Automatic Recognition of Computational Kernels for Platform-Dependent Code Optimizations

María H. Rodríguez Blanco, Georg Reinke, Gerd Ascheid, and Rainer Leupers
Institute for Communication Technologies and Embedded Systems
RWTH Aachen University
52056 Aachen, Germany
Email: rodriguez@ice.rwth-aachen.de

***Abstract*—We present a novel approach that assists the task of porting code to an embedded platform. Our tool automatically identifies code segments in the input program that can be replaced with optimized kernels from a platform-dependent library. Using a C-function as a model that describes the computational kernel, the tool identifies equivalent code regardless of syntactic and computational variations. For a case study using the Texas Instruments C66x DSP library, our approach identified code replacement opportunities that resulted in runtime performance speedups of up to 2.2x.**

I. Introduction

Embedded platforms vendors often provide libraries that are highly tuned to their target architectures. These libraries consist of functions implementing computational kernels, often used in a given domain. A common strategy in the task of porting an application to a specific platform is to replace functionally equivalent code segments with optimized kernels from those libraries. We propose an approach that assists this task by automatically spotting locations in the input program where these optimized library routines can be employed. Our tool performs a recognition that otherwise the programmer would perform manually, which requires time and a deep understanding of the algorithm of the input program.

The proposed approach belongs to the algorithmic concept recognition field. Associated technologies, including ours, consist of the following elements: *i)* a set of *algorithmic concepts*, which in our case correspond to a set of computational kernels for which an optimized version is available in a target-dependent library, *ii)* their corresponding *concept model*, used as reference for the identification of algorithmic concept *instances* within an input program. For the concept model generation, we use a graph representation of a user-defined *sample implementation* per computational kernel, namely its Program Expression Graph. Finally, *iii)* a *recognition system* performs the detection of the concept instances in the source code, here achieved by means of a pattern matching algorithm.

The problem of concept recognition exposes different challenges. One of them is the so-called *variations* [1]. It refers to differences allowed by the programming language within functionally equivalent implementations of a given concept. In this work, we handle two source code variation types: syntactic and computational variations, which proved to be significant for our use case. They are considered for the concept model generation, so that our recognition system can identify instances in the input source code regardless of the presence of these variations.

The main contributions of this paper are the following:

1) A novel concept recognition mechanism that identifies opportunities for replacement of code segments with highly-optimized kernels (Section IV).
2) An automatic concept model generation that handles syntactic and computational variations (Section V).
3) A performance evaluation of the proposed recognition algorithm (Section VI).
4) Results and analysis of a case study using a commercial embedded platform from Texas Instruments (the Keystone I [2]) and the C66x DSP library[3] (Section VII).

II. Related Work

Automatic kernel recognition has been studied for a wide variety of application domains and objectives. Similar to our work, the scope of the Automatic *Algorithm Recognition* and Replacement approach [4] is to recognize code segments to be replaced by library calls. It first extracts different *subprograms* (sets of statements) from the input program. To minimize the effect of variations, they preprocess the code through a set of semantics-preserving transformations performed by an optimizing compiler. Then, for testing if a subprogram is an *algorithmic instance* of a *pattern* (an algorithmic concept), they convert both pattern and subprogram to a canonical form of their intermediate representation (IR). The pattern matching becomes a trivial pair-by-pair comparison of nodes. Their subprogram extraction allows to recognize subprograms consisting of non-contiguous (delocalized) statements. However, the recognition success of this approach relies on the non-trivial subprogram extraction problem. They need to extract statement groups that form meaningful algorithms and potentially match exactly against a pattern. They propose a heuristic, that as a consequence, restricts the pattern types, e.g. at least one statement should be a loop.

Our approach uses a subgraph isomorphism algorithm for pattern matching. Despite its higher complexity, it allows us to detect simultaneously all algorithmic instances of an input program, given a database of computational kernels. We have no further restrictions to our patterns except for their specification as a C-function. We also analyze the runtime performance

of our detection process, which performs well in practical cases. Except for results of the practical complexity of their heuristics, we did not find any evidence of their succesful recognition rate for comparison. Another difference is how we handle variations. We extend the PEG code representation to encode explicitly possible variations. Unlike the sequential application of compiler transformations, ordering does not play any role in our process.

In the scope of automatic parallelization, *XARK* [5] recognizes a collection of domain-independent computational kernels, such as inductions, reductions and array recurrences usable for a variety of techniques. An example is the replacement of an irregular reduction with a platform-optimized parallel version. There are also domain-specific approaches. *PRT* [6] recognizes parallelizable computational patterns that frequently occur in the DSP domain, mostly loop-based patterns. *MPIIMGEN* [7] is a code transformer that automatically transforms sequential image processing codes into parallel versions. *ALCOR* [8] refers as Parallelizable Algorithmic Patterns (PAP) a group of algorithms parallelizable by a common strategy. All these approaches are rule-based, sharing a common problem [4]. Rule-based systems require an expert of the internal workings of the system to add recognition capability and new patterns.

Earlier work of concept recognition seeks to support other application areas, mainly software maintainers and software engineering. A comprehensive survey is provided in [4].

III. Background

A. Program Expression Graphs

Program Expressions Graphs (PEG) have been proposed by Tate et al. [9] to represent intraprocedural imperative code with branching and looping constructs. Figure 1 (a) shows an example of a source code snippet and (b) its PEG representation. Conceptually, in a PEG each node represents an operation, outgoing edges represent operands, and the incoming edges represent uses of the result. In Figure 1, ϕ is a PEG operator that selects between the value of the second and the third child depending on the value of the first operand (y). It represents the merging of the two possible values of x. The incoming edge at the top is the return value in the code snippet.

Unlike this code snippet, the PEG representation of a complete C-function by definition has two return nodes, one representing the return value of the function, and the other one representing memory state changes. Additionally, according to [9], PEGs are referentially transparent, which means that all PEG operators (even branches and loops) are mathematical functions with no side effects. This characteristic enables equality reasoning, meaning we can insert information of equalities. *Equality Saturation* was proposed in [9] as a equality reasoning technique.

