TeMa : an Efficient Tool to find High-Performance Library Patterns in Source Code

Christophe Alias

TeMa: an Efficient Tool to Find High-Performance Library Patterns in Source Code

Christophe Alias

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

Abstract. In this paper, we present a linear algorithm to find fastly all possible instances of a pattern in a program. It has been implemented in our TeMa tool and connected with a more expensive method already described to check whether the program slices found are effectively instances of the pattern. In case of success, our exact method provides the corresponding values of pattern wildcards. Experimental results on SPEC benchmarking suite show that our method is able to handle much program variations than previous approaches, and can be applied to real-life applications.

1 Introduction

High performance computing applications require obviously a quite optimized code. Unfortunately, the compiler optimizing pass is not enough, and leads the programmer to use optimized libraries such as ATLAS [24], FFTW [9] or PhiPAC [25]. It is surprising how little the compiler helps the programmer in this fastidious task. Research effort has been made to provide a convivial and efficient environment to browse library functions. Several approaches are able to retrieve a library function given pre- and post-conditions (see for example [11, 20, 8]). But these approaches can only be used at first development time, and are useless when the programmer have to rewrite a software that use a performance library.

A natural solution should be to search naive occurrences of library functions through the program. Paul and Prakash [19] proposes an extension of `grep` to find *program patterns* in source code. Unfortunately, their approach is purely syntactic and cannot handle many program variations. Wills [26] 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. We 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. Unfortunately, Wills' approach use expensive since the flow-graph recognition problem is NP-complete. Moreover, a pattern data base made of graph grammar rules seems to be difficult to maintain.

In this paper, we present a linear algorithm to quickly find all possible instances of a pattern in a program. The basic idea is to step simultaneously

the pattern and the program among def-use chains while the arithmetic operators found are equal. If the stepping reach the last statement if the pattern, all reached program statements are returned as a potential occurrence of the pattern. It is implemented in the `TeMa` tool and connected with a more expensive method already described in [1] to check whether the program slices found are effectively instances of the pattern. In case of success, our exact method provides the corresponding values of pattern wild-cards. Experimental results on SPEC benchmarking suite [10] show that our method is able to handle much program variations than previous approaches, and can be applied to real-life applications.

The paper is organized as follows. Section 2 introduces the notations and definitions used in the paper. Section 3 describes our new technique to extract the program slices similar to a given pattern. We also provide a complexity study, and a classification of program variations handled. Section 4 presents the `TeMa` tool, and its graphical interface. Section 5 provides experimental results. Finally, section 6 gives a short survey on related works.

2 Background

This paper presents a tool to find all relevant instances of a specific pattern in source code. A *pattern* a schema of program with wild-cards values and functions. For example, figure 1 gives the pattern of a reduction, where wild-cards are denoted by □. The 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 in 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 □.

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*. Instead of indicating whether two algorithms compute the same (mathematical) function, Herbrand-equivalence just indicate 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 weak than semantic equivalence, and seems to be easier to check, it has been unfortunately proved 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 [21, 22] in the context of circuit verification. Most of usual operations on word automata (determinization, minimization, cartesian product, ...) extend naturally to tree automata [4].

3 Pattern Detection

In this section, we present our algorithm to detect patterns in source code. We also provide a complexity study showing that our algorithm is linear in the program size. Finally, we evaluate the power of detection in terms of program variations handled.

3.1 Our Algorithm

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 step simultaneously the pattern and the program among def-use chains while pattern's and program's operators are equal. If the stepping reaches the last statement of the pattern, all reached program statements will be returned as a candidate slice.

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. 2. *Build_Automaton*

The pattern and the program are assumed to be given in scalar SSA-form, and normalized with one operator by statement. 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 2. 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's operator (step 2). Pattern wild-cards are handled as Kleene star in word automata. Note that value wild-card is particular case of function wild-card $\Box(\ldots)$ with arity 0. We loop on the corresponding state with all operators which can be found in the program (step 3).

