimizations [3], [19], [4]. The latter includes support for buffering and for on-the-fly twiddle factor computation. These optimizations require transformations of (1) that produce additional recursion steps together with additional choices.

Figure 6 shows the decision graph of such a library generated by Spiral. Observe the increase in the number of recursion steps and the associated increase in choices inside the library.

Library generation. The Spiral project [4] has demonstrated that online adaptive libraries for transforms as the one discussed above can be computer-generated directly from an algorithm like (1). The generator works for a variety of transforms and can generate different libraries for each. Our work is designed to interface with each of these libraries to automatically convert them into offline adaptive libraries. Deriving and writing the heuristics by hand in each case would be unfeasible.

VI. Generating Decision Trees for Libraries

In this section, we first explain how statistical classification can be used to convert an online adaptive library into an offline one. Then we explain two techniques, *hinting* and *automatic correction* that enlarge the class of problems that the classifier can tackle.

A. Mapping of the Problem

Figures 5 and 6 show the structure of choices inside an adaptive library. These are used to tune to the hardware for a given problem size. As said before, each of these choices depends on the parameters or arguments of its recursion step: the questions read "should the DFT *of size 1024* be threaded?", "should the strided DFT *of size 16 and stride 4* be implemented as a base case?," and so on.

First we note that applying the classification framework from Section IV implicitly assumes that the best decisions only depend on the arguments of the current recursion step. In other words, the context of the function call has no significant impact and therefore every subprogram can be optimized independently. This assumption is equivalent to the one underlying *dynamic programming*, the preferred search method in this domain.

Providing a deterministic library means that heuristics are given for each one of these choices. Providing an *offline* adaptive library means that *customized* (i.e., platform specific) heuristics are given for each one of these choices. We use C4.5 to generate these customized heuristics.

Features. Selecting relevant features is the crucial problem for machine-learning based compilation. However, given our approach and assumption above, the features are precisely the arguments of each recursion step, omitting pointer addresses. For example, in Figure 4, the only relevant feature for the heuristics of `dft` is the size `n`; for `dft_strided`, it would be the size `n` and the input stride `istr`.

Decisions. The set of choices is fixed by the given online adaptive library (e.g., Figures 5 and 6). Since each choice takes numerical values (see Section II), C4.5 is applicable.

Training set. The training set for the main recursion step has to be selected by the developer of the library. As we will show in Section VII, it is interesting to choose a variety of sizes that encompasses the different performance regimes of the library. Note that training cases for the additional recursion steps are automatically derived, since they consist of the different possibilities that stem from the main recursion step.

B. Advanced Manipulation of the Decision Trees

C4.5 is limited from a learning point of view and might require good *hints* to produce good results. Due to the finite character of the training set, it might also generate trees that

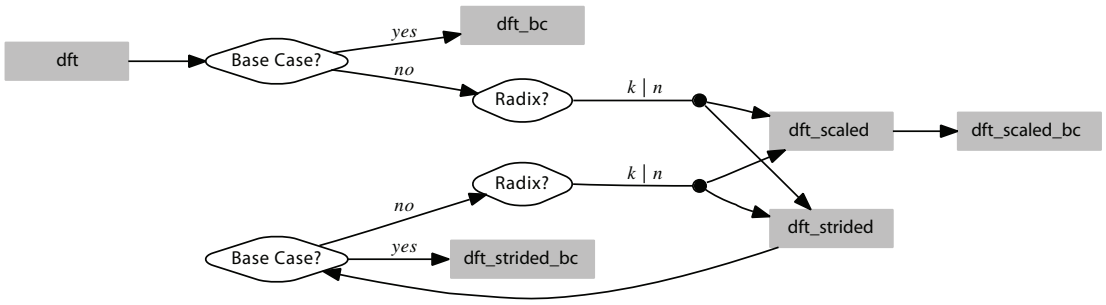


Figure 5. Decision graph underlying the implementation from Figure 4. Note that, in the case of radix choices, both children actually need to be computed.