B. Equality Saturation

It consists of transforming and augmenting the PEG by repeatedly applying a set of equality rules. The process stops when the graph is *saturated*, i.e., when no rule in the set can be

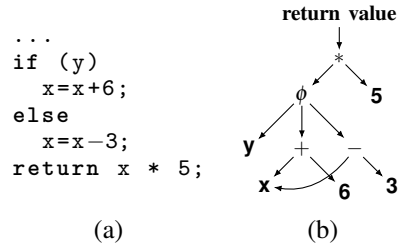

Fig. 1. (a) example code, and (b) its PEG representation

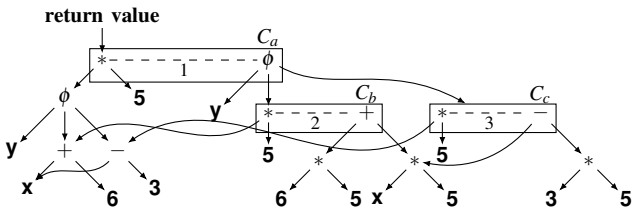

Fig. 2. EPEG example

further applied. The resulting extended graph is called EPEG. As an example, for the PEG in Figure 1 (b) we can obtain the EPEG of Figure 2 for the following exemplary rule set:

1) $\phi(a,b,c)*m = \phi(a,b*m,c*m)$
2) $(a+b)*m = a*m+b*m$
3) $(a-b)*m = a*m-b*m$

EPEGs are PEGs with additional equality information, captured in *clusters*. All nodes in a cluster are equivalent, i.e, represent the same value. Nodes with no equivalent partner form single-node clusters. For visual purposes in Figure 2, we do not draw boxes for single-node clusters. Also, nodes belonging to the same cluster, i.e., equivalent nodes, are connected with dashed edges. Its label refers to rule number proofing the equality.

An important characteristic for our approach is that an EPEG can efficiently encode multiple versions of the original program in a single representation [9]. Edges in EPEGs go from nodes to clusters. They represent several possible parent/children combinations. For example, by selecting one node from each cluster C_a, C_b and C_c, we can derive alternative components to build up one of the implicit PEGs. The EPEG in the figure implicitly represents five different PEGs (i.e., five distinct implementations of the same code).

IV. Proposed Approach Overview

Our objective is to recognize code segments in an input program that are functionally equivalent to the computational kernels of a platform-dependent library. The main challenge is that functionally equivalent implementations can differ due to the flexibility of the programming language. To solve this problem, we propose to use a PEG representation of every desired computational kernel, comprising the *algorithmic concept set*. We apply Equality Saturation using a rule set encoding typical variations that can prohibit the recognition. The resulting EPEGs represent the *concept models* that serve

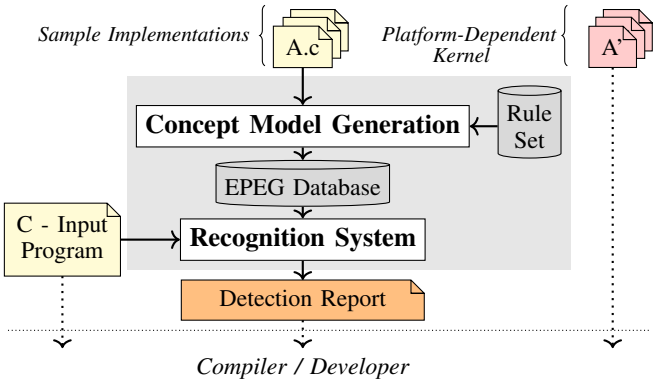

Fig. 3. Automatic recognition tool flow

as reference for the identification of *algorithmic instances* within the input program. To this end, we not only need to determine a proper rule set that can describe possible variations, but also we need to design a recognition system that performs the detection based on an EPEG representation.

Figure 3 shows the main components of our approach. As input it takes a *set of sample implementations* of the computational kernels and an input program, both written in C. The tool flow consists of two main phases: the concept model generation and the recognition process. The output is a report with the list of concept instances and their locations in the input program, where a developer or an external tool can perform the replacement with its corresponding optimized kernel. We use a C-function as a description of the functional behavior of an algorithmic concept. Therefore, inputs of an algorithmic concept are listed as parameters, and outputs are the return nodes of its PEG representation. These are sufficient to prove the functional equivalence of two implementations in the assumption that their inputs are equivalent.

V. Concept Model Generation

As described earlier, the *concept model* is generated using the Equality Saturation technique with rules explicitly encoding common variations that a programmer could implement. To be exhaustive in the enumeration of all possible variations is a hard problem and will impact the recognition performance of the tool. Part of our contribution is to find a minimum set of rules, that covers most practical cases. However, the addition of more rules to cover specific domain features is possible.

The generation consists of two steps. The first step converts the sample implementations to its PEG representations. An engine, publicly available in [10], returns the PEG representation out of the LLVM IR [11] of our sample implementations. The second step is to sequentially apply Equality Saturation with the variation-tuned rule set. At the end of the process, we obtain one EPEG (a concept model) per algorithmic concept, comprising the EPEGs database. In following subsections, we describe the variation types handled by our tool, what they are, and which kind of equality rules are needed.

TABLE I
Set of Rules for Computational Variations

C-Operator	Rules or Properties	# Rules
Arithmetic	Commutativity: * , + Associativity: * , + Distributivity: *	5
Relational	Inverse Operator Pair: (>,<),(!=,==), (>=,<=)	6
Bitwise	Shift-Multiplication:«,»	2
Logical	De Morgan's laws:!,&&,‖	2

```
if(a<w) a = a*2;
```
(a)

```
if(w>a) a = a<<1;
```
(b)

Fig. 4. Exemplary computational variations (a) *less than* with *multiplication* operator, and (b) *greater than* with *shift* operator

A. Rule Set for Computational Variations

For us, *computational variations* are concept instances with different means to compute the same result, as in Figure 4. In most high level languages, including C, there is more than one way to compute the same result. We relate this flexibility of the language to the language operators' mathematical properties. For example, the `*` operator's commutative property allows variations such as `c = a * b` and `c = b * a`. This additional knowledge has been given to the tool by means of equality rules, e.g., `A > B = B < A` or `!(A == B) = (B != A)`. We implemented a total of 15 equality rules, summarized in Table I, to handle most common computational variations in our use case. Notice that identity properties of arithmetic and logical functions, e.g. `X * 1 = X`, can be added. Since programmers do not usually write code that needs trivial simplifications, we left them out.

B. Rule Set for Syntactic Variations

Syntactic variations occur when the same functionality is implemented using different ways to bind values to names or a different selection of control constructs [1]. This degree of freedom is in most cases ruled by the C-grammar itself. As in the example in Figure 5, we can implement an iteration element using a `for` or a `while` loop. Their corresponding abstract syntax trees (AST) clearly differ.

Interestingly, not all of these variations need to be handled by a new equality rule. For example, this `for-while` variation actually results in the same PEG representation. Since PEGs relate to the Static Single Assignment (SSA) form, which has a higher level of abstraction than the AST, we normalize most syntactic differences by simply converting into a PEG. Also the assignment C-operators, e.g. +=, as well as the arithmetic operators ++ and --, do not reflect a syntactic variation in its PEG representation, but at the AST level.