Consider the pattern and the program given in figure 3. For sake of clarity, we have chosen a pattern and a program with unary operators ($1+.$, $\exp.$, $\tan.$, $1/.$) which will lead to nices word automata given in figure 4. But of course, our method can handle operators with any arity (more often $.+.$, $.-.$, $.\times.$, $./.$).

```
r1 = 1
do i = 1,n
| r2 = □(φ(r1,r3))
| r3 = 1+r2
enddo
r4 = exp(φ(r1,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. 3. Pattern (left) and Program (right)

The idea is now to step simultaneously the two automata up to pattern's final state while operators are equals. The two automata have as many entry points as constant leafs (1(), 2() here). To be exhaustive, we have thus to start a comparison from each couple of leafs. It is equivalent to compute the *cartesian product* of the pattern's and the program's automata. The detected slices can then be computed by collecting all programs states along the paths from initial states to each state with a final pattern state ($q_{\text{final}}, .$). The detection step is summarized in the algorithm described in figure 5.

Let us summarize our algorithm. Given a pattern and a program, we put them in SSA-form. We then compute their tree-automata $\mathcal{A}_T$ and $\mathcal{A}_P$ by applying the algorithm *Build_Automaton* described in figure 2. The slices of «good» candidates are then obtained by stepping simultaneously $\mathcal{A}_T$ and $\mathcal{A}_P$. This is done by applying the algorithm *Output_Slices* described in figure 5.

3.2 Complexity issues

Now, let us study the complexity of our detection algorithm. The significant operation is the creation of a transition. Consider a program P given in scalar SSA form, with a statement $r = f(\phi(Q_1)\ldots\phi(Q_n))$. *Build_Automaton* will

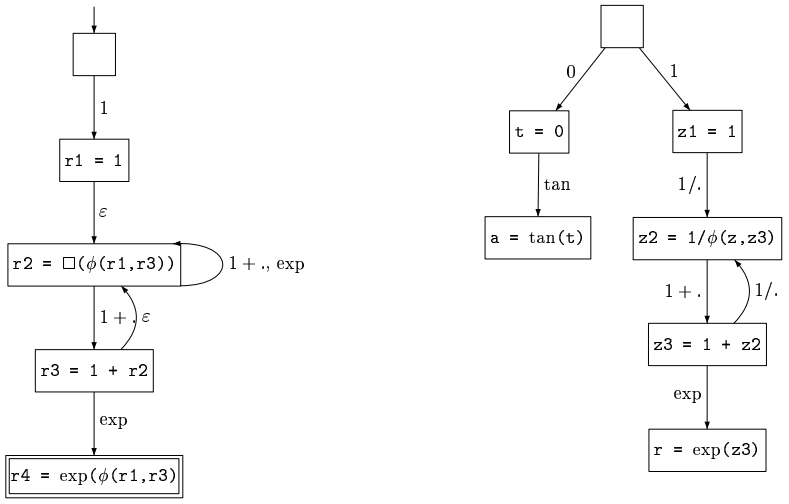

Fig. 4. Pattern's automaton (left), and Program automaton (right)

creates $|Q_1| \times \ldots \times |Q_n|$ transitions. Thus the total number of transitions can be written $P \times d^a$ where d is a average number of sources by reference, and a denotes the average arity of program's operators. Experimentations on SPEC benchmarking suite [] show that $\bar{d} = 4$ and $\bar{a} = 2$. Indeed, f is more often standard arithmetic operators such as $+$, $-$, $\times$ or $/$. Thus, the average complexity of *Build_Automaton* can be written $16P = \mathcal{O}(P)$.

Algorithm *Output_Slices*

Input: $\mathcal{A}_T$ *and* $\mathcal{A}_P$, *pattern's and program's 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 $\mathcal{P}\text{rev}(q) = \{q',\ q' \longrightarrow^* q\}$.
   Then return the $\mathcal{A}_P$ part of $\mathcal{P}\text{rev}(q)$.

Fig. 5. *Output_Slices*

Consider now a pattern T. In the worst case, the number of transitions of $\mathcal{A}_T \times \mathcal{A}_P$ is $\mathcal{T}(\mathcal{A}_T \times \mathcal{A}_P) = \mathcal{T}(\mathcal{A}_T) \times \mathcal{T}(\mathcal{A}_P)$ where $\mathcal{T}(\mathcal{A})$ denotes the number of transitions of $\mathcal{A}$. This case occurs when T and P uses only one operator, which never occurs in real life examples. Thus the worst-case complexity can be

written: $(d^a)^2 T \times P = 256T \times P = \mathcal{O}(P)$. Therefore, our detection algorithm is *linear* in the program size.

3.3 Program variations detected

A common way to evaluate an algorithm recognition system is to provide the different kinds of pattern variations it can handle [26, 12, 17, 18]. 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 give 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
```

Once again, def-use chains are stepped without take into account of variable names. This allows to handle data-structure variations. As stated above, it is also a cause of wrong detections.

