PERGAMON

Computer Languages 24 (1998) 55–72

COMPUTER LANGUAGES

Evaluating inlining techniques

Owen Kaser[a], C.R. Ramakrishnan[b]

[a]*Department of Mathematics, Statistics, and Computer Science, University of New Brunswick, Saint John, N.B., Canada, E2L 4L5*

[b]*Department of Computer Science, SUNY at Stony Brook, Stony Brook, NY, 11794, USA*

Received 31 March 1997; received in revised form 10 March 1998

Abstract

For efficiency and ease of implementation, many compilers implicitly impose an "inlining policy" to restrict the conditions under which a procedure may be inlined. An inlining *technique* consists of an inlining policy and a *strategy* for choosing a sequence of inlining operations that is consistent with the policy. The effectiveness of an inlining technique is affected by the restrictiveness of the inlining policy as well as the effectiveness of the (heuristic) inlining strategy. The focus of this paper is on the comparison of inlining policies and techniques, and the notions of *power* and *flexibility* are introduced. As a major case study, we identify and compare policies based on the *version* of the inlined procedure that is used. © 1998 Elsevier Science Ltd. All rights reserved.

Keywords: Inline; Version; Optimization; Comparison

Categories and subject descriptors: D.3.4 [*Programming Languages*]: Processors—*optimization*

1. Introduction

Procedure inlining is widely used by optimizing compilers for imperative and declarative languages [1–7]. Inlining permits a trade-off between code size and execution speed. The goal of inlining has traditionally been to reduce the execution time with a limited expansion of code space. The gain in execution speed may be due to *direct effects*, such as the reduction in the number of call and return instructions executed or due to *indirect effects*, such as cache and virtual memory behavior and in-context optimizations permitted on the inlined code [8–10]. Inlining has also been used to convert as much indirect recursion to direct recursion as possible [11], and to minimize the control stack usage when the program is run with a given input [12].

For efficiency and ease of implementation, many compilers restrict the conditions under which a procedure may be inlined. For instance, a compiler may inline only leaf procedures, or only those that are explicitly declared as inlineable by the user. In some compilers (e.g., [4]), the calls that may be inlined depend on the order in which the procedures are declared. Several diverse heuristics are used to restrict inlining of recursive procedures. We call the restrictions imposed on inlining the *inlining policy*. An inlining *technique* consists of an inlining policy and a *strategy* for choosing a sequence of inlining operations that is consistent with the policy. Given a policy and a goal, it is often difficult to find an efficient strategy. For instance, maximising the reduction in dynamic function calls, given a limited code space expansion, is NP-hard [6]. Thus, most strategies are based on heuristics, and a greedy strategy is often chosen. Clearly, the effectiveness of an inlining technique is affected by the restrictions imposed by the inlining policy as well as the effectiveness of the (heuristic) inlining strategy. The focus of many previous studies has been to show that some restricted inlining techniques still achieves good practical results. However, there has been no attempt to clearly separate the limitations imposed by the policy and the strategy.

The focus of this paper is on the comparison of inlining policies and techniques. First, we formalize the *power* and *flexibility* of inlining policies (see Section 3). We then identify policies based on the version of the inlined procedure that is used, as described below. Finally, using the greedy strategy with different version-based policies, we obtain various techniques that are then compared experimentally.

The version issue: There may be many versions of each procedure during inlining, but only one version will be used at a particular step. For instance, suppose a call site within the body of procedure P is inlined. After inlining, we have a new version of P, say P', which is semantically equivalent to the version of P before inlining. Now, if a call to P is subsequently inlined, it is equally valid to substitute P' (in fact, *any* version of P) in place of the call. However, the two versions may have important operational differences: for instance, substituting with P' may result in fewer function calls but may consume more code space. Thus, the choice of the version to inline may impact the effectiveness of the inlining technique. Although some inliners have such restricted policies that the version issue is unimportant, the less restricted inliners described in the literature implicitly use a policy based on version. For instance, the inliners described in [3, 4, 6] use the current version of a procedure, i.e., the most recent version available when the call is inlined. We show that any policy based exclusively on inlining current versions is inherently less powerful than a policy that allows any version to be inlined. In contrast, we describe a policy based exclusively on inlining original versions of procedures (the version *before* any inlining was performed), which is as powerful as a policy that allows any version to be inlined. We also show, in Section 3.3, that the version issue is important only in the context of recursive programs.

