Deciding Where to Call Performance Libraries

Christophe Alias and Denis Barthou

Laboratoire PRiSM, Université de Versailles, France.
Christophe.Alias@prism.uvsq.fr
Denis.Barthou@prism.uvsq.fr

Abstract. As both programs and machines are becoming more complex, writing high performance codes is an increasingly difficult task. In order to bridge the gap between the compiled-code and peak performance, resorting to domain or architecture-specific libraries has become compulsory. However, deciding when and where to use a library function must be specified by the programmer. This partition between library and user code is not questioned by the compiler although it has a great impact on performance. We propose in this paper a new method that helps the user find in its application all code fragments that can be replaced by library calls. The same technique can be used to change or fusion multiple calls into more efficient ones. The results of the alternative detection of BLAS 1 and 2 in SPEC are presented.

1 Introduction

The recent generation of microprocessors can deliver high performance thanks to a large number of mechanisms: cache hierarchies, branch prediction, specific instructions such as fused multiply add, speculative execution, predicated instructions, prefetches, etc. One way to obtain high performance code is to rely on compiler optimizations. However the complex optimizations that tap these hardware features come at the expense of performance stability. For instance, multiversioning is an optimization generating several versions for the same code fragment, these versions are selected dynamically depending on parameters such as loop iteration count or data alignment. But a bad choice for the strategy selecting the different versions can introduce important latencies. Another approach has focused on library tuning as a more reliable way to deliver performance. The assembly code is either generated by hand, using architecture specific instructions, or by adaptative code generation (*e.g.* ATLAS [14], FFTW [10] or STAPL [12]). The important compilation time is then balanced by the reusability of the libraries. In all cases, library functions can be considered as the building blocks, essential to get high performance on real codes. In general programming languages, code tuning is performed in the last stage of the development process. The selection of the library functions and the rewriting of the code falls under the responsibility of the user. The usual steps of this process are: find out code fragments and library functions that are semantically equivalent, replace these fragments by function calls with correct parameters, debug the application and finally evaluate performance. In the case of non-portable libraries, this time

consuming process has to be reconducted for each target architecture. It is surprising how little the compiler helps the user in this tedious task. Compile-time optimizations neither change the partition between library and user code, nor cross library boundaries.

This paper presents an efficient method to find in a program all code fragments that match library functions. Programs under study are any C or Fortran codes, and libraries can be template libraries (with the meaning of C++ templates). In general, deciding whether two codes are semantically equivalent is undecidable. The equivalence considered in our approach does not take into account any special operator semantics, such as associativity or commutativity. Within this framework, the method presented is conservative: some of the fragments found are not semantically equivalent to the library codes, but none of the truly equivalent fragments is missed. The analysis produces "may" information: between lines 537 and 541, it may be a matrix-vector product. Combined with an exact but more expensive method [2] applied only on fragments, both analyses would produce "must" information, also providing the effective parameters for the library call and its instantiation if this is a template. Finally, we describe in this paper the conditions for which code substitution by function calls is safe. Note that as a prerequisite for the detection step, each library function has to be described by a program. We do not assume that the analysis has access to the source of the library. Instead we assume the library designer provides a public version for each function. This program must have the same semantics than the optimized, private version but the algorithm used can be completely different.

Section 2 presents some related work. Section 3 describes the new detection technique. In Section 4, we sum up, out of completion, the method used to prove the equivalence and to find the parameters of the call. We then give the conditions for a safe substitution in Section 5 and conclude in Section 6 with the results of our experiments on SPEC benchmarks.

2 Related Work

The detection of code matching library functions is related to the detection of slices, which consists of identifying all the statements contributing to a given computation. Cimetile et al. [1] propose a semi-automatic approach to extract program parts (slices) verifying pre- and post-conditions. They rely on a theorem prover which requires user interaction to assert some invariants, and has a high complexity, which makes it unrelevant to large applications.

