On Domain-Specific Languages Reengineering

Christophe Alias and Denis Barthou

Laboratoire PR*i*SM, Université de Versailles, France.
Christophe.Alias@prism.uvsq.fr
Denis.Barthou@prism.uvsq.fr

Abstract. Domain-specific languages (DSL) provides high-level functions making applications easier to write, and to maintain. Unfortunately, many applications are written from scratch and poorly documented, which make them hard to maintain. An ideal solution should be to rewrite them in a appropriate DSL. In this paper, we present `TeMa` (`Te`mplate `Ma`tcher), an automatic tool to recognize high-level functions in source code. Preliminary results show how `TeMa` can be used to reformulate Fortran code into Signal Processing Language (SPL) used in SPIRAL. This opens new possibilities for domain-specific languages.

1 Introduction

With domain-specific languages, algorithms are described in a more abstract and compact way than with traditional imperative programming languages. Expressing algorithms at a higher level of abstraction adds portability and improves programmer productivity for code writing and maintenance. Moreover, the higher the representation of the program, the more aggressive the compiler optimizations can be: for instance a generative approach can yield from a mathematical formula in SPIRAL [16] a finely tuned code for a particular architecture, exploring both algorithmic variations and traditional code optimizations. The same benefit appears also for code verification.

However, this approach assumes that the user specifies his algorithm using a domain-specific language. Starting from an existing C or Fortran program and finding in it fragments that correspond to domain specific templates would be desirable but seems difficult to obtain. The reason is that it amounts to algorithm recognition, an old problem in computer science. Basically, one would like a compiler or analyzer to automatically find that lines 10 to 23 are an implementation of a DFT for instance. A natural solution would be to search the code for patterns of known functions (matrix-matrix product, tensor product or direct sum for the DFT). Such a facility would enable many important techniques:

- Program comprehension and reverse engineering: we can rewrite part of the code with a higher level language, enhancing code maintenance and portability.
- Program optimization: if we have the necessary items in our library or if we can use a generative approach, we may replace lines 10 to 23 by an

automatically tuned version. We may even replace the relevant part of the code by a completely different implementation.
- Program verification: if we know that the program specification asks for a DFT and the analyzer does not find it, we may suspect an error.
- Hardware-software co-design: if we recognize in the source program a piece of code for which we have a hardware implementation (e.g. as a co-processor or an Intellectual Property component) we can remove the code and replace it by an activation of the hardware.

The pattern could be a naive implementation of the function but obviously the approach is interesting only if the detection abstracts away some variations between the code fragment and the reference implementation. Program variations can arise from data structure variations (coming from scalar promotion, array expansion, structures instead of arrays,...), control variations (coming from loop fusion, unroll, skewing,...), organization variations (coming for permutation of program statements) or semantic variations (using associativity or commutativity for instance). Obviously, the abstraction obtained depend on the range of variations handled by the detection.

In this paper, we present an approach to automatically find, in linear time, all possible instances of a given template in a program. Implemented in the `TeMa` tool and connected with a more expressive method already described in [1], the method is able to find the parameters of the template corresponding to a particular instance. It handles many implementation variations and extends previous works by handling variations concerning data structures using arrays. Experimental results on SPEC benchmarks show that our method is able to abstract away many variations. Finally, preliminary results show that this technique is able to reformulate Fortran code into Signal Processing Language (SPL) used in SPIRAL. This opens new possibilities for domain-specific languages.

The paper is organized as follows: Section 2 introduces the notations and definitions used in the paper. Section 3 describes the algorithmic content of `TeMa` and provides a classification of program variations handled. Finally, section 4 presents the `TeMa` tool and provides experimental results.

2 Background

SPL [21] (Signal Processing Language) is a domain-specific language for describing matrix factorizations, and thus fast algorithms for computing matrix-vector products. In particular, it can be used for describing fast signal transforms such as FFT or WHT. An SPL program is an expression involving a variety of operations including composition, direct sum and tensor product. Let us give an example of SPL program. Briefly recall that given two matrices A and B, the *tensor product* of A and B, is the matrix $A \otimes B = [a_{ij}B]$. The WHT (Walsh-Hadamard Transform) over a sampled signal of 2^n elements can be written $\mathrm{WHT}_{2^n} = F_2 \otimes \ldots \otimes F_2$ n times , where $F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ denotes the FFT trans-

form over 2-dimensionnal vectors. The SPL program computing the WHT over 2^3-dimensionnal input vectors can thus be written:

```
(tensor (F 2) (tensor (F 2) (F 2)))
```

In addition to improve readability and thus maintenance, recovering an SPL program from a C or Fortran program allows to benefit of the SPL compiler optimizations.