However, this is not the case for member and pointer operators (e.g *a, &a, a[]). They offer a flexibility to express

```
for(i=0; i<N; i++)
  sum += A[i];
```

(a)

```
Aptr = &A[0];
i = 0;
while(i<N) {
  sum += *Aptr++;
  i = i+1;
}
```

(b)

Fig. 5. Vector summation kernel with (a) *for-array* based, and (b) *while-pointer* based variation

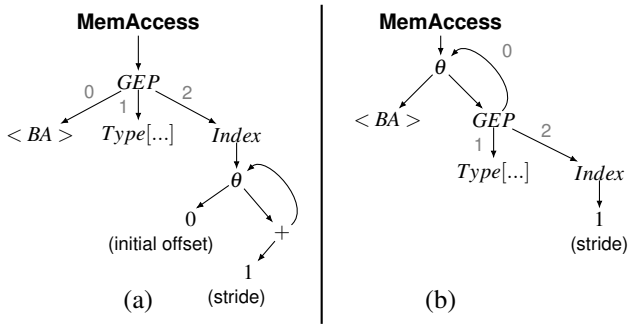

Fig. 6. Vector access PEG representation of an (a) *array* based, and (b) *pointer* based variation. BA:Base Address, GEP:GetElementPointer

memory accesses in different ways, that result in different PEGs. In C, a pointer can be used as an array and vice versa, as exemplified by Figure 5. The PEG in Figure 6 represents the memory access performed on each loop iteration for version (a) and (b) of Figure 5 respectively. The array-based version (a) represents the access sequence: $BA+0$, $BA+1...BA+i$, while the pointer version (b) represents the memory access according to the sequence: BA, $(BA)+1$, $((BA)+1)+1$,...$(((BA)+1)+1)...+1$.

To handle this, we distinguish among patterns accessing an element, a list, and a block, using arrays or pointers. Instead of specifying the equivalence among the different patterns variations with equality rules, we abstract the patterns into new nodes called `ElementAccess`, `VectorAccess` and `BlockAccess`. In Figure 6, both versions are abstracted to `VectorAccess(<BA>,<Type>,<InitOffset>,<Stride>)`. These abstraction rules allow the handling of syntactic variations due to array and pointer based notations.

VI. Recognition System

Every EPEG concept model in the database is a compact representation of multiple *concept instances*. The recognition system's task is to find one of the many variations encoded implicitly in the EPEG database matching a subgraph in the input program's PEG. This problem is called *subgraph isomorphism detection*. Our system is built on the decomposition-based subgraph isomorphism (DSI) approach [12], which solves the detection and the database organization. It consists of two main steps: decomposition and detection.

A subgraph isomorphism is detected whenever we can find a mapping associating an input graph nodes subset to an EPEG nodes subset, which completely represents a concept instance, i.e., one of its implicit PEGs. To guide this search, the detection step uses a representation of the concept model database, called *network* (created by the decomposition step). The network describes how the concept models can be recursively decomposed into smaller subgraphs until reaching single nodes. In this way, the detection first finds all occurrences of the individual nodes of the model in the input graph and then, guided by the network, it recursively merges them into larger components until a complete model is found.

In the DSI algorithm, input and model graphs are labeled graphs (i.e., they consist of a node set, an edge set and an associated function that assigns labels to its nodes). The PEG input graphs are also labeled graphs. Each node has a label which specifies the PEG operator being represented by the node. However, EPEGs are a more complex form of labeled graphs, due to the additional equivalence relations represented by the clusters. Therefore, we modified the DSI algorithms to identify clusters with equalities, so that alternative mappings can be successfully matched to a concept model.

A. Definitions and Notations

Our recognition algorithms work with two types of graphs: Program Expression Graphs (PEG) as input graphs, and EPEG as model graphs. Their definitions are based on the originals presented in [9].

Definition VI.1. A PEG G is a labeled, ordered, directed graph, defined as a 4-tuple $G=(N,E,\lambda,R)$, where N is the set of nodes, $E:N\mapsto N^*$ is a function mapping each node to its ordered list of child nodes, $\lambda:N\mapsto F$ is the node labeling function, F the set of mathematical functions (PEG operators) being represented by the node, and R is the set of nodes marked as return nodes. They are the roots of the PEG.

Definition VI.2. An EPEG is a 5-tuple $\mathcal{G}=(N,C,E,\lambda,R)$, where N is the set of nodes, C is a set of clusters that divides N into equivalence classes, $E:N\mapsto C^*$ is a function mapping each node to its ordered list of child *clusters*, λ is the node labeling function, and R is the set of return clusters.

We use C_A with a capital subindex or C with no subindex to denote a set of clusters. Anything else, e.g., C_i or C_1, is used to denote a single cluster, that by definition correspond to a set of nodes. Therefore, if $C=\{C_1,C_2,...,C_t\}$ is the cluster set of an EPEG, then we can state that $N=\bigcup_{i=1}^{i=t}C_i$ and $C_i\cap C_j=\emptyset$, for any $i,j\in[1,t]\wedge i\neq j$. Additionally, we introduce the following distinction between the clusters of an EPEG.

Definition VI.3. Let $C=\{C_1,...,C_t\}$ be the cluster set of an EPEG. We define $\widetilde{C}$ and $\widehat{C}$ as disjoint subsets of C such that

- $\widetilde{C}=\{C_i\in C:|C_i|=1\}$, $\forall i\in[1,t]$
- $\widehat{C}=\{C_i\in C:|C_i|>1\}$, $\forall i\in[1,t]$

In the previous definition, $|C_i|$ represents the number of nodes in the cluster C_i. We refer to $\widehat{C}$ as the set of *complex clusters* since they contain more than one node, and $\widetilde{C}$ as the set of *basic clusters*. As an example, the complex clusters in Figure 2 are basically those where at least one dashed edge

is present. Complex clusters are of special interest for the proposed decomposition approach. From each node belonging to a complex clusters, we can derive alternative components to build up one of the implicit PEGs.

Definition VI.4. Reached Clusters $\mathcal{R}_{\mathcal{G}}$: Given an EPEG $\mathcal{G} = (N, C, E, \lambda, R)$, we define $C_A \leftarrow \mathcal{R}_{\mathcal{G}}(C_i)$ as the function that returns the cluster set C_A that can be reached traversing only basic clusters, starting from the cluster C_i.

Algorithm 1: $\mathcal{R}_{\mathcal{G}}(C_i)$

```
1  C_A = {C_i};
2  if |C_i| = 1 then
3      Pick the node δ ∈ C_i;
4      foreach C_x ∈ E(δ) do
5          C_A ← C_A ∪ R_G(C_x);
6  return C_A
```

Algorithm 1 gives a simplified description of the $\mathcal{R}_{\mathcal{G}}(C_i)$ behavior. Notice that the return set C_A can contain complex clusters, but their children are not considered. Based on this function we define a special kind of EPEG subgraphs, which is the key element for our EPEG decomposition algorithm.