We then consider greedy strategies with limited lookahead and propose a method for estimating the effect of one inlining step (see Section 4). We show that the lookahead-based strategy naturally generalizes the special-purpose greedy strategies used in previous inliners. For instance, we demonstrate that our strategy specializes to Scheifler's inliner when Scheifler's inlining policy [6] is used; if our less-restrictive policy is used, we obtain a new "hybrid inliner". We present experimental results (in Section 4.4) that show that the hybrid inliner performs uniformly better when compared with Scheifler's inliner, and with the inliners described in [4] and [7].

2. Preliminaries

We assume a C-like language with block structures, and without nested procedure declarations. A program $\mathscr{P}$ is composed of a collection of n procedures P_1 , $P_2,\ldots,P_n$. The locations in P_i from which calls are made to statically known procedures are the *call sites* of P_i. For convenience, all call sites in a program are uniquely numbered.[1] There may be other locations where calls are made to procedures that are known only at run-time (as in C, via function pointers), but this article does not consider these locations to be call sites. A call graph provides an abstract of this program information.

Call graph: A *call graph* for a program $\mathscr{P}$ is a labeled, directed multigraph with a node for each procedure P_i. There is an arc (P_i, P_j) labeled a in the call graph if and only if a is a call site in P_i (the caller) at which P_j (the callee) is invoked. The notation *caller*(a) and *callee*(a) is used to indicate the caller and callee of arc a. (We may also refer to a as the a^{th} call site in the program.) Following [4], we add an extra node, SYSTEM, to the call graph. It reflects the interaction of the program with its environment, especially the operating system. The call graph contains an arc from SYSTEM to every other node, since the operating system can potentially invoke any procedure. Since we need not consider arcs from other procedures to SYSTEM, we omit them from the call graphs. We say that a program is recursive if its call graph contains a circuit; otherwise, it is said to be nonrecursive.

An inlining operation replaces a *single* call site, for instance a call from P_i to P_j, with the code of P_j, nesting this new code within P_i as a new block. Minor adjustments to the code may be required to preserve the semantics of P_i—for instance, parameter passing must be simulated. Note that only one replacement is done, even if there are other call sites invoking P_j. Moreover, any semantically equivalent version of P_j's code may be used. Also note that the operation will create new call sites, if and only if the chosen version of P_j contains call sites. After the operation, P_i now has another, semantically equivalent version—the current version of P_i—available for future inlining operations. After all inlining operations, we obtain a collection of versions for each procedure. The final program is assumed to include the most recently derived version of each procedure.

Code size: Consider inlining a call to P_j from P_i to create a new version of P_i, say, P_i'. The size of P_i' is often less than size (P_i) + size (P_j) due to the elimination of the space overhead in parameter passing, and the increased scope for optimization. Moreover, if this were the only call to P_j in the program, and P_j cannot be called directly from SYSTEM (e.g., static functions in C), we could omit P_j from the program. Hence, the code size of an inlined program may actually be smaller than that of the original program. For simplicity, though, we conservatively assume the size of the new version is the sum of its parts' sizes, and that no procedure is omitted from the program after inlining.

[1] Call sites are not "renumbered" when other sites are eliminated or new sites are added.

2.1. Flexibility and power

Recall that an inlining *policy* specifies the conditions under which calls may be inlined, and hence restricts the set of programs that can be derived by inlining. Examples of inlining policies are: "No directly recursive calls may be inlined", and "Only the current version of the callee's code may be used when inlining". Let I_1 and I_2 be two inlining policies, $\mathscr{P}$ be any program, and $\mathbb{P}_{I_1}^{\mathscr{P}}$ and $\mathbb{P}_{I_2}^{\mathscr{P}}$ be the sets of final programs that can be formed when inlining on $\mathscr{P}$ is constrained to follow I_1 or I_2, respectively. The notion of relative *flexibility* of two inlining policies is defined as follows.

Definition 1 (flexibility). The policy I_1 is **at least as flexible as** I_2 if and only if $\forall\mathscr{P}\ \mathbb{P}_{I_2}^{\mathscr{P}}\subseteq\mathbb{P}_{I_1}^{\mathscr{P}}$. If I_1 is at least as flexible as I_2 and $\exists\mathscr{P}\ \mathbb{P}_{I_2}^{\mathscr{P}}\subset\mathbb{P}_{I_1}^{\mathscr{P}}$ then we say I_1 is **more flexible** than I_2.