The approach investigated in this paper is to recognize within the program the slices corresponding to naive implementations of SPL functions, then to rebuild the SPL formula. The naive implementations to find are expressed with program patterns. A *pattern* is a schema of program with wildcards values and functions. For example, figure 1 gives the pattern of a reduction, where wild-cards are denoted by $\square$. One contribution of this paper is the algorithm to quickly find

```
s = □
do i = 1,n
| s = □(s,□)
enddo
return s
```

Fig. 1. Pattern of a reduction

all possible instances of a pattern within a program. In the `TeMa` tool presented thereafter, it is connected with a more expensive method already described in [1] to check if the program slices found are effectively instances of the pattern. In case of success, our exact method provides the corresponding values of $\square$.

Finding instances of a pattern in a program means finding all program slices *equivalent* to an instance of the pattern. But what does exactly means «equivalent» ? We will consider a weak version of semantic equivalence called *Herbrand-equivalence* and denoted by $\equiv_{\mathcal{H}}$. Instead of indicating whether two algorithms compute the same (mathematical) function, Herbrand-equivalence just indicates if they used the same mathematical formula, syntactically. In this way, Herbrand-equivalence can be considered as a true algorithmic equivalence. Even if Herbrand-equivalence is weaker than semantic equivalence, and seems to be easier to check, it has been unfortunately proven undecidable [2].

Our detection algorithm uses a powerful extension of automata called tree-automata. A *tree automaton* is a tuple $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$, where Σ is a signature, Q the set of states, $Q_f \subset Q$ the set of final states, and Δ a set of transition rules of the type $f(q_1 \dots q_n) \longrightarrow q$, where $n \geq 0$, $f \in \Sigma$ and $q, q_1, \dots, q_n \in Q$. Tree automata were introduced by Doner [5, 6] and Thatcher and Wright [17, 18] in the context of circuit verification. Most of usual operations on word automata (determinization, minimization, cartesian product, ...) extend naturally to tree automata [4].

3 Detecting SPL functions

In this section, we present our method to detect SPL functions in a given C or Fortran program. We also present our preliminary approach to build an SPL program from relevant program slices. Finally, we evaluate the detection capabilities of our method in terms of program variations handled.

3.1 Overview of the method

Figure 2 gives the main steps of our method. The slicing method search through the program the slices which *potentially* implement an SPL function. Then we check whether the slices are equivalent to a given SPL function by applying our exact instantiation test, already described in [1].