Definition VI.5. For a given EPEG $\mathcal{G} = (N, C, E, \lambda, R)$ with $C = \{C_1, ..., C_t\}$, we define $\mathcal{S}_\alpha = (N_S, C_S, E_S, \lambda_S, R_S)$ with $C_S = \{C_{s1}, ..., C_{sn}\}$ and $\alpha \in N$ as the *e-subgraph derived from* α, denoted $\boldsymbol{\mathcal{S}_\alpha \subseteq_e \mathcal{G}}$. Let $C_r = \{\alpha\}$ be a basic cluster, then

- $C_S = \mathcal{R}_{\mathcal{G}}(C_r)$
- $N_S = \bigcup_{i=1}^{i=n} C_{si}$
- $E_S(\beta) = E(\beta)$, if $\beta \in C_{si} \wedge |C_{si}| = 1, \forall i \in [1, n]$
- $\lambda_S(\beta) = \lambda(\beta)$ if $\beta \in N_S$
- $R_S = \{C_r\}$

By using this definition we can decompose the nodes of a complex clusters into basic clusters, and derive their corresponding e-subgraph, that represents alternative components for the constitution of an implicit PEG. By reunifying the return cluster of every e-subgraph into a complex cluster we can reestablish our original EPEG. Let α_1, α_2 be the nodes belonging to the complex cluster C_a of the EPEG in Figure 2. Therefore, $C_a = \{\alpha_1, \alpha_2\}$, where $\lambda(\alpha_1) = \phi$ and $\lambda(\alpha_2) = *$. We can decompose C_a into $\mathcal{S}_{\alpha_1}$ and $\mathcal{S}_{\alpha_2}$. This decomposition is shown at the top of Figure 7.

B. Decomposition of EPEG Model Graphs

Algorithm 2 describes how a single EPEG is decomposed. For every EPEG model $\mathcal{G}_y = (N_y, C_y, E_y, \lambda_y, R_y)$ belonging to the concept model database, the decomposition sequentially calls the process *DecomposeEPEG*($\mathcal{G}_y$,R_y). Similar to DSI, it consists of a recursive partition of the model graph into smaller subgraphs. Every partition decision is recorded as a tuple in a global and initially empty network $\mathcal{N}$.

Definition VI.6. The decomposition network $\mathcal{N}$ is defined as a finite set of tuples $\mathcal{N} = \{(S_i, S_{i1}, ..., S_{im_i}, \mathcal{E}) : i \in [1, q]\}$ where q is the total number of tuples belonging to the network, and

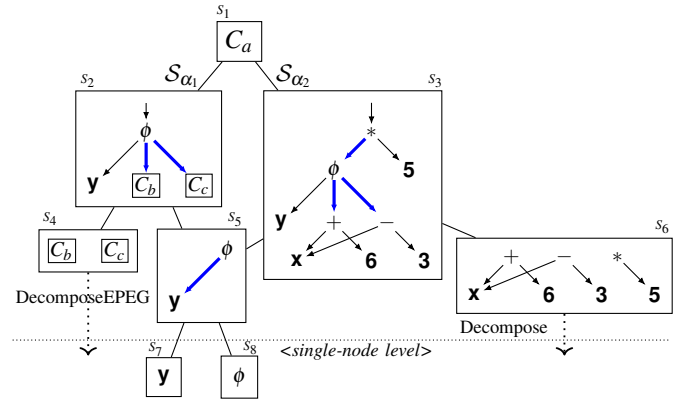

Fig. 7. Decomposition network. C_a decompose into $\mathcal{S}_{\alpha_1}$ and $\mathcal{S}_{\alpha_2}$. Thick edges are the $\mathcal{E}$ edge set of each network node S_i.

Algorithm 2: *DecomposeEPEG*$(\mathcal{G}, C_X)$

```
1  Let C_X = {C_1, C_2, ..., C_p};
2  N ← N ∪ (C_X, C_1, ..., C_p, ∅);
3  foreach C_i ∈ C_X do
4      if C_i ∈ N then continue;
5      Let C_i = {α_1, α_2, ..., α_x};
6      N ← N ∪ (C_i, S_α1, ..., S_αx, ∅), S_αj ⊆_e G;
7      foreach α_j ∈ C_i do
8          Let S_αj = (N_S, C_S, E_S, λ_S, R_S);
9          if ĈS ≠ ∅ then
10             Get the PEG G = S_αj − ĈS;
11             N ← N ∪ (S_αj, G, ĈS, E), S_αj = ĈS ∪_E G;
12             Decompose(G);
13             DecomposeEPEG(G, ĈS);
14         else
15             Get the PEG G = S_αj;
16             Decompose(G);
```

m_i is the number of parts in which each *network node* S_i is decomposed. Finally, $\mathcal{E}$ is the set of edges in S_i that connect the parts S_{ik} (with $k \in [1, m_i]$) together, denoted $S_i = \bigcup_{\mathcal{E}} S_{ik}$.

From the proposed algorithm, we can see that the network nodes S_i can be of different types. The algorithm starts with the cluster set of $\mathcal{G}_y$, marked as return cluster (R_y). This cluster set, called C_X within *DecomposeEPEG*, represents a *multi-cluster* network node, which is decomposed into its constituting single-clusters C_i (line 2). Then each of these clusters constitutes a *single-cluster* network node, that is decomposed into the e-subgraphs derived from every node belonging to each cluster (line 6). As previously annotated, an e-subgraph can contain complex clusters. If that is the case (line 9), we decompose the *mixed* network node into two parts (line 11), one containing only the set of complex clusters, and the other containing the remaining graph G, which constitutes a PEG since all clusters contain only one single node. The PEG G can be decomposed into *basic* network nodes, using the routine called *Decompose*, proposed in the DSI original algorithm (line 12). This is also the case when the e-subgraph does not

have any complex cluster (line 16). Finally, the complex cluster set of the mixed network node is recursively decomposed by our algorithm *DecomposeEPEG* (line 13).

The routine *Decompose*, as proposed in the DSI algorithm, recursively decomposes G into two smaller subgraphs (i.e., $\mathcal{N} \leftarrow \mathcal{N} \cup (G, G', G'', \mathcal{E})$, where $\mathcal{E}$ is the set of edges in G between G' and G''). The process continues with the derived components until individual nodes are reached. This routine also looks within the already existing basic network nodes for the maximun common subgraph to G, and uses it as criterion to perform the graph partition, so that, if a subgraph occurs multiple times in one model or in multiple models, it is represented only once. In the same way, if a particular node label appears multiple times, the decomposition represents this node only once at the bottom of the network. According to [12], this property not only leads to a compact representation of the model set, but is also the key to an efficient matching procedure at detection time. Figure 7 shows the resulting network during the first iteration of *DecomposeEPEG*, when using the input EPEG in Figure 2.