Another approach proposed by Paul and Prakash [16] describes an extension of `grep` in order to find *program patterns* in source code. They use a pattern language with wild-cards on syntactic entities *e.g.* declaration, type, variable, function, expression, statement,...allowing to search for specific sequences and nested control structures. Their algorithm has a $O(n^2)$ complexity with n the code size. This detection method has the same goal as ours; one of its drawbacks is that the same pattern cannot handle variations in control (loop unroll,

tiling) or in data structures (array expansion, scalar promotion) whereas this is addressed in our framework.

Finally, several approaches encode the knowledge of the functions to be identified in the form of programming plans. Top-down methods [15] use the knowledge of the goals the program is assumed to achieve and some heuristics to detect both the program slice and the library functions that can achieve these goals. Bottom-up methods [9] start from statements and try to find the corresponding plans. Wills [9] represents programs by a flow-graph, and patterns by grammar rules. The recognition is performed by parsing the program graph according to the grammar rules and has an exponential cost at worst. Metzger and Wen [13] have built a complete environment to recognize and replace algorithms. They first normalize both program and pattern abstract syntax tree by applying usual program transformations (if-conversion, loop-splitting, scalar expansion...). Then they consider all strongly connected components in the dependence graph, containing at least one `for` statement as candidate slices. Their method provides therefore a large number of candidate slices with many false detections, which is balanced by the low complexity of their equivalence test. Compared to the combination of the detection with the instantiation test we recall in this paper, they can handle fewer program variations (reuse of temporaries across loop iterations for instance is not handled) for a lower cost.

3 Detection of Library Templates

The detection of library templates consists in localizing in a code the lines that possibly correspond to a given library function or template. In the case of a template, the code detected is a possible instance of the template. We propose an efficient method based on a symbolic execution of both program and template, following the def-use chains. The method symbolically executes both program and template slices simultaneously and compares the sequence of operators along these slices, abstracting away the number of iterations of the loops.

3.1 Principle

The template and the program are assumed to be given in SSA-form, and normalized with one operator by statement. Each edge of the program SSA-graph is labeled with its operator. Loops create cycles in this graph but we abstract away the number of iterations. The sequence of operators along a path is considered as a word and the graph can be considered as a finite automaton. The idea of the algorithm is to check whether the language of operators generated by some code fragment is included in the language of operators generated by a library function. Intuitively, this ensures that the same sequence of operations can happen in the code and in the library function.

Figure 1 provides a very simple example of matching problem between a template and a program. The template and the program are assumed to be given in SSA-form, which means that the variables are assigned one time *at*

most in the program text. In addition, each reference to a variable is substituted by a ϕ-function providing the set of its potential values. For example, the ϕ-function used in the assignment P_4 means that `z2 = 1/z1` or `z2 = 1/z3`. Since statements assigning a constant such as T_1, P_1 or P_2 have no predecessors in the graph of def-use chains, they can be taken as a starting point for the inspection.

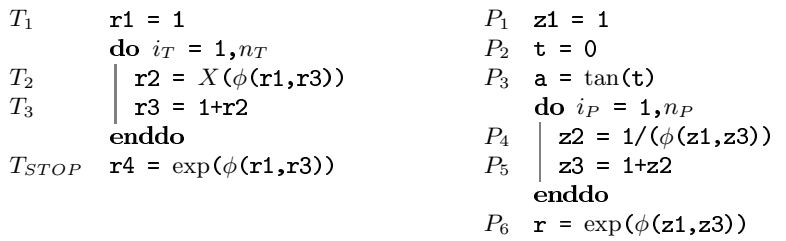

Fig. 1. A template (left) and a program (right)

Starting from P_1, a stepping among def-use chains would follow the sequence:

$$\xrightarrow{1} P_1 \xrightarrow{1/.} P_4 \xrightarrow{1+.} P_5 \xrightarrow{\exp} P_6$$