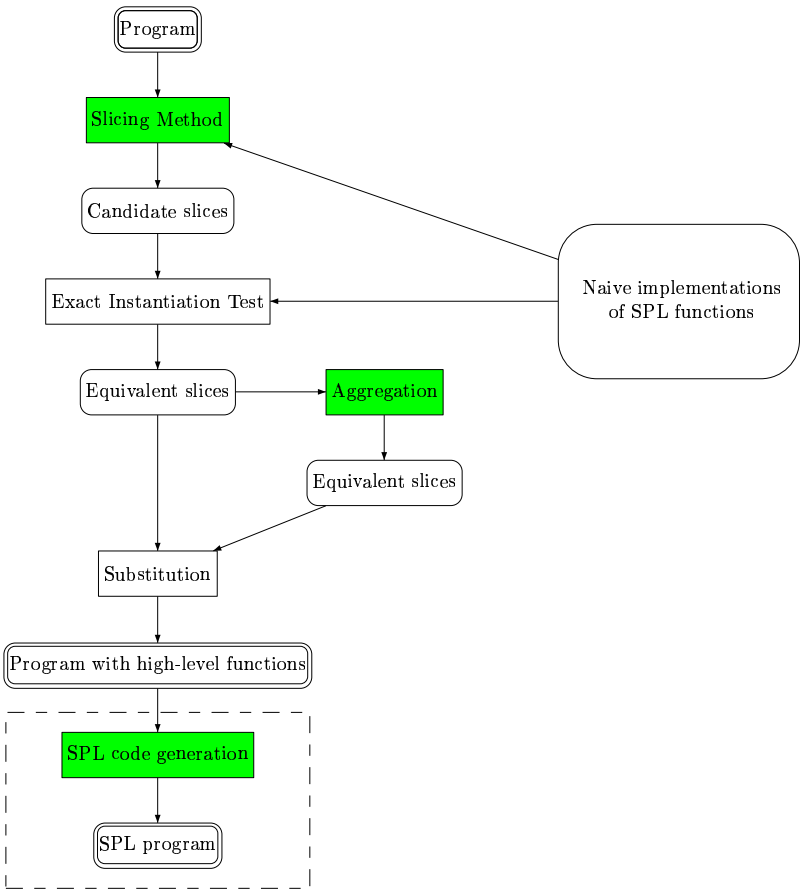

Fig. 2. Overview of the method

The aggregation allows to detect slices with multiple output statements, which is not allowed by previous steps. This is typically the case of matrix operations. Once implementations of SPL functions are found in program, it remains to substitute them by the relevant call to the SPL function (substitution). The program is then ready to be translated into an SPL program.

The contributions of this paper are the slicing method and the aggregation. We also propose a preliminary approach to recover SPL programs from relevant program slices. These important steps are described thereafter.

3.2 Slicing method

The aim of our detection algorithm is to provide the parts of the program which potentially compute the same arithmetic expression than an instance of the pattern. The main idea is to walk through the pattern and the program def-use chains as long as pattern and program operators are equal. If the method reaches the last statement of the pattern, all reached program statements will be yield as a candidate slice, meaning that they may be Herbrand-equivalent since the expressions computed may have the same sequence of operators.

Algorithm *Build_Automaton*

Input: *The pattern or the program.*
Output: *The corresponding tree automaton.*

1. Associate a new state to each assignment statement.
2. For each state:

$$q = \boxed{\texttt{r = } f(\phi(Q_1) \dots \phi(Q_n))}$$

   Add the transitions: $f(q_1 \dots q_n) \longrightarrow q$, for each $q_i \in Q_i$.
3. For each state:

$$q = \boxed{\texttt{r = } \Box(\phi(Q_1) \dots \phi(Q_n))}$$

   Add the transitions: $q_i \longrightarrow q$, for each $q_i \in Q_i$ *(input transitions)*.
   And: $f(q \dots q) \longrightarrow q$, for each operator f used in the pattern and the program, including constants (0-ary operators) *(looping transitions)*.

Fig. 3. *Build_Automaton*

The pattern and the program are assumed to be given in scalar SSA-form, a classical form in compilation that provides def-use chains. We first associate to the program and the pattern a tree automaton allowing to step them easily. This is done by using the algorithm described in figure 3. Basically, each state corresponds to a statement (step 1), and the transitions to a state are driven by def-use chains, and labeled by the statement operator (step 2). Pattern wildcards

are handled as Kleene star in word automata. Note that the wildcard value is a particular case of wildcard function $\square(\ldots)$ with arity 0. Step 3 of the method builds a loop for these states with any operator which appears in the program.

Consider the pattern and the program given in figure 4. For sake the of clarity, we have chosen a pattern and a program with unary operators which will lead to the word automata given in figure 5. But of course, our method can handle operators with any arity. The ϕ-functions can be seen as multiplexers selecting the last definition for a given value.

```
r1 = 1
do i = 1,n
| r2 = □(φ(r1,r3))
| r3 = 1+r2
enddo
r4 = exp(r3)
```

```
z1 = 1
t = 0
a = tan(t)
do i = 1,10
| z2 = 1/(φ(z1,z3))
| z3 = 1+z2
enddo
r = exp(z3)
```

Fig. 4. Pattern (left) and Program (right)

The idea is now to step simultaneously the two automata up to the pattern final state while the operators are equal. The two automata have as many entry points as constant leaves (1(), 2() here), and we have to start a comparison from each couple of leaves. The operations corresponds to the definition of the *cartesian product* of the pattern and program automata. The detected slices can then be computed by collecting all program states along the paths from initial states to each state with a final pattern state (q_{final},.). The detection step is summarized in the algorithm described in figure 6.