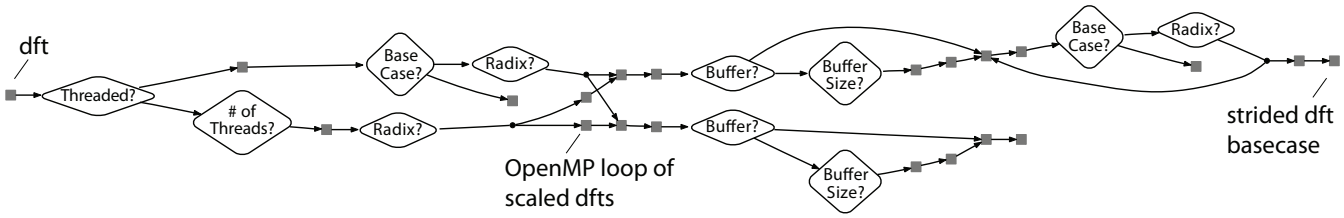


Figure 6. Decision graph underlying a vectorized, multi-threaded and buffered implementation of the DFT. Recursion step names are omitted for readability except for the initial `dft` recursion step and two examples. The goal of this paper is to automatically provide heuristics for all the choices (white boxes).

cannot be generalized as heuristics and therefore require an *automatic correction* pass.

Hinting. C4.5 can only cut the feature space into orthogonal rectangles and can only consider one dimension at a time. In some cases, these restrictions prevent a good generalization which in turn leads to a disappointing performance.

For instance, we have seen that in DFT libraries the choice of the radix must divide the size so the performance actually depends on the prime factorization of a number. Since multiples of 2 and multiples of 3 are interleaved and even mixed on the real axis, C4.5 is not able to discover by itself that both groups naturally exhibit a very different behavior. The tree C4.5 then produces treats individual cases, shows no global understanding of the problem and, ultimately, performs poorly.

However, it is possible to drastically improve the quality of the trees by providing "hints" to the classifier. A hint is a function that is directly computed from the features and fed to the classifier as if it was an extra feature. For instance, in the above case, hints that could be provided are the number of powers of 2 and of powers of 3 in the prime decomposition of the size. In practice, this creates two new dimensions, which enables the selection of meaningful groups using only rectangles. Finally, note that designing hints is not pushing the burden onto developers: anybody using DFTs knows that powers of two sizes are inherently very different from non-powers of two. Writing a hint is only giving this information to the classifier without explaining the implications of such a difference—the classifier will figure that out for itself.

Figure 7 shows a generated heuristic that chooses a radix as a function of the size. The functions `nfactor(f, n)`, that computes the number of times the factor `f` appears inside `n`, were provided as hints to the classifier. The comments in the code display the effect of the automatic correction which is the topic of the following paragraph. It can be noticed that, for this recursion step, this library and this training set, the chosen radix is often, *but not always*, the largest integer of $\{2, 3, 4, 6, 12, 18\}$ that divides the size.

Automatic correction. The C4.5 classifier is, in some way, short-sighted: it can come up with decisions that are not correct in the general case but were true inside the finite training set. For instance, if the classifier trains only on the set $\{12, 14, 16, 18, 20, 22, 24\}$, it might conclude that the best radix for any number divisible by 3 is 6 (since all numbers of the set divisible by 3 are also divisible by 6), overlooking the fact that choosing such a radix also needs a factor 2 which might lead to unpleasant surprises at runtime.

Consider Figure 7. We want to make sure that the returned radix is always valid, that is, always divides the input size. As an example, let us focus on the first `return 8`. It is easy to prove that 8 is always a correct radix because the above condition, `nfactor(2, n) <= 3` is false so `n` is necessarily a multiple of 8. If we focus on the `return 18` line now, we observe that `n` is only guaranteed to be divisible by 6. A correction needs to take place there.