Notice, this simplified algorithm handles loop-free EPEG. For considering loops, we extend the $\mathcal{R}_{\mathcal{G}}$ function to check if a visited complex cluster is being decomposed. If so, this cluster is decomposed in its single nodes and not in its e-subgraphs. This avoids cycles in the decomposition process.

Algorithm 3: *Detection*$(\mathcal{N}, G_I)$

```
1  Let G_I = (N_I, E_I, λ_I, R_I);
2  foreach S_i = (N, E, λ, R) ∈ S_B with |N| = 1 do
3      F_{S_i} = VertexTest(β, λ(β), G_I) where {β} = N;
4      UpdateStatus(S_i, F_s);
5  while ∃S_i ∈ S_B : S_i.status = unsolved do
6      foreach S_x ∈ S do
7          Let N_X = (S_x, S_x1, ..., S_xm_x, E) ∈ N;
8          if S_x.status ≠ unsolved then continue;
9          if S_xk.status = alive, ∀k ∈ [1, m_x] then
10             F_{S_x} = CombineBasic(N_X, G_I);
11         UpdateStatus(S_x, F_s);
12 while ∃S_i ∈ (S − S_B) : S_i.status = unsolved do
13     foreach S_x ∈ S do
14         Let N_X = (S_x, S_x1, ..., S_xm_x, E) ∈ N;
15         if S_x.status = unsolved then continue;
16         if S_xk.status ≠ unsolved, ∀k ∈ [1, m_x] then
17             if S_x ∈ S_S ∧ ∃S_xk : S_xk.status = alive then
18                 F_{S_x} = CombineSingleCluster(N_X, G_I);
19             if S_x ∈ S_C ∧ S_xk.status = alive, ∀k then
20                 F_{S_x} = CombineMultiCluster(N_X, G_I);
21             if S_x ∈ S_M ∧ S_xk.status = alive, ∀k then
22                 F_{S_x} = CombineMixed(N_X, G_I);
23         UpdateStatus(S_x, F_s);
```

C. PEG Detection

For the detection process, we convert each function of the input program to its PEG representation to serve as *input graphs*, that can be analyzed either sequentially or in parallel. The procedure first searches for all occurrences of single nodes of the model graphs in the input graph, and stores them as a set of mappings in the corresponding network node at the network bottom-line. Then, it gradually combines them into larger mappings according to the composition described by the network until it reaches the level of a complete concept instance. To combine the mappings of two subgraphs, the algorithm checks not only that the mappings are disjoint, but also that the edges joining the two subgraphs within the parent network node (i.e, $\mathcal{E}$) exist in the input graph.

For a more detailed description, let S be the set of all network nodes occurring in the network $\mathcal{N}$, i.e., $S = \bigcup_{i=1}^{i=n} \{S_i\}$. Every network node has a *status* property that is initialized as *unsolved*. After a network node is processed, its status can be either *dead*, i.e., no mappings were found for that network node, or *alive* otherwise. Let S_C, S_S, S_M, S_B be the network node set of each type: multi-cluster, single-cluster, mixed and basic respectively. To detect a subgraph isomorphism of a concept version to an input graph, one needs to find a mapping function f that associates the matching nodes pairs of each graph. Two nodes match, whenever their labels are equal, and their children are also matching pairs.

Algorithm 3 displays the subgraph isomorphism detection algorithm of concept instances represented by $\mathcal{N}$ to the input PEG G_I. The algorithm consists of three main steps, identified by the three main loops. In the first step (line 2–4), we iterate over the basic network nodes S_i containing a graph with one single node $N = \{\beta\}$, and execute the following two routines:

- *VertexTest:* it iterates over all nodes α_I of the input graph and returns the mappings:
$$F_S = \{f(\beta) = \alpha_I : \lambda_I(\alpha_I) = \lambda(\beta)\}.$$
- *UpdateStatus:* this procedure updates the status of a network node to *dead* if $F_S = \emptyset$, and to *alive* otherwise.

Then in the second step (line 5–11), it processes all remaininig basic network nodes. For those network nodes S_x whose components S_{xk} are **all** *alive*, we call:

- *CombineBasic:* it combines the subgraph isomorphism of the network nodes S_{xk} whenever: *i)* the set of edges $\mathcal{E} : S_i = \bigcup_{\mathcal{E}} S_{ik}$ also exists in the input graph and *ii)* the mappings of the children network nodes are disjoint.

Finally, the third step (line 12–23) iterates over the network nodes generated by the *EPEGDecompose* routine, and combines the mappings of their components according to the network node type, by calling *CombineSingleCluster*, *CombineMultiCluster* and *CombineMixed*. Notice that, for combining mappings in a single-cluster network node, at least one of the components S_{xk} is *alive* (line 17). We require only one of the e-subgraphs derived from a node in a single complex cluster for the detection of a concept instance. Therefore, the combination of mappings in this type of network node consists of the *union* of the component mappings into a single set. The multi-cluster combining procedure differs from the *CombineBasic* by the fact that the joining edge set is $\mathcal{E} = \emptyset$. Therefore, we only check that the children mapping sets are disjoint. And finally, the mixed combining procedure has the peculiarity that

for checking the joining edges we require only the mapping associated to the node belonging to the single complex cluster (represented by the single-cluster network node), and not the complete mapping set associated representing an alternative component to build an implicit PEG. At the end of this process, if there is a multi-cluster network node alive that originally represented the return cluster set R_y of a model EPEG $\mathcal{G}_y$, then we have found a concept instance in the input graph.

D. Performance Analysis

Subgraph isomorphism detection is NP-complete, so the detection time is exponential in the worst case. In [12], it was shown that the computation time of their proposed decomposition-based approach is also affected by parameters other than the *number of nodes*. The worst case arises when all nodes have the same label and each node is connected to each other, thus the *number of labels and edges* also affects the detection time.

To determine how these parameters affect our algorithm, we performed few practical experiments with parameterized randomly generated model and input graphs. The parameters for generating the EPEG model graphs are:

- $|C|$: total number of clusters in the graph,
- $|\widehat{C}|/|C|$: ratio of complex clusters to total clusters,
- $|C_i|$: number of nodes per complex cluster,
- $|L|$: number of unique labels.