Let us summarize our algorithm. Given a pattern and a program we first compute their tree-automata by applying *Build_Automaton*. The slices of "good" candidates are then obtained by stepping simultaneously $\mathcal{A}_T$ and $\mathcal{A}_P$. This task is achieved by *Output_Slices*. We finally apply the exact equivalence test described in [1] to check whether the slices are instances of patterns or not.

3.3 Aggregation

Our method is able to detect template occurrences with only one output statement. We present thereafter an extension to detect slices with several outputs by using an aggregation hierarchy of domain-specific functions.

Motivating example The templates to match often use an array. Yet our method is able to detect the occurrences with only one output statement. Figure 7 provides an example of matching, where the template is the daxpy function of BLAS 1. Our slicing method yields the candidates slices S_1 and S_2, but the

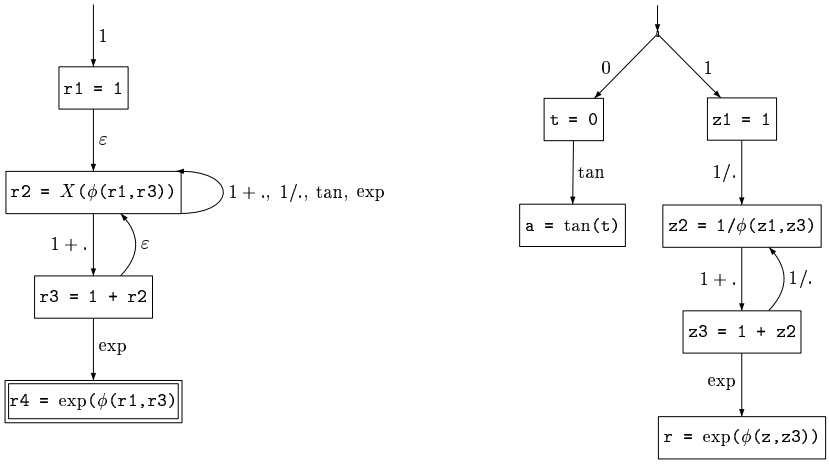

Fig. 5. Pattern automaton (left), and Program automaton (right)

Algorithm *Output_Slices*

Input: $\mathcal{A}_T$ *and* $\mathcal{A}_P$*, pattern and program automata.*
Output: $\{s_1 \ldots s_n\}$*, the last statements of each candidate slice.*

1. Compute the Cartesian product $\mathcal{A} = \mathcal{A}_T \times \mathcal{A}_P$.
2. Mark the nodes with a final state of $\mathcal{A}_T$, and emit the $\mathcal{A}_P$ part of marked states.
3. For each marked node q:
   Compute the set of previous states $\mathcal{S}lice(q) = \{q',\ q' \longrightarrow^* q\}$.
   Then return the $\mathcal{A}_P$ part of $\mathcal{S}lice(q)$.

Fig. 6. *Output_Slices*

candidate $S_1 \cup S_2$ where $a = 2$, $\boldsymbol{x} = [s(1), s(2), s(3), u]$ and $\boldsymbol{y} = [1, 1, 1, v]$ is missing since its outputs are shared by several statements.

Aggregation hierarchy One can remark that daxpy is constituted of 1-dimension daxpy instances. Another solution would be to detect «atomic» daxpy using the slicing method, then to *aggregate* them to make a larger daxpy.

In a more general manner, consider an algorithm A which produces an array, and a family of algorithms $(A_{\boldsymbol{i}})_{\boldsymbol{i}}$, where $A_{\boldsymbol{i}}$ outputs the $\boldsymbol{i}$-th array cell of A for each possible input:

$$A_{\boldsymbol{i}}(I) \equiv_{\mathcal{H}} A(I)[\boldsymbol{i}]$$

For each relevant input I and array index $\boldsymbol{i}$. Then A is said to be an *aggregation* of the $A_{\boldsymbol{i}}$.

```
do i = 1,n
  y(i) = a*x(i) + y(i)
enddo
return y
```

```
S1  do i = 1,3
S1    s(i) = 2*s(i) + 1
S1  enddo
S2  u = 2*u + v
```

Fig. 7. Two detections of daxpy

Aggregation induces a hierarchy between algorithms, and particularly between templates. Typically, a daxpy is an aggregation of several scalar daxpy , and a matrix-vector product is an aggregation of dot products. Figure 8 provides an aggregation hierarchy between some BLAS 1 and 2 functions. $A \rightarrow B$ means "B is an aggregation of A instances".