In our system, this verification phase is mechanized by traversing each tree and automatically correcting decisions (leaves) that cannot be justified by the information contained in the internal nodes. Correcting means that decisions are

```
choose_dft_radix(int n) {
   if ( nfactor(3, n) <= 0 ) {
      if ( nfactor(2, n) <= 2 )
         return 2;
         //Corrected to: error(`` no divisors'');
      else {
         if ( nfactor(2, n) <= 3 ) return 4;
         else return 8;
      }
   }
   else {
      if ( nfactor(2, n) <= 1 ) {
         if ( nfactor(2, n) <= 0 ) return 3;
         else {
            if ( nfactor(3, n) <= 1 ) return 2;
            else return 6;
         }
      }
      else {
         if ( nfactor(2, n) <= 3 ) {
            if ( nfactor(2, n) <= 2 ) {
               if ( nfactor(3, n) <= 1 ) return 6;
               else return 12;
            }
            else return 18;
            //Corrected to: else return 6;
         }
         else return 12;
      }
   }
}
```

Figure 7. A computer-generated heuristic for the implementation from Figure 4. The function `nfactor(f,n)` returns the number of times the factor `f` appears inside `n` and constitutes an hint. After the generation of the heuristic, the automatic correcter enforces the divisibility policy (the effect of which is displayed in commented lines that replace the line above them).

either changed to more conservative ones or additional internal nodes are added. We also insert a proper error message in case all tests are failed which means that the situation was not learned during training.

VII. Experiments

Platform. Our benchmark platform has two dual core 3 GHz Intel Xeon 5160 processors (server version of Core 2 Duo) with 4 MB of shared L2 cache per processor, running Linux in 64-bit mode. We generate different online adaptive C++ libraries using Spiral [4], which are then inserted into our tool and converted into offline libraries. Libraries are compiled using the Intel C/C++ Compiler 10.1. We compare against FFTW 3.2 alpha 2 and Intel IPP 5.3.

All libraries were timed out of the box. In particular, slightly better performance for FFTW and Spiral-generated libraries could have been achieved for mid-range sizes by ensuring that only two threads are used and properly pinned to the processors. However, these choices are not automatically handled by the library and hence not considered by our tool.

We use pseudo-GFlop/s to show performance, which is standard for the DFT. It assumes a (real) operations count of $5n \log_2 n$ (a slight overestimate) for the complex DFT of size n.

Clothesline experiments. First, we want to demonstrate that useful heuristics can be learned. To do this, we show that the quality of the generated heuristics improves with the size of the training set and approaches the best runtimes found.

The experiment is performed using the Spiral-generated DFT library corresponding to Figure 6, i.e., the library has eleven choices. Only two-powers are considered. First, online search is used to find the best runtimes for all 20 sizes 2^1–2^{20} (2–1M). The competitiveness of these runtimes, and hence the library is established by comparing to FFTW and IPP in Figure 8(d). The runtimes serve as ground truth or *clothesline*.