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
```

If the control variation do not change the operators nest in the expression computed by the program, it is handled.

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 find a large amount of correct candidates.

4 The TeMa tool

TeMa (Template Matcher) is the implementation of our detection algorithm. Options allow to apply the exact instantiation test described in [1] to check whether the program slices are effectively instances of pattern. In case of success, it provides the corresponding values of pattern wild-cards. TeMa has been implemented in Objective Caml, and represents 10388 lines of code. 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 a visual version with a graphical interface for human usage such as re-engineering, program comprehension or software maintenance.

Schema given figure 6 summarizes the main steps of TeMa . We first analyze the source code of the pattern and the program, in order to put them in SSA-form. This step will outputs a graph representation called SSA-graph. Our front-end is able to handle C and Fortran 90 programs. C front-end use the LLVM compiler infrastructure [15], 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. The two front-ends represent 701 lines of C++ for the C/C++ part, and 3463 lines of Objective Caml for the Fortran 90 part.

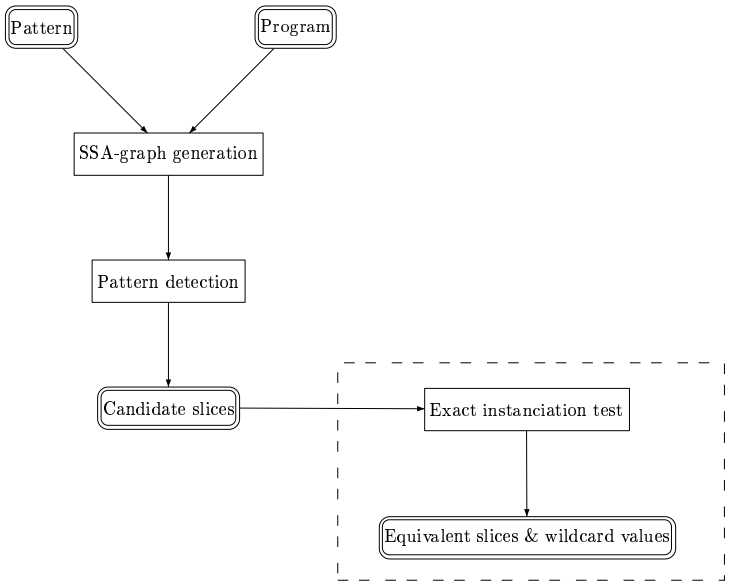

Fig. 6. The main steps of `TeMa`

Next, we apply our detection algorithm to the pattern's and program's SSA-graphs. As stated above, it output all the program slices which potentially match with the pattern. The experimental results presented in the next section shows that the wrong detections are low, thus candidate slices can be keep as a final results. Our detection algorithm represents 3378 lines of Objective Caml.

The user can nevertheless asking for a confirmation that program slices are effectively instances of pattern. Our exact instantiation test is then applied and gives the corresponding values of wild-cards in case of success. Our exact instantiation test represents 2846 lines of Objective Caml.

Figure 7 gives the interface of `TeMa` . The pattern and the program can be loaded in the appropriate panel. Pattern wild-cards are denoted by X. *Options* button allows the user to customize the matching, by choosing for example to apply or not the exact instantiation test after the detection. *Match* button apply the whole algorithm. The user can then inspect the different occurrences found in the program by using the buttons $<$ and $>$.

5 Experimental results

We have applied our pattern detection algorithm to detect potential calls to the BLAS library [16] in LINPACK [7] and four programs involved in the SPEC benchmarking suite [10]. Our pattern base is constituted of direct implementations of BLAS functions from the mathematical description. After having applied our al-