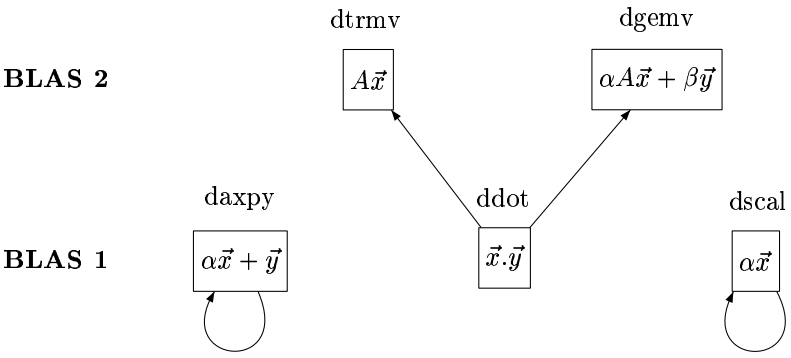

Fig. 8. Aggregation hierarchy of BLAS 1 and 2 functions

A solution is to detect the templates of the hierarchy by using the slicing method. Then we aggregate them in a bottom-up manner, from the leaves functions to the top functions. If A is an aggregation of $(A_i)_i$, all combinations of A_i instances are aggregated, and yielded as A instances. The aggregation is just a concatenation of slice outputs, as stated in the motivating example.

3.4 SPL code generation

Once the program is rewritten by using SPL functions, it remains to generate the corresponding SPL program. The preliminary approach investigated in this paper, but not yet implemented, is to select the program slices which can be completely unrolled, then to use the data-flow dependences to build the corresponding SPL program. The output of the SPL generation step is thus a set of unrollable program slices, and their corresponding SPL program. Unrolling is possible whenever the program slice uses `for` loops with bounds as expressions with constants and surrounding loop counters. The reaching definitions can

be easily computed on the unrolled program slice by using usual methods [15]. These restrictive conditions lead to select the slices which can be unrolled, then to translate them into an SPL program. We believe that this approach is able to recover SPL programs corresponding to relevant program slices.

3.5 Program variations detected

The efficiency of our approach directly depends on its capacity to recognize SPL functions in a source code. A common way to evaluate an algorithm recognition system is to provide the different kinds of pattern variations it can handle [20, 9, 13, 14]. We provide thereafter a detailed description of each variation. We also state whether our algorithm is able to detect them.

Organization variations Any permutation of independent statements and introduction of temporary variables. The following example provides an organization variation with legal permutations (LP), garbage code (GC) and temporaries (T):

```
s = a(0)
c = 0
do i = 1, n
  s = s + a(i)
  c = c + 1
enddo
return s + c
```

```
    s = a(0)
    c = 0
GC  garbage = 0
    do i = 1, n
LP    c = c + 1
T     temp = a(i)
      do j = 1, p
GC      garbage = garbage + 1
      enddo
      s = s + temp
GC    garbage = garbage + a(i)
    enddo
    OUTPUT = s + c
```

Our algorithm works on a def-use graph, which avoids the artificial precedence constraints due to the text representation of the program. This allows our algorithm to handle legal permutations and garbage code. Our method compares two by two the operators used in the template and the program without handling variables, this allows to handle temporaries.

Data structure variations The same computation with a different data structure. The following example gives a data structure variation with arrays and non-recursive structures:

```
s(0) = a(0)
do i = 1, 2*n
  s(i) = s(i-1) + a(i)
enddo
OUTPUT = s(2*n)
```

```
s.sum1 = a(0)
do i = 1, n
  s.sum1 = s.sum1 + a(i)
enddo
s.sum2 = a(n+1)
do i = n+2, 2*n
  s.sum2 = s.sum2 + a(i)
enddo
OUTPUT = s.sum1 + s.sum2
```

One of the important add-on of the paper is the ability of the detection to cope with different representation of arrays. Transformations such as scalar promotion (transforming an array section into as many scalars) or array expansion (the reverse) are handled thanks to the aggregation step.