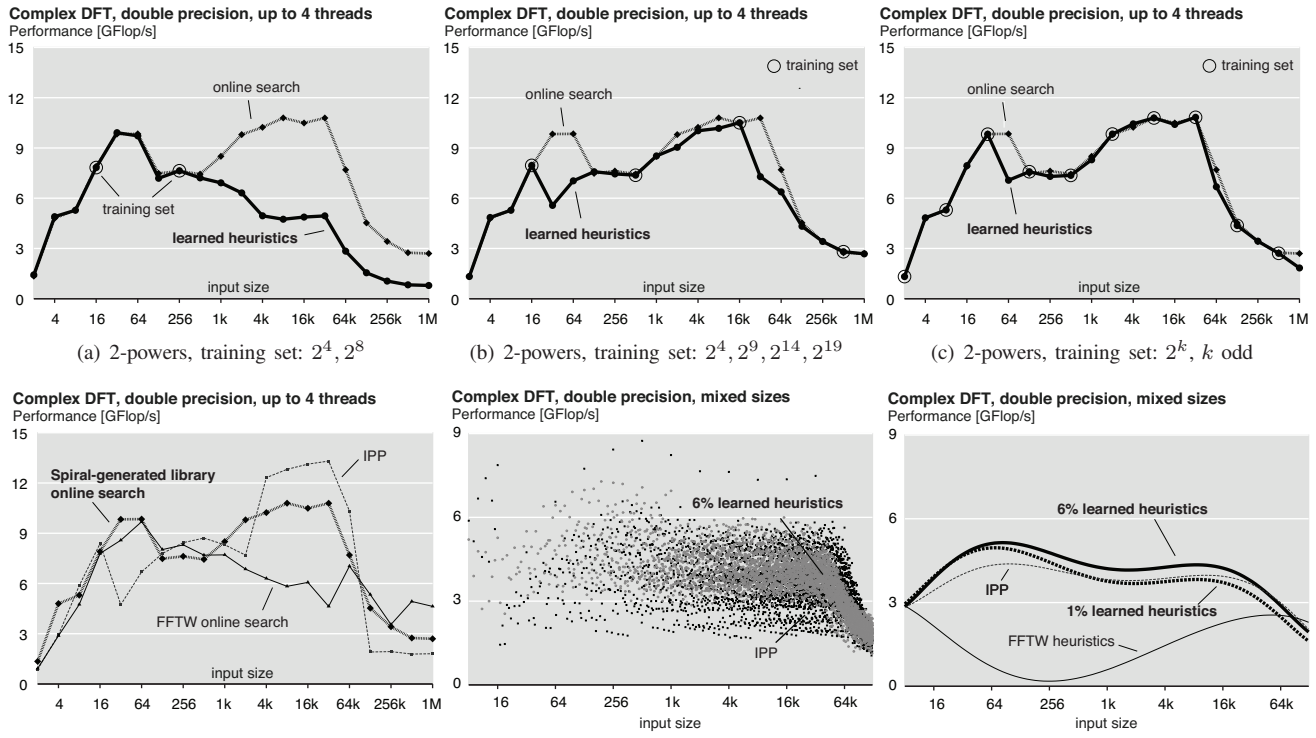
Next, a subset of the 20 sizes is chosen as training set for our tool. These sizes serve as *clothes-pins*. From the training set the tool generates heuristics for all sizes and inserts them into the library. The library is then timed on all sizes. The resulting line is the *cloth*. The goal is to verify that with increasing number of pins (increasing training set) the cloth hangs straighter, i.e., approaches the clothesline (ground truth).

The results are shown in Figure 8(a)–(c). In each case the training set is marked by circles. In Figure 8(a), the training set contains only two small sizes: 2^4 and 2^8. For these sizes, threading is irrelevant, hence the generated heuristics avoid it, which then yields bad performance for larger sizes. In Figure 8(b), the training set includes four sizes that are well distributed so different scenarios (e.g., in-cache and out-of-cache, threading or not) can be learned. The performance already closely approaches the ground truth. In Figure 8(c), the training set consists of half of the sizes and the resulting library is practically as fast as the online adaptive one in all sizes.

Note that for a given choice in Figure 6, a chosen training size may give a varying number of training points: zero (e.g., the buffer size is not known if the best alternative doesn't buffer), one, or more than one (e.g, due to the recursion, the best radix is simultaneously selected for the problem and all subproblems).

Mixed sizes experiments. The learning approach becomes particularly relevant if we consider a much larger set of sizes by including non-two-powers. We do so in the second experiment which considers all sizes up to 2^{18} (256K) that decompose into prime factors smaller or equal than 19. We generate two offline adaptive libraries that differ in the size of their training set, which respectively amounts for 1% and 6% of all sizes. As can be seen in Figure 8(e), and as is known, DFT performance varies greatly depending on the prime factorization of the size, which makes it very irregular and difficult to read. In this first plot, it can however be seen that the performance of the library with our generated heuristics exhibits somewhat less variation than IPP.