Although flexibility is useful for comparing two policies, it is possible that the additional programs that one policy can derive are inferior to the best program that another policy can derive, under some metric. To capture this, we define the notion of relative *power* of two policies as follows.

Definition 2 (power). Given a metric μ that measures the merit of a program, a policy I_1 is **as least as powerful as** I_2 under μ if and only if $\forall\mathscr{P}.\ \max_{p\in\mathbb{P}^{\mathscr{P}}I_1}\mu(p) \ngtr \max_{\mathbb{P}_{I2}^{\mathscr{P}}}\mu(p)$. The policy I_1 **is more powerful than policy** I_2 under μ if and only if I_1 is at least as powerful as I_2 under μ, and $\exists\mathscr{P}\ .\max_{p\in\mathbb{P}_{I1}^{\mathscr{P}}}\mu(p) > \max_{\mathbb{P}_{I2}^{\mathscr{P}}}\mu(p)$.

Note that flexibility is purely a structural notion, whereas a metric must be fixed to discuss power. Moreover, it is clear that if I_1 is at least as flexible as I_2, then I_1 is at least as powerful as I_2 under *any* metric. In addition, if I_1 is at least as flexible as I_2 and I_1 is more powerful than I_2 under some metric, then I_1 must be more flexible than I_2. For a simple example comparing policies, consider I_1 = "no directly recursive calls may be inlined" and I_2 = "only calls to leaf procedures may be inlined". Since by definition a leaf procedure contains no call site, and a directly recursive call implies a nonleaf procedure, clearly I_1 is at least as flexible as I_2. It is also easy to see that I_1 is, in fact, more flexible than I_2.

3. Power of inlining original versions

There is an implicit assumption in previous inlining work—that the current version of callee's code is inlined. This is not always the best policy, as we next show. We consider the following three inlining policies.

cv—The *c*urrent *v*ersions of the caller and callee are always used.

ov—The *o*riginal *v*ersion of the callee and the current version of the caller are always used.

av—*A*ny *v*ersions of the caller and callee may be used.

Clearly, *av*-inlining is the least restrictive policy. In this section, we show that *ov*-inlining is at least as flexible as *av*-inlining. We then show that *ov*-inlining is more powerful than *cv*-inlining under a reasonable metric.

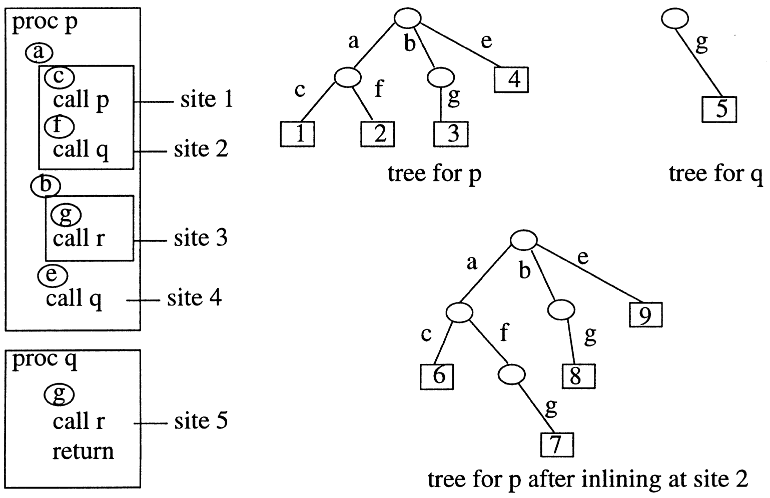

Fig. 1. Block-nesting graph and the effect of inlining. On the right, *mb*-nodes are depicted as circles, and *c*-nodes as squares. Within each *c*-node, the associated call-site number is shown. Arc labels a, b, and e represent the call sites of the original version of p.

3.1. Block nesting graph

In order to compare the flexibility of one policy with that of another, we represent the program with its *block nesting graph*. This graph abstracts the structure of each version of each procedure, showing how the *major blocks* in each version nest (see Fig. 1). Each procedure initially has only a single major block, which includes all its code. Whenever a call site is inlined, all major blocks within the callee are duplicated within the caller.