Control variations Any control transformation as if-conversion, dead-code suppression and loop transformations as peeling, splitting, skewing, etc. The following example give a control variation with a simple peeling:

```
s = a(0)
do i = 1,n
  s = s + a(i)
enddo
OUTPUT = s
```

```
s = a(0)
s = s + a(1)
do i = 2,n-1
  s = s + a(i)
enddo
s = s + a(n)
OUTPUT = s
```

In a general manner, we are abe to hande any variation which does not affect the operators nest of the expression computed by the program.

Each of these variations provide an Herbrand-equivalent slice, which our algorithm is able to detect in a general way. But we are not able to detect non-Herbrand-equivalent variations, such as *semantics variations*, which uses semantics properties of operators such as associativity or commutativity. Nevertheless, experimental results given thereafter shows that our method finds a large amount of correct candidates.

4 Experimental results

In this section, we present TeMa, the implementation of our algorithm recognition system. We provide experimental results on SPEC benchmarking suite demonstrating the power of our slicing method. In addition, we show how TeMa can be used to recover an SPL program from a naive implementation of the Walsh-Hadamard transformation.

TeMa (Template Matcher) is the implementation of our algorithm recognition system, including the slicing method, the exact instantiation test, the aggregation method described in figure 2 and the substitution. For the moment, TeMa does not implements the SPL code generation. TeMa has been implemented in Objective Caml, and represents 10 kloc. TeMa is declined in two versions: a batch version for automatic usage such as benchmarking, or systematic discovery of patterns in a large application ; and an interactive version with a GUI which aims to be used in re-engineering, program comprehension or software maintenance. Our front-end is able to handle C and Fortran 90 programs. C front-end uses the LLVM compiler infrastructure [11], which is based on gcc front-end. Thus TeMa is able to handle any C real-life application. We have also implemented our own Fortran 90 front-end. Most Fortran 90 programs are correctly handled, but some syntactic constructions are not yet accepted, and need to be modified

by hand. Our front-end has handled with success all fortran programs of SPEC benchmarking suite.

We have applied our slicing method to detect potential calls to the BLAS library [12] in LINPACK [7] and four programs involved in the SPEC benchmarking suite [8]. Our pattern base is constituted of direct implementations of BLAS functions from the mathematical description. Figure 9 shows the results.

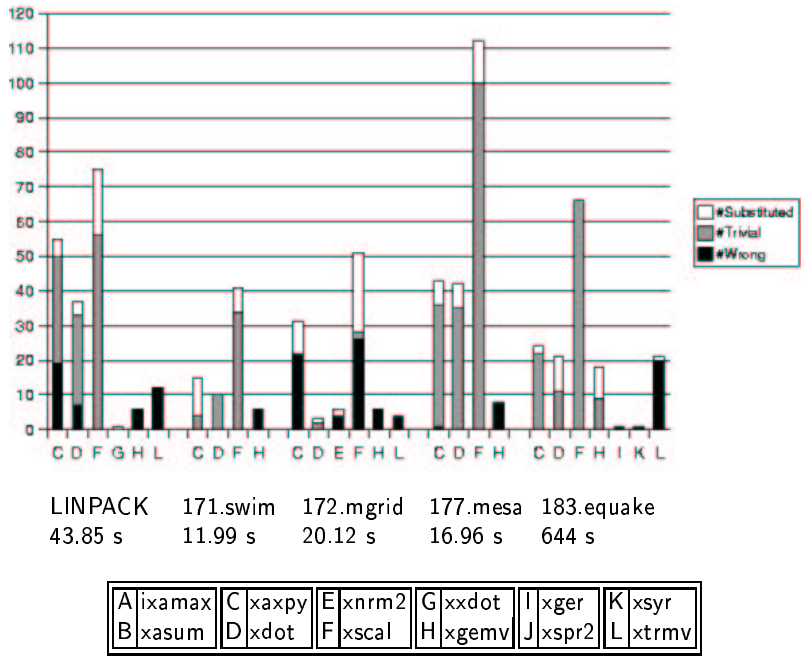

Fig. 9. For each SPEC program, we provide each BLAS function found, the number of non-equivalent slices (# Wrong), the number of equivalent slices with one statement (# Trivial), and the number of other equivalent slices (# Substituted). The execution times are given for a Pentium 4 1,8 GHz with 256 Mo RAM.