Likewise for the template a possible sequence of operators is:

$$\xrightarrow{1} T_1 \xrightarrow{X(.)} T_2 \xrightarrow{1+.} T_3 \xrightarrow{\exp} T_{STOP}.$$

Walking through both program and template, with the condition that for each transition, the operator must be the same, we obtain the sequence:

$$\xrightarrow{1} (T_1, P_1) \xrightarrow{1/.,X(.)=1/.} (T_2, P_4) \xrightarrow{1+.} (T_3, P_5) \xrightarrow{\exp} (T_{STOP}, P_6).$$

This provides the candidate slice $\{P_1, P_4, P_5, P_6\}$, that possibly corresponds to the template provided that $X(.) = 1/.$ (this condition appears on the transition). This condition is necessary for the sequence to be the same for both template and program. Note however that the method will not check the coherence between the values of template variables. Likewise, the number of iterations in loops or the branches chosen in conditionals are ignored. These important points will be checked during the exact instantiation test (see Section 4).

3.2 Detailed Algorithm

Following the idea described above, we build an automaton recognizing the sequences of operators executed by all possible instances of the template, and an automaton recognizing the sequences of operators executed by the program. The

simultaneous stepping of he template and the program is then achieve by computing the Cartesian product of the template's and the program's automaton, which provides the candidate slices.

Figure 2 provides the automata built from the template and the program provided in the above example. we associate the states represent the assignments, and the transitions are driven by the flow-dependences given in the ϕ-functions, and labeled by the operator used in the destination state. Since the template's automaton aims to recognize all possible template's instances in the program, the states involving a template variable X are handled by adding a looping transition for each program's operator. Since most operators have an arity greater

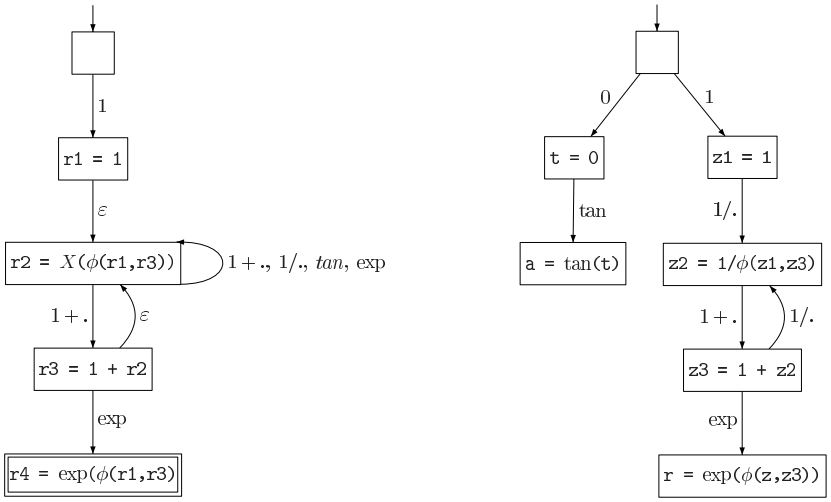

Fig. 2. Automata build associated to the template (left), and the program (right)

than 2, word automata are not expressive enough in general. Instead we build a *tree-automaton*, using the algorithm described in figure 3. There is no major difference with the word automata: we associate a state to each assignment then we add transitions according to the dependences given by the ϕ-functions (step 2). Remark that when $n = 1$, we obtain a word automaton since $f(q_1) \longrightarrow q$ can be interpreted as $q_1 \xrightarrow{f} q$. X is handled as a wild-card, which leads to add looping transitions with the operators used in the program (step 3). Likewise, remark the ε-transitions $q_i \longrightarrow q$.