Next we measured the performance for all sizes up to



(a) 2-powers, training set: $2^4, 2^8$

(b) 2-powers, training set: $2^4, 2^9, 2^{14}, 2^{19}$

(c) 2-powers, training set: 2^k, k odd

(d) The 2-powers library is competitive

(e) Mixed sizes

(f) Mixed sizes (trendlines)

Figure 8. (a), (b), and (c) show *clotheslines* experiments with a Spiral-generated online adaptive DFT library, whose competitiveness is shown in (d). The clotheslines experiments demonstrate the effectiveness of our tool in learning and generating heuristics. Both (e) and (f) present the DFT performance of different libraries on all numbers smaller than 256K that decompose into prime factors smaller or equal than 19. In (f), the data is interpolated using a sixth order logarithmic trendline. This plot also includes FFTW using its heuristics mode instead of online search.

256K for all libraries that are not online-adaptive: IPP, FFTW with the included (hand-written) heuristics, and our Spiral-generated libraries with generated heuristics for the 1% and 6% training set. To reason about the performance data, we computed a logarithmic regression of order 6 in each case and present them in Figure 8(f). First, as expected, we observe that the library that is trained on the larger training set (6%) performs better than the one with the 1% training set. Second, we observe that the 6% generated-heuristics library performs better than IPP. Precise computation shows that the average performance gain is 10.7%. Finally, the poor performance of the hand-written heuristic mode of FFTW shows that writing heuristics is no simple task.

VIII. Conclusion

We proposed a machine learning technique to automatically convert online adaptive libraries into offline adaptive libraries by generating heuristics. In practice this means that the burden of searching can be pushed from runtime to installation, which improves usability considerably. The performance penalty incurred by doing so is very minor, at least for the DFT libraries we considered.

Our method is entirely automatic and capable of handling a very diverse set of choices inside an online adaptive library including the choices of radix (i.e., the block size), threading, and buffering. This ability should make the method applicable to other problem domains and libraries.

Together with Spiral's previous capability to automatically generate online adaptive libraries, we hence demonstrated for the first time the computer generation of a state-of-the-art offline adaptive library, directly from an algorithm specification.

References

[1] Intel, "Integrated Performance Primitives (IPP) 6.0, User Guide."

[2] R. C. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software (ATLAS)," in *Proc. Supercomputing*, 1998.

[3] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[4] Y. Voronenko, F. de Mesmay, and M. Püschel, "Computer generation of general size linear transform libraries," in *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, 2009, pp. 102–113.

[5] F. de Mesmay *et al.*, "Bandit-based optimization on graphs with application to library performance tuning," in *Proc. Int'l Conf. on Machine Learning (ICML)*, 2009, pp. 729–736.

[6] J. R. Quinlan, *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993.

[7] B. Singer and M. Veloso, "Learning to construct fast signal processing implementations," *Journal of Machine Learning Research, special issue on ICML 2001*, vol. 3, pp. 887–919, 2002.

[8] K. D. Cooper *et al.*, "ACME: adaptive compilation made efficient," *SIGPLAN Not.*, vol. 40, no. 7, pp. 69–77, 2005.

[9] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, 2006, pp. 319–332.

[10] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 1999, pp. 1–9.

[11] E. Moss *et al.*, "Learning to schedule straight-line code," in *Proc. Advances in Neural Information Processing Systems (NIPS) '97*, 1997, pp. 929–935.

[12] A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristics," in *Artificial Intelligence: Methodology, Systems, Applications*. Springer, 2002, pp. 41–50.

[13] J. Cavazos and E. Moss, "Inducing heuristics to decide whether to schedule," *SIGPLAN Not.*, vol. 39, no. 6, pp. 183–194, 2004.

[14] B. Calder *et al.*, "Evidence-based static branch prediction using machine learning," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 1, pp. 188–222, 1997.

[15] M. Stephenson *et al.*, "Meta optimization: improving compiler heuristics with machine learning," in *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, 2003, pp. 77–90.

[16] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, 2005, pp. 123–134.

[17] H. Leather, E. Bonilla, and M. O'Boyle, "Automatic feature generation for machine learning based optimizing compilation," in *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, 2009, pp. 81–91.

[18] C. E. Shannon, "A mathematical theory of communication," *Bell system technical journal*, vol. 27, 1948.

[19] M. Püschel *et al.*, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.