It appears that 50% of candidates do not match, 25% are instances of patterns with one-dimension vectors, and 25% of candidates are correct and can be replaced by a call to BLAS. We present the different kind of candidates involved in these categories. Most of the incorrect detections are due to the approximation of the dependences with ϕ-functions. Neither loop iteration count, nor `if` conditions, nor complex dependences due to array index functions are taken into account.

25% of the slice is constituted of interesting candidates whose substitution can potentially increase the program performance. Our algorithm seems to have discovered all of them, and particularly hidden candidates. Indeed, most slices found are interleaved with the source code, and deeply destructured. Our method

has been able to detect a dot product in presence of a splitting and a loop unroll, which constitute important program variations that a `grep` method would not catch. The same remark applies on `equake` program. Two versions of matrix-vector product appear, one hand optimized and the other not. Both are detected whereas a method based on regular expressions would detect only the second.

`TeMa` allows to recover SPL programs by recognizing and substituting SPL functions including matrix composition, direct sum and tensor product. Consider the following program, which is a naive implementation of the Walsh-Hadamard Transform (WHT):

```
      c(1) = f2
      do iter = 2,5
      |  rank = 2 ** (iter - 1)
⊗     |  do i = 0,1
⊗     |  |  do j = 0,1
⊗     |  |  |  do k = 0,rank-1
⊗     |  |  |  |  do l = 0,rank-1
⊗     |  |  |  |  |  c(iter,i*rank+k,j*rank+l) = f2(i,j)*c(iter-1,k,l)
⊗     |  |  |  |  enddo
⊗     |  |  |  enddo
⊗     |  |  enddo
⊗     |  enddo
      enddo
      wht = c(5)
```

`TeMa` detects the slice marked by ⊗ as a tensor product between `f2` and `c(iter - 1)`, and substitutes it in the following manner:

```
      c(1) = f2
      do iter = 2,5
      |  rank = 2 ** (iter - 1)
      |  c(iter) = f2 ⊗ c(iter-1)
      enddo
      wht = c(5)
```

Applying by hand our preliminary SPL code generation method, we finally obtain the following SPL program:

```
(tensor (F 2) (tensor (F 2) (tensor (F 2) (tensor (F 2) (F 2)))))
```

Even if the SPL generation is not yet implemented, the rewriting of the program with high-level functions increases readability, making `TeMa` a promising tool to improve program comprehension and help programmers in the tedious task of software maintenance.

5 Related work

We first present related work about program slicing as a tool to help software maintenance, then we present some methods for pattern detection, and more specifically *algorithm recognition*.

Program slicing was first introduced by Mark Weiser [19], to help programmers to debug their code. He defined a *slicing criterion* as a pair (p, V), where p is a program point and V a subset of program variables. A program slice on the slicing criterion (p, V) is a subset of program statements that preserves the behavior of the original program at the program point p with respect to the program variables in V. Weiser has shown that computing the minimal subset of statements which satisfies this requirement is *undecidable* [19]. However an approximation can be found by computing consecutive sets of indirectly relevant statements, according to data-flow and control-flow dependences.

Cimetile et al. [3] defined a method to identify slices verifying given pre-conditions and post-conditions. They first compute a symbolic execution of the program, which assign to each statement its pre-condition, then they use a theorem prover to extract the slices. They need user interaction to associate post-condition variables to program variables. Moreover, as the problem of finding invariant assertions is in general *undecidable*, symbolic execution can require user interaction in order to prove some assertions and assert some invariants. No practical evaluation of their method, or theoretic study of complexity is given, but their method seems to be costly. Moreover, the need of user interaction makes the method inappropriate in a fully automatic framework.

Several approaches encode the knowledge about the functions to be identified in the form of programming plans, and can be classified as either top-down or bottom-up methods. Top-down methods [9, 10] use the knowledge about the goals the program is assumed to achieve and some heuristics to locate both the program slice and the plan from the library which can achieve these goals. Bottom-up methods [13, 20] start from the program statements and try to find the corresponding plans. Wills [20] represents programs by a particular kind of dependence graph called *flow-graph*, and patterns by *flow-graph grammar* rules. The recognition is performed by parsing the program's graph according to the grammar rules. She finally obtain a *parsing tree* which represents a hierarchical description of a plausible project of the program. This approach is a pure bottom-up code-driven analysis based on exact graph matching. Patterns are represented by grammars rules, encoding a hierarchy among them, but making the pattern base difficult to maintain. Organization variation is partially supported and temporary variables can be handled by adding specific rules. All others algorithmic variations can be handled only if they are explicitly described in the pattern base.