The detection is achieved by stepping simultaneously the template's and program's automata as soon as the operators are the same. Each stepping leading to the final state of the template will provide a candidate slice in the program, built of all reached program's statements. These steppings can be performed in an exhaustive manner by computing the Cartesian product $\mathcal{A}_T \times \mathcal{A}_P$ of the

Algorithm *Build_Automaton*

Input: *The template 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:

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

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

Fig. 3. *Build_Automaton*

program and template automata. It remains to mark the states (q_T, q_P) with a final state q_T of $\mathcal{A}_T$, and to emit the states of $\mathcal{A}_P$ on a path from the initial state as a *potential* instance.

Our method is able to detect any template variation which does not involve the semantic properties of operators such as associativity, or commutativity. Particularly we can handle any loop transformation and most control restructuring transformations. Moreover, our method is completely independent of data structure used, which allows the detection of a large amount of template variations in the program. Whether a slice detected is a real instantiation of the template is determined during the exact instantiation test.

In the worst case, the construction of the Cartesian product of the template and the program automata is computed in $O(T \times P)$ where T is the number of template statements and P is the number of program statements, i.e. the complexity is linear in the size of the program analyzed.

4 Exact Instantiation Test

Once the candidate slices are found, we have only detected a code that "may" match the library template. Either the user decides from this information to substitute or not, or another procedure decides if both program and template are indeed equivalent and finds the instantiations. We recall the main steps of this procedure eliminating false detections, described in [2].

The instantiation test follows the steps of the detection method described in Section 3. An exact instance-wise reaching definition analysis is performed. As reaching definitions may depend on the values of iteration counters, these

conditions are put on the transitions of the tree-automata. Deciding if the code fragment under study is an instantiation of the template boils down to compute the loop counter values that can reach final states of the Cartesian-product automaton. Efficient heuristics [17] perform this computation.

The power of this instantiation test is assessed according to its capacity to prove the equivalence between two codes, one a variation of the other. The test handles variations coming from loop transformations (splitting, fusion, skewing, tiling, unroll,...), from data structures (scalar expansion, scalar promotion, use of temporaries), from common subexpression elimination or other factorization of computation. However, the test does not handle the semantic properties of the operators, such as commutativity or associativity.

5 Substitution

Once candidate slices are found, it remains to substitute them by a call to an optimized library. We describe thereafter an algorithm to decide whether a substitution preserves the program semantics, and to perform the substitution in case of success.

Detected slices are often interleaved with other program statements. We have first to separate them from these statements. Consider an algorithm A consisting in the set of operations $\{(A_1, I_1) \dots (A_a, I_a)\}$, where A_i is a statement, and I_i a set of iteration vectors. Let $(A_1, \boldsymbol{i}_1)$ be its first operation, and $(A_a, \boldsymbol{i}_a)$ its last operation. Its *complementary* is the set of program operations executed between the first and the last operations of A:

$$\overline{A} = \{(S, \boldsymbol{i}) \mid (A_1, \boldsymbol{i}_1) \prec (S, \boldsymbol{i}) \prec (A_n, \boldsymbol{i}_n) \text{ and } S \text{ is not an } A_i\} \qquad (1)$$

Consider the following example (left):

```
P1  s = 0
    do i = 1, 10
P2  |  a(i) = a(i-1) + 1
    |  if i >= 9 then
A1  |  |  dot = dot + 2*a(i)
    |  endif
    enddo
A2  dot = dot + b*c
P3  s = s + 1
    do i = 1, 4
P4  |  s = s + b(i)
A3  |  dot = dot + a(i)*b(i)
    enddo
```

(a). Original Program

```
P1  s = 0
    do i = 1, 10
P2  |  a(i) = a(i-1) + 1
    |  if i ∉ { 9, 10 } then
    |  |  if i >= 9 then
A1  |  |  |  dot = dot + 2*a(i)
    |  |  endif
    |  endif
    enddo
(A2 removed)
P3  s = s + 1
    do i = 1, 4
P4  |  s = s + b(i)
    |  if i = 4 then call Optimized_A
    |  if i ∉ { 1, 2, 3, 4 } then
A3  |  |  dot = dot + a(i)*b(i)
    |  endif
    enddo
```