The first three of these parameters define the total *number of nodes* in the model graph. By specifying them separately, we can simulate the various aspects of the equality saturation process, since every application of a rule can modify each of these three parameters. Since the *number of outgoing edges* of an EPEG node depends on its label, for the sake of these experiments we collected a database of possible labels and their corresponding number of edges from EPEGs of some computational kernels of the *C66x DSP Library* [3] (same kernels used for the case study in Section VII). The average outgoing degree of these labels was determined to be 0.56, with 4 as the maximum degree. When generating a graph, a specified *number of labels* $|L|$ is randomly chosen from this database, and each node is associated with one of these labels. Then, the respective amount of edges are added to the graph, and two random clusters are chosen as the return clusters.

We study the behavior of the proposed algorithm by varying the graph generation parameters. Table II summarizes the different parameter values used in each specific experiment for the concept model generation. The constant parameters correspond to the average value observed in the concept model set used in the case study of Section VII. As the graphs are generated randomly, we perform each experiment 20 times and evaluate the average runtimes. In the first three experiments, we randomly derive one of the implicit PEG from the model graph as input graph, to ensure that a match is found.

In the first experiment, we compare the detection algorithm runtime when given a single model graph and a single input graph while varying two parameters: the number of clusters in the original EPEG (10-100), and the ratio of complex clusters

TABLE II
EXPERIMENTS AND THE MODEL GRAPH GENERATION PARAMETERS

No.	$\|C\|$	$\|L\|$	$\|\widehat{C}\|/\|C\|$	$\|C_i\|$	# Models
1	10-100	20	10%-100%	2, 2-3, 5	1
2	30	20	20%	2-3	1-15
3	30	3-30	20%	2-3	1
4	30	20	20%	2-3	1

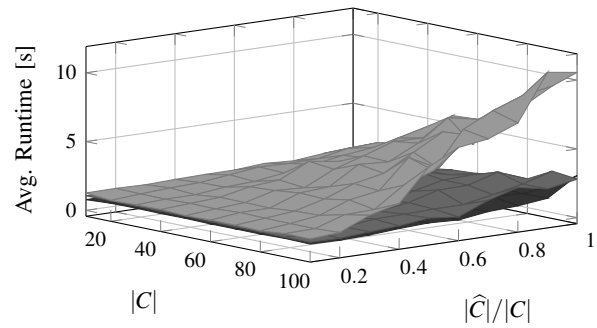

Fig. 8. Experiment 1: average runtime for varying graph sizes and the relative number of complex clusters. From darkest to lightest, the planes correspond to 2, 2 or 3, and 5 nodes per complex cluster, respectively.

to the total number of clusters (10-100%). The results are shown in Figure 8. The experiment was performed for different number of nodes in a complex cluster: first 2, then—randomly chosen—2 or 3, and finally 5 nodes per cluster. As it can be seen, the runtime increases for both larger numbers of total clusters and larger numbers of complex clusters. Furthermore, the algorithm performs worse if the complex clusters contain more nodes, especially for a high amount of complex clusters. In the detection algorithm, the submatches associated with a complex cluster are the union of the submatches of the nodes in the cluster, so if a larger percentage of the clusters are complex, the number of submatches to be processed per network node generally rises. Also, increasing this ratio as well as the number of nodes per complex clusters results in a larger number of total nodes in the graphs, explaining the increasing runtimes.

In the second experiment, we examine the effect of the model database size on the algorithm performance. For that, we change the database size from 1 to 15 model graphs, all generated under the same parameters. The input graph is a random implicit PEG derived from a random model. In Figure 9, the results of this experiment can be seen. Clearly, the runtime increases for larger model databases, but the dependency is less than linear. This can be easily explained by the fact that our algorithm is based on the decomposition-based approach presented in [12]. As it was shown there, the runtime of the detection is sublinearly dependent on the model database size for practical cases.

For experiment 3, we vary the number of unique labels (3-30) that can occur in the graphs. Figure 10 shows the outcome of this experiment. Our first expectation was a rising runtime for a smaller number of labels, similar to the experimental

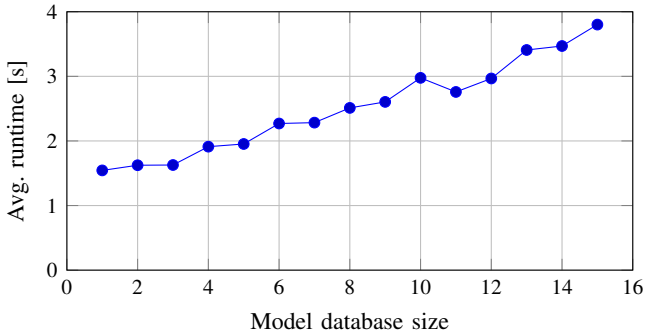

Fig. 9. Experiment 2: average runtime vs. model database size

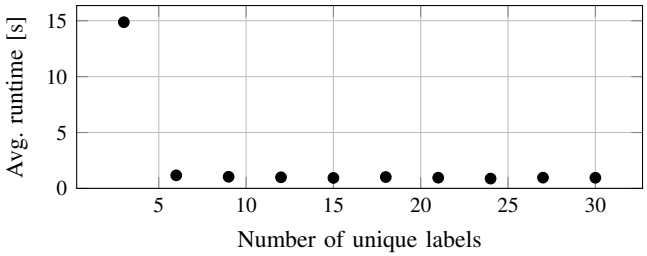

Fig. 10. Experiment 3: average runtime vs. number of unique labels

results in [12]. They varied the number of label graphs (4-40) in graphs with 50 nodes and 60 edges, and showed a steadily increase for less than 10 labels. According to [12], a few labels means that the likelihood of repeating substructures increases and the number of matches that are found for small subgraphs of the model graphs is thus usually very large. In our experiment, aside from the first data point, the average runtime tends to be constant. This is a result of the outgoing degrees of the nodes in our graphs generally being comparatively low. In [12], the average degree of each vertex was kept at 2.5. In our experiment the average degree depends on the specific label set chosen in each case. For our outlier case, the 3 labels chosen had an outgoing degree of 4, 2, and 0, respectively, resulting in a large number of edges which then leads to the big spike in computation time. The theoretical worst case of the algorithm is when all nodes have the same label and are connected to all other nodes, which leads to an exponential increase in possible matches. Here, we limit the amount of edges in the graph by our database of labels and associated outgoing degrees (which is the case for practical EPEGs), making the average outgoing degree of a vertex generally lower than the total number of nodes and prohibiting the exponential growth in matches.