Metzger and Wen [14] have built a complete environment to recognize and replace algorithms. They first normalize the program and pattern AST by applying classical program transformations (if-conversion, loop-splitting, scalar expansion...). Then they look for good candidate slices within the program. The candidate slices are SCCs of the dependence graph, containing at least one `for`

statement. Their equivalence test is based on an isomorphism between the slice and pattern AST. Obviously, this approach is low cost, and scalable. One may point out the large amount of candidate slices given by their method, but it is not a real problem due to the low complexity of their equivalence test. Organization variations, resulting from the permutation of independent statements or the introduction of temporaries are not handled by the algorithm itself, but by pre-treatments applied to the program. Reuse of temporaries accross loop iterations for instance is not handled. In the same way, the control variations supported are bounded to pre-treatments.

6 Conclusion

In this paper, we have presented an automatic method to find high-level functions in a given program, and its implementation in the tool `TeMa`. We have also proposed a preliminary approach to reformulate Fortran or C code into Signal Processing Language (SPL). Our detection method has been validated by recognizing BLAS functions in different kernels of the SPEC benchmarking suite. Our method is able to detect a large amount of program variations such as loop transformations (unroll, splitting, tiling, etc...) and appears to be scalable, and thus applicable to real-life applications. In addition, the rewriting of the program with high-level functions increases readability, making `TeMa` a promising tool to improve program comprehension and help programmers in the tedious task of software maintenance.

In future works, we would like to automatize the generation of SPL code, and validate it on benchmark applications. Additionnaly to portability and software maintenance, it should also increase performance since SPL enable specific algorithmic optimizations, which were not possible at a lower level of semantics.

References

1. C. Alias and D. Barthou. Algorithm recognition based on demand-driven data-flow analysis. In *10th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, November 2003.
2. D. Barthou, P. Feautrier, and X. Redon. On the equivalence of two systems of affine recurrence equations. In *8th International Euro-Par Conference*, page 309. Springer, LNCS 2400, 2002.
3. A. Cimetile, A. De Lucia, and M. Munro. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance: Research and Practice*, 8(3):145–178, 1996.
4. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1997. release October, 1rst 2002.
5. J. E. Doner. Decidability of the weak second-order theory of two successors. *Notices Amer. Math. Soc.*, 12:365–468, March 1965.
6. J. E. Doner. Tree acceptors and some of their applications. *Journal of Comput. and Syst. Sci.*, 4:406–451, 1970.

7. J. Dongarra. The linpack benchmark: An explanation. In *Proceedings of the 1st International Conference on Supercomputing*, pages 456–474. Springer-Verlag, 1988.
8. J. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.
9. S.-M. Kim and J. H. Kim. A hybrid approach for program understanding based on graph-parsing and expectation-driven analysis. *Journal of Applied A.I.*, 12(6):521–546, September 1998.
10. W. Kozaczynsky, J. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Trans. on S.E.*, 18(12):1065–1075, December 1992.
11. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of CGO'2004*, Palo Alto, California, Mar 2004.
12. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.
13. B. Di Martino and G. Iannello. PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *IWPC'04*, pages 164–174. IEEE Computer Society Press, 1996.
14. R. Metzger and Z. Wen. *Automatic Algorithm Recognition: A New Approach to Program Optimization.* MIT Press, 2000.
15. S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann, 1997.
16. M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *J. of High Perf. Computing and Applications*, 1(18):21–45, 2004.
17. J.W. Thatcher and J.B. Wright. Generalized finite automata. *Notices Amer. Math. Soc.*, 1965.
18. J.W. Thatcher and J.B. Wright. Generalized finite automata with an application to a decision problem. *Mathematical System Theory*, 2:57–82, 1968.
19. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
20. L. M. Wills. *Automated Program Recognition by Graph Parsing.* PhD thesis, MIT, July 1992.
21. Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. Spl: A language and compiler for dsp algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, pages 298–308, 2001.