(b). Program with substitution

where the recognized algorithm is constituted of operations:

$$A = \{(A_1, \{9, 10\}), (A_2, \{\mathbf{0}\}), (A_3, \{1, 2, 3, 4\})\}$$

Its complementary is thus: $\overline{A} = \{(P_2, \{10\}), (P_3, \{\mathbf{0}\}), (P_4, \{1, 2, 3, 4\})\}$. For each statement P in the program, we compute the set of corresponding operations between the first and last operations of A by giving relation (1) to a solver [5]. If it is not empty, we emit it.

Once $\overline{A}$ is computed, it remains to separate it from A in order to replace A by a call to an optimized library. A is *separable* if all dependences go exclusively from A to $\overline{A}$, or exclusively from $\overline{A}$ to A. In the first case, A can be substituted by a call to A before $\overline{A}$. In the other case, the call has to be insert after $\overline{A}$. Otherwise, we do not perform substitution. In the example given above, A is separable and can be replaced by a call after $\overline{A}$ because of a dependence from $(P_2, 10)$ to $(A_1, 10)$. In addition, if an intermediate variable is alive outside the slice, we do not perform the substitution.

The substitution can now be performed by deleting operations of A, and placing the relevant call before, or after $\overline{A}$. Consider the above example (right). Relevant operations of statements A_1 and A_3 are disabled using a condition. Because A_2 have no nesting loops, it is just removed from the program text (step 2). As said above, the optimized call is inserted after the last operation of $\overline{A}$ $(P_4, 4)$, using a condition. A more efficient code can be produced by first reschedule operations of $\overline{A}$, and then generating efficiently the code with an appropriate method [6].

6 Experimental results

We have implemented the SSA-graph construction from fortran applications, and for `C` and `C++` applications using the LLVM compiler infrastructure [3]. We have applied our slicing algorithm to detect potential calls to the BLAS library [4] 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. After having applied our algorithm to each pair of pattern and program, we have checked by hand whether the slices are equivalent to the pattern, and if the substitution by a call to BLAS is possible. Figure 4 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 different 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 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