The block nesting graph is a forest, with a tree T_i^v for each version v of each procedure P_i. The tree has two kinds of nodes, *mb*-nodes that represent major blocks, and *c*-nodes that represent call sites. The *c*-nodes may only be leaves, whereas the *mb*-nodes may be internal nodes or leaves. If a call site c (or another major block M') is directly nested within a major block M, then the *mb*-node for M has the node for c (or M', respectively) as a child. Each arc in the tree is labeled with a call site in the original program, and each *c*-node is in 1-1 correspondence with a call site in the version being represented and is labeled thus.

The initial forest contains trees with a single *mb*-node and as many *c*-nodes as call sites. Suppose a new version v'_i of P_i is formed by taking version v_i of P_i and, at site s, inlining version v_j of P_j. This inlining step is reflected in the block nesting graph by adding a new tree $T_i^{v'}$. The new tree is constructed by replacing the *c*-node labeled s in a copy of $T_i^{v_i}$ by a copy of $T_j^{v_j}$ and relabeling all the *c*- nodes derived from $T_j^{v_i}$. The bottom-right part of Fig. 1 shows the new tree added to the block nesting graph as a result of inlining at site 2.

3.2. Power of ov-inlining

Using the abstraction of block nesting graphs, we can establish the following theorem:

Theorem 1. *The* ov-*inlining policy is at least as flexible as the* av-*inlining policy.*

Proof sketch: Consider any final program $\mathscr{P}$ obtained using *av*-inlining. We show how the same program can be obtained using *ov*-inlining. First, consider the subset of the block nesting diagram for $\mathscr{P}$ that contains only one tree per procedure—the tree for the version of the procedure that appears in $\mathscr{P}$. Observe that, if we can construct a program $\mathscr{Q}$ whose block nesting diagram matches that of $\mathscr{P}$, then $\mathscr{Q}$ and $\mathscr{P}$ are the same (structurally, as well as semantically). (We require the trees be isomorphic and arc labels, but not node labels, must match.) However, any required tree is easily obtained by "growing" it top down—wherever the desired tree has an *mb*-node and the current tree has a *c*-node, perform an *ov*-inlining operation on the call site corresponding to the *c*-node. This can be repeated until the desired tree is obtained.[2] For a precise proof, see [13].

This result has some practical implications: an inliner need only support *ov*-inlining, but by "replaying" an appropriate sequence of *ov*-inlining steps, as determined by the block nesting diagram, it can simulate any inlining operation. This implies that an index structure could be pre-computed to permit efficient access to the (static) original procedure bodies. For programs of modest size, all required data may even fit in main memory, since only the block nesting diagram, call graph, and the smaller original procedure bodies are required.

Next, we consider a figure of merit μ for inlined programs, which rewards the reduction of function calls executed at runtime, while requiring the program's size not be excessive. Let $f(\mathscr{P})$ denote the number of function calls performed when a program $\mathscr{P}$ is executed with a prespecified input. Let $\sigma(\mathscr{P})$ denote the size of $\mathscr{P}$, and let $\mathscr{P}_{\text{orig}}$ be the original program whose inlined form is $\mathscr{P}$.

$$\mu(\mathcal{P}) = \begin{cases} -\infty & \text{if } \sigma(\mathscr{P}) > S \\ f(\mathscr{P}_{\text{orig}}) - f(\mathscr{P}) & \text{otherwise} \end{cases}$$

Theorem 2. *Under μ,* ov*-inlining is more powerful than* cv*-inlining.*

To prove the theorem, consider the program in Fig. 2. Clearly, the call site within `f` should be inlined. Inlining the original version twice leaves us with a program where `f(x)` invokes `f(x-3)` , with `f` now thrice its original size. If this is the maximum code growth permitted, *cv*-inlining is only able to inline once at the recursive call site. (The next inlining would leave `f` at quadruple its original size.) Thus, the best *cv*-inlined program would perform about 50% more calls than the *ov*-inlined program.

Note: In [13], we find an example where the call ratio of the best *cv*-inlined program to the best *ov*-inlined program is $\approx N/\log N$.