Until now, all previous experiments actually deal with the problem of graph isomorphism, as the input graph is always directly derived from the EPEG model. In our fourth experiment, we run the detection algorithm for input graphs with more nodes than the model graph, thereby changing the problem to subgraph isomorphism detection. After deriving a possible match from the model graph, we add random nodes and edges to the graph until the input graph contained a

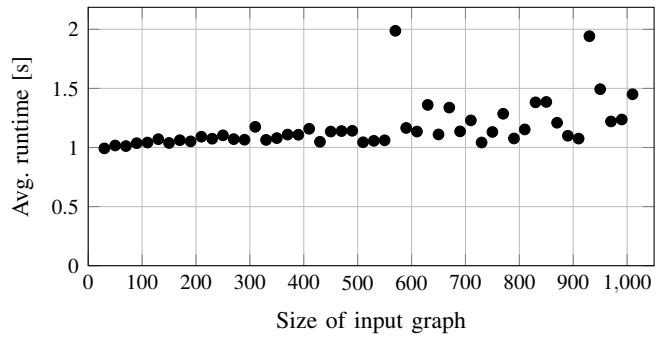

Fig. 11. Experiment 4: average runtime vs. input graph size

given number of nodes. We then examined the runtime of our algorithm for varying this input graph size. The results can be seen in Figure 11. Overall, the runtime increases slightly for larger input graphs. We also note that for input graph sizes larger than 500, the variance of our data points seems to increase. Two outliers at 570 and 930, both corresponding to an average runtime of 2 seconds, are the most noticable.

We conclude that the parameters that influence the runtime of our algorithm the most are the number of clusters in the model graph, the ratio of complex clusters, and the number of nodes they contain. The computation time is only sublinearly dependent on the model database size, and mostly independent of the number of unique labels. Furthermore, we see that our algorithm performs reasonably well for the problem sizes of our use case, despite its theoretically exponential complexity.

VII. Case Study

In this section we show the evaluation results of our approach when used as a support for porting sequential legacy C code to a given target system. Unlike the previous evaluation that characterizes the runtime performance with synthetic EPEGs, the objective is to assess the recognition efficiency in a practical scenario, using EPEGs derived directly from a set of computational kernels of a commercial platform library.

A. Experimental Environment

For this experiment, we use the *C66x DSP Library* from Texas Instruments [3]. The library contains a collection of optimized C-callable routines used in computationally intensive real-time applications, which have been tuned for the DSP processor family C66x. Each kernel includes a *natural C* version, an implementation without optimization, and an optimized C version. This is convenient, since the natural C versions readily serve as reference for the sample implementations. Table III shows the kernel selection constituting our algorithmic concept set. We use the UTDSP [13] and StreamIT [14] benchmark suites for the evaluation. Both are collections of applications typically executed on DSPs in commercial products. In particular, the UTDSP suite provides several versions written in different coding styles (using either array or pointer notation). The target system is the KeyStone I platform [2] that has eight C6678 DSP cores running at 1 GHz.

TABLE III
SET OF ALGORITHMIC CONCEPTS

ID	Concept	DSP Routine
DOTP	Dot Product	DSPF_sp_dotprod
MTRAN	Matrix Transpose	DSPF_sp_mat_trans
VMIN	Vector Minimal Value	DSPF_sp_minval
VMAX	Vector Maximal Value	DSPF_sp_maxval
VADD	Vector Addition	DSPF_sp_vecadd
VMUL	Vector Multiplication	DSPF_sp_vecmul
MMUL	Matrix Multiplication	DSPF_sp_mat_mul
FIR	Impulse Response Filter	DSPF_sp_fir_gen
AUTOC	Autocorrelation	DSPF_sp_autocor
WSUM	Weighted Summation	DSPF_sp_w_vec

TABLE IV
HISTOGRAM OF FUNCTION RUNTIMES

Runtime [mins]	< 0.5	$0.5 - 1$	$1 - 10$	$10 - 15$	$15 - 60$
Number of Functions	670	19	36	1	0

TABLE V
RECOGNITION RESULTS

Application	Runtime (seconds)	Manual	Tool	Local Speedup
compress	47	MMUL (2x)	✓(1x)	31.06
edge_detect	150	DOTP	✓	N/A
lpc	732	VMUL	✓	5.73
		DOTP	✓	4.00
		FIR (2x)		
		AUTOC (3x)		
spectral	280	VMUL		
audiobeam	1282	VMIN	✓	N/A
		VMAX	✓	N/A
fm	114	DOTP	✓	10.53
dct_ieee	133	DOTP(4x)		
fir	25	FIR		
matrixmult	29	MMUL		
matmul-block	110	MTRAN	✓	34.69
		DOTP	✓	4.35
nokia	143	MMUL		
		DOTP	✓	1.06

The performance is measured using the C6678 Device Cycle Approximate Simulator provided by Texas Instruments.

B. Results and Analysis

We run our recognition tool on 27 applications and a total of 726 functions. A histogram of the per-function runtimes is given in Table IV. We observe that for more than 90% of the functions, the tool completes in less than 30 seconds. The median tool runtime per function is 10.58 seconds.

The recognition results are summarized in Table V. The *runtime* column shows the per-application tool runtime, which is the sum of the per-function runtimes. The *manual* column lists the concept instances identified in each application by looking manually for code parts that could potentially be replaced with a library routine call. For example, in the application *compress* we found two code segments that implement a matrix multiplication concept. Note that we exclude applications from the table where no algorithmic concept was manually found. Our tool successfully recognizes the instances marked with "✓" in the *tool* column.

The tool reported no false positives, also in the applications not listed in the table. Except for the concept instances found in `audiobeam`, all differ from their respective concept model due to one or more variations. The UTDSP applications, where a recognition was successful, were tested for both pointer and array based version. We analyze the cases where the automatic recognition is not successful with respect to the manual one. Our findings are summarized according to known challenges for concept recognition [1]:

Data Structure Variations: The library kernel and the input program act on different data layouts. These variations appear in three cases: `FIR`s in *lpc*, `MMUL` in *matrixmult*, and `DOTP`s in *dct_ieee*. In the first case, the `FIR` library kernel assumes that its input vector (`h`) of length `nh` should be given in reverse order i.e., `{h[nh-1], h[nh-2],..., h[0]}`. In the second case, the matrices used as input and output in *matrixmult* are implemented using columns as first and rows as second dimension of a two-dimensional array, i.e., `A[columns][rows]`. The inverse order is assumed by the kernel. In the case of the dot products in *dct_ieee*, one of the input vectors is the first dimension of a two-dimensional array, i.e., `sum += A[i][n] * B[i]`, where `i` is the loop induction variable. Even if, from the algorithmic point of view, this is still a dot product, the kernel assumes that the vector elements are contiguous in memory, which is not the case for the implementation in the application. The direct consequence of these variations is the way each concept instance performs memory accesses.