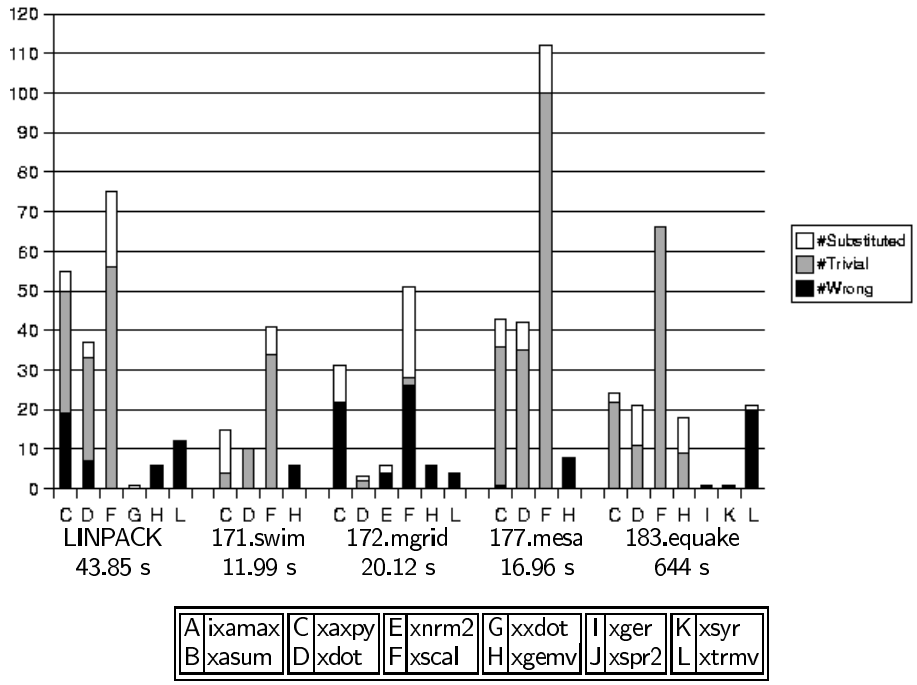

Fig. 4. For each kernel, we provide each BLAS function recognized, the number of wrong slices (# Wrong), the number of trivial detections (# Trivial), and the number of candidates interesting to replace (# Substituted). The experimentation was done on a Pentium 4 1,8 GHz with 256 MB RAM.

vector is a particular case of matrix, but the code should not be substituted by a BLAS 3.

For 1/4 of the slices, the 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, execution times confirm that our algorithm is linear in the program size. Thus, our slicing method is scalable and can be applied to real-life applications.

7 Conclusion

The method presented shows that the compiler can help the user write or rewrite a code with high performance libraries. Combined with an instantiation test,

this process can be fully automatic. The advantages are a better portability and higher productivity of the programmer. The detection only requires that each library function has a public version, in C or Fortran, semantically equivalent to the real code. The experiments on the SPEC benchmarks are encouraging: the method detects a significant number of linear algebra functions with linear complexity. The evaluation of the performance gain expected when using library calls is still however an ongoing work.

More generally, this approach can change the abstraction level of the program, replacing C code by algorithms or formulae. From this higher level of abstraction, it enables a change of algorithm [11] or simply improves code comprehension. For large scale applications, high performance cannot be at the expense of portability. The method described could be a solution to combine both and this will be the subject of future work.

References

1. A.Cimetile, A.De Lucia, and M.Munro. A specification driven slicing process for identifying reusable functions. *J. of Software Maintenance: Research and Practice*, 8(3):145–178, 1996.
2. C.Alias and D.Barthou. Algorithm recognition based on demand-driven data-flow analysis. In *Working Conf. on Reverse Engineering*. IEEE, 2003.
3. C.Lattner and V.Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of CGO'2004*, Palo Alto, 2004.
4. C.Lawson, R.Hanson, D.Kincaid, and F.Krogh. Basic Linear Algebra Subprograms for Fortran usage. *Trans. on Mathematical Software*, 5(3):308–323, 1979.
5. D.Wilde. A library for doing polyhedral operations. INRIA TR 2157, 1993.
6. F.Quilleré, S.Rajopadhye, and D.Wilde. Generation of efficient nested loops from polyhedra. *Int. J. of Parallel Programming*, 28(5):469–498, 2000.
7. J.Dongarra. The linpack benchmark: An explanation. In *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. L.Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, MIT, 1992.
10. M.Frigo and S.Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. Intl. Conf. Acoustics Speech and Signal Processing*, volume 3, pages 1381–1384. IEEE, 1998.
11. 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.
12. N.Thomas, G.Tanase, O.Tkachyshyn, J.Perdue, N.Amato, and L.Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *Proc. ACM PPOPP'05*, Chicago, 2005.
13. R.Metzger and Z.Wen. *Automatic Algorithm Recognition: A New Approach to Program Optimization*. MIT Press, 2000.
14. R.Whaley and J.Dongarra. Automatically tuned linear algebra software. In *SuperComputing*. Springer-Verlag, 1998.

15. S.Kim and J.Kim. A hybrid approach for program understanding based on graph-parsing and expectation-driven analysis. *J. of Applied A.I.*, 12(6):521–546, 1998.
16. S.Paul and A.Prakash. A framework for source code search using program patterns. *IEEE Trans. on S.E.*, 20(6):463–475, 1994.
17. W.Kelly, W.Pugh, E.Rosser, and T.Shpeisman. Transitive closure of infinite graphs and its applications. *Int. J. of Parallel Programming*, 24(6):579–598, 1996.