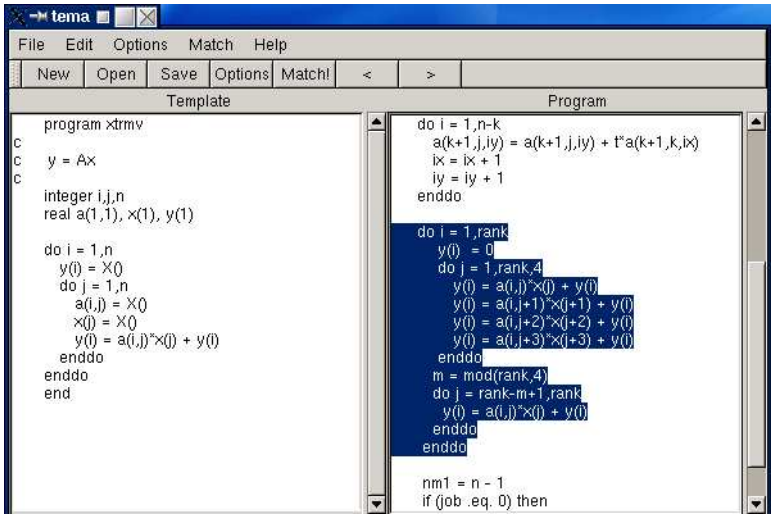

Fig. 7. `TeMa`'s graphical interface

gorithm to each pair of pattern and program, we have checked by hand whether the substitution by a call to BLAS is possible. Table 8 shows the results.

It appears that 1/2 of candidates do not match, 1/4 are instances of patterns for vectors of size 1, and 1/4 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. Nor loop iteration count, nor `if` conditions, nor complex dependences due to array index functions are handled. In addition, our method handles arrays as scalar variables, which can lead to detect a BLAS1 `xaxpy` `y(i) = y(i) + a*x(i)` when there is a reduction `s = s + a(i)*a(i)`. Likewise, the method detects the same number of matrix-matrix multiplication than of matrix-vector multiplication. Note that the detection is correct since a vector is a particular case of matrix, but the code should not be substituted by a BLAS 3.

1/4 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 fits only the second. In addition,

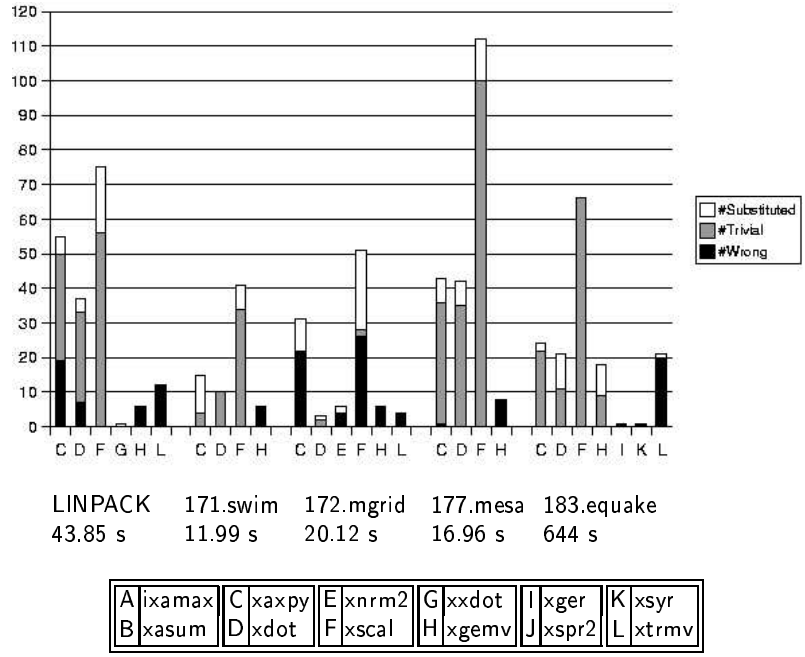

Fig. 8. For each kernel, we provide each BLAS function found, the number of non-equivalent slices (# Wrong), the number of detections with one statement (# Trivial), and the number of candidates interesting to replace (# Substituted). The execution times are given for a Pentium 4 1,8 GHz with 256 Mo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

execution times confirms that our algorithm is linear in the program size. Thus, our detection method is scalable and can be applied to real-life applications.

6 Related Works

We first present related works 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 [23], 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 a 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 that satisfies this requirement is *undecidable* [23]. However an approximation can be found by computing consecutive sets of indirectly relevant statements, according to data-flow and control-flow dependences.

Lanubile and Visaggio [14] added the set of input variables to the slice criterion. They introduced the notion of *transform slice*, as the slice that computes

the values of the output variables at a given point, from the values of the input variables. Basically, the computation of the slice stops as soon as statement that defines values for the input variables are included in the slice.

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.

Paul and Prakash [19] proposes an extension of `grep` to find *program patterns* in source code. They use a pattern language with wild-cards on syntactics entities *e.g.* declaration, type, variable, function, expression, statement, ... which allow to find patterns with specific sequences and imbrications of control structures. For example, here is pattern used to find the maximum in an array of integers:

```
{*
  @[while|dowhile|for] {*
    if($v_1[#] > $v_2) {*
      $v_2 = $v_1[#]
    *}
  *}
*}
```

The wild-card `{* *}` means a statement with an arbitrary nesting depth. `@[while | dowhile | for]` means either `while`, `dowhile` or `for`. `$v_{name}` means a variable labeled by `name` and `#` means an expression. Their algorithm produces and interprets a Code Pattern Automaton (CPA), which traverse the program's AST according to the pattern, and decides if it is an instance or not. They argue the complexity of their algorithm is $O(n^2)$ with n the number of AST nodes. Obviously, their approach is limited to one programming language, and forces the user to make strong assumptions in the implementation of patterns. It seems this approach cannot handle many other program variations than variable renaming (`$v_{name}`) and garbage code (`{* *}`). Thus candidate slices should be quite limited. Moreover, their algorithm is more expensive than ours.

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 [12, 13] 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 that can achieve these goals. Bottom-up methods [17, 26] start from the program statements and try to find the corresponding plans. Wills [26] 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. We 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. 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 [18] 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 search the program for good candidate slices. 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 across loop iterations for instance is not handled. In the same way, the control variations supported are bounded to pre-treatments.

7 Conclusion

In this paper, we have presented a new and efficient pattern detection algorithm, implemented in the tool `TeMa`. Given a pattern and a program, `TeMa` allows the user to browse to different occurrences of the pattern in the program. Our detection method is linear in the program size, and has been validated by searching immediate mathematical descriptions of 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 exact method already described in [1] can applied by `TeMa` to check whether the program slices found are effectively instances of the pattern.

In a future work, we would like to automatize the replacement of relevant program slices by a call to the corresponding library function. Such a method will fully automatize the software rewriting *w.r.t.* a high performance library. Apart from high-performance, it will also ensure a better portability of software in a fully automatic manner.

Acknowledgments

The author would like to thank Denis Barthou and Paul Feautrier for their valuable help and suggestions.

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. B. Fischer. Specification-based browsing of software component libraries. In *Automated Software Engineering*, pages 74–83, 1998.
9. M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.
10. J. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.
11. J.-J. Jeng and B. H. C. Cheng. Using formal methods to construct a software component library. In Ian Sommerville and Manfred Paul, editors, *Proceedings of the Fourth European Software Engineering Conference*, pages 397–417. Springer-Verlag, 1993.
12. 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.
13. W. Kozaczynsky, J. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Trans. on S.E.*, 18(12):1065–1075, December 1992.
14. F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Trans. on S.E.*, 23(4):246–259, 1997.
15. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of CGO'2004*, Palo Alto, California, Mar 2004.
16. 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.
17. 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.
18. R. Metzger and Z. Wen. *Automatic Algorithm Recognition: A New Approach to Program Optimization*. MIT Press, 2000.

19. S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Trans. on S.E.*, 20(6):463–475, June 1994.
20. John Penix, Phillip Baraona, and Perry Alexander. Classification and retrieval of reusable components using semantic features. In *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, pages 131–138, 1995.
21. J.W. Thatcher and J.B. Wright. Generalized finite automata. *Notices Amer. Math. Soc.*, 1965.
22. J.W. Thatcher and J.B. Wright. Generalized finite automata with an application to a decision problem. *Mathematical System Theory*, 2:57–82, 1968.
23. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
24. R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.
25. Chee whye Chin, Jeff Bilmes, Jim Demmel, and Krste Asanovic. The PHiPAC v1.0 matrix-multiply distribution. Technical report, October 23 1998.
26. L. M. Wills. *Automated Program Recognition by Graph Parsing.* PhD thesis, MIT, July 1992.