Delocalization: The logically related elements of the concept instance are not consecutive in the code. This is the case in *spectral*. In the kernel, the execution of the vector multiplication is performed contiguously without any other statements in between iterations. In the application, the loop body contains other statements, creating false dependencies. Other frameworks (e.g. [6]) have circumvented this problem by applying loop distribution prior to the recognition.

Optimization: During the implementation, additional knowledge about the input can lead to variations (optimizations) of the algorithm. This is the case for `AUTOC`s in *lpc*. It implicitly assumes that the input signal is padded with zeros. The code segments' loop boundaries are determined to exclude the multiplications with those zeros, which is not the case for the more generic library kernel.

For each detected kernel we manually replaced the identified code segment with the optimized routine. For the successful cases in Table V, the last column shows the *local* speedup obtained per kernel considering only the execution time of the transformed code segment. Those concept instances, where the substitution is not possible, are marked with N/A. As expected, the replacements result in a significant *local* perfor-

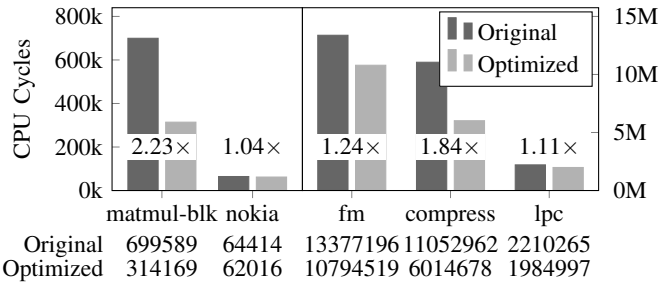

Fig. 12. Original and optimized performance for the applications where the replacement was successful

mance improvement. The *global* speedup of the application performance is analyzed at the end of this section. We could not insert the library kernel at all detected concept instances. The main reason is that optimized routines usually have a set of restrictions. For example, the dot product expects the input vector's length to be a multiple of eight.

Finally, Figure 12 shows the *global* per-application speedups. We report the CPU cycles for the original and optimized versions of the applications. Although most of the local speedups in Table V are significant, not all the replaced code segments reside in a critical section. Therefore, we obtain significant global speedups for some replacements like in the case of *matmul-block* and *compress*, while in other cases, the speedups are small such as for *nokia*.

C. Discussion

Using C for the computational kernel specification is an advantage. In rule-based approaches (e.g., [6], [7]), the user needs to derive new *if-then* rules, typically written in terms of the IR used by the detection system [4]. In our case, no special knowledge on the internals of the system is needed. Also no special code style is expected from the sample implementations. Still to write the sample implementations requires an effort. The developer of optimized library routines (platform vendor) requires a golden reference to test for functional correctness. Therefore, in principle, kernel functional specifications already exists. All we need is that the vendors shares them, as done by Texas Instruments. Nevertheless, to migrate to a different platform in a similar domain is also a less-demanding task, since algorithmic kernels can overlap. Only tuning to the specific routine interface might be required.

The per-function level detection analysis is a well-defined task that can be parallelized, allowing the tool to scale well for big programs. For example, *mpeg2* consisting of 201 functions, sequentially analyzed, takes 70 mins. But the maximum per-function detection time is only 7.35 minutes. However, the per-function analysis has the limitation that a particular program structure can prohibit a successful recognition. Inlining can be applied to overcome this limitation at the cost of bigger function sizes, potentially impacting also the dectection time. An additional optimization to the tool flow would be to pre-process the input program with a profiler, so that we can focus the analysis only on the hotspots, which can also allow a guided application of inlining.

VIII. Conclusions

This paper presented a computational kernel recognition mechanism as a novel way to assist the task of porting C-code to an embedded target platform. The first phase generates a concept model based on an equality reasoning process, where the applied rules have been tuned to handle common source code variations for our use case. Then the recognition system has been implemented by means of a decomposition-based subgraph isomorphism detection algorithm. The main conclusions can be summarized as follows. Most source code variations are implicitly handled due to our intermediate representation choice. Implicit constraints or assumptions on the input data are only specified in the library kernel's documentation, thus it requires a second validation step. Data layout differences between the benchmark and the library prevent the recognition of instances as the same algorithmic kernel. Identified code segments are not necessarily hot spots, but for our case study, most of the kernels were found in critical sections. We plan to investigate how to overcome delocalization in future work, which will allow the recognition of corner cases, since the common case is that a meaningful algorithm is normally placed contiguous in code. Also, our approach could, for example, be extended to support different optimization strategies, like code parallelization.

References

[1] L. M. Wills, "Automated program recognition by graph parsing," DTIC Document, Tech. Rep., 1992.

[2] "SPRS691E:multicore fixed and floating-point digital signal processor TMS320C6678," [Online] http://www.ti.com/lit/ds/symlink/tms320c6678.pdf (accessed 11/2015).

[3] "SPRUEB8B:TMS320C6xx+ DSP little-endian DSP library programmers reference," [Online] http://www.ti.com/lit/ug/sprueb8b/sprueb8b.pdf (accessed 11/2015).

[4] R. C. Metzger and Z. Wen, *Automatic algorithm recognition and replacement: a new approach to program optimization*. MIT Press Cambridge, 2000.

[5] M. Arenaz, J. Touriño, and R. Doallo, "XARK: An extensible framework for automatic recognition of computational kernels," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 6, pp. 32:1–32:56, Oct. 2008.

[6] A. Shafiee Sarvestani, E. Hansson, and C. Kessler, "Extensible recognition of algorithmic patterns in DSP programs for automatic parallelization," *Int. J. Parallel Program.*, vol. 41, no. 6, pp. 806–824, Dec. 2013.

[7] U. Vinod and P. K. Baruah, "MPIIMGEN—a code transformer that parallelizes image processing codes to run on a cluster of workstations," in *Int. Conf. Cluster Computing*. IEEE, 2004.

[8] B. D. Martino, "ALCOR — an algorithmic concept recognition tool to support high level parallel program development," in *Proc. PARA*. Springer-Verlag, 2002, pp. 150–159.

[9] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality saturation: A new approach to optimization," *SIGPLAN Not.*, vol. 44, no. 1, pp. 264–276, Jan. 2009.

[10] "Peggy: A system for equality saturation," [Online] http://goto.ucsd.edu/~mstepp/peggy/ (accessed 03/2016).

[11] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. CGO*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–.

[12] B. T. Messmer and H. Bunke, "Efficient subgraph isomorphism detection: A decomposition approach," *IEEE Trans. on Knowl. and Data Eng.*, vol. 12, no. 2, pp. 307–323, Mar. 2000.

[13] C. Lee, "UTDSP benchmark suite," [Online] http://www.eecg.toronto.edu/corinna/DSP/infrastructure (accessed 08/2015).

[14] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Compiler Construction*. Springer, 2002, pp. 179–196.