3.3. Importance of recursion

It is not coincidence that the program in Fig. 2 was recursive. For non-recursive programs, we have the following result:

Theorem 3. *For a non-recursive program,* cv*-inlining is at least as flexible as* av*-inlining.*

[2] In our experiments, we have used this approach to simulate *cv*-inlining with *ov*-inlining. Each *cv*-inlining is replaced by the sequence of *ov*-inlining operations that makes the same additions to the block nesting graph.

```
proc f(x)
begin
  if x=1 then
    return 0
  else
    y = f(x-1)
    return 1 - y
end
proc main
begin
  z = f(300)
end
```

Fig. 2. Program used to demonstrate that *ov*-inlining is more powerful than *cv*-inlining.

Proof sketch: Consider some non-recursive program $\mathscr{P}$, from which *av*-inlining has obtained $\mathscr{P}'$. Since the call graph of $\mathscr{P}$ is acyclic, its procedures can be ordered topologically; thus, a procedure precedes all procedures it calls (directly or transitively).

We can obtain the block-nesting diagram for $\mathscr{P}'$ (and thus $\mathscr{P}'$ itself) by working exclusively on one tree at a time. First, we inline call sites within the topologically first procedure, until the final form of the tree (procedure) is achieved. It will be, since the current version of each callee is the original version. Then we proceed similarly to the topologically second procedure, and so forth. For every inlining operation, the callee has only one version, since it appears topologically after the caller.

4. A multi-version inlining technique

Having shown theoretical advantages to the policy of *ov*-inlining, we now show, as a major case study, the benefit of greater flexibility in an inliner. In this case study, we will focus on inlining techniques whose goal is to maximize the reduction in function calls when the program is run with a fixed input, without exceeding a code-size bound. Scheifler has shown that finding a strategy for choosing a sequence of inlining steps to attain our goal is NP-hard.[3] We examine suitable greedy heuristics in Section 4.2 and 4.3. Note that our goal has previously been used in [2–4, 6].

We believe that if greater flexibility leads to improvements with this inlining goal, we should also expect improvements from other goals—for instance, inliners that take code-improvement possibilities or cache performance into account should also benefit. Clearly, though, we cannot test this thesis for all reasonable inlining goals; thus, we choose the above goal for its simplicity and widespread use.

[3] It should be noted that although the hardness result was established under *cv*-inlining, the result is general since the result was established for the case of nonrecursive programs.

An obvious problem arises at this point: how are we to know the effect of an inlining operation? It is relatively easy to gather profile data to obtain, for a given input, the count of the number of calls performed at each call site in the original program. However, except when inlining a leaf procedure, new call sites will be created. To compute accurately the number of calls that will be performed at these sites, we would need the complete call history upon reaching every call site in the original program. This is prohibitively expensive, as discussed in [4, 6]. Instead of attempting to predict the effect of an inlining step exactly, we estimate its effect using profile data with a one-level history: the number of times each procedure was originally entered, and the number of times each call site was taken. The fundamental assumption behind the estimation technique is that the behavior of a procedure can be approximated well by an "average behavior" that is independent of where the procedure is called from. This corresponds to the *constant ratios assumption* made in [6]. The use of the assumption is not new, and has been validated in [6, 13]. The estimation technique, however, is novel and is based on a probabilistic model described below. The model itself resembles the stochastic program model described for flowcharts in [14, pages 439–448].

4.1. Probabilistic model

In this model we associate, with each call site a, an *expected execution frequency* denoted by ρ_a, that corresponds to the average number of times a is reached per invocation of *caller* (a). Observe that, in programs free of loop constructs, the expected execution frequency of site a coincides with the conditional probability of reaching a on any invocation of *caller* (a). We also maintain the *expected call frequency* m_{ij} for every pair of procedures P_i and P_j, which is the expected number of times P_i *directly invokes* P_j (from statically known call sites) for each invocation of P_i. Clearly, $m_{ij} = \Sigma_a \rho_a$ for sites a in the program with *caller* (a) = i and *callee* (a) = j. For a n-procedure program, the expected call frequencies m_{ij} form an $n \times n$ matrix $\mathbf{M}$, the *direct call* matrix of the program. Each row of $\mathbf{M}$ is a vector that describes the calling behavior of one invocation of (the current version of) a particular procedure.

The total number of times procedure P_i is entered, denoted by v_i, is divided into *direct* entries, from known call sites, and *indirect* entries, from SYSTEM. The latter do not vary with inlining, unlike the former. The number of indirect entries to P_i is denoted by s_i. Let row vectors $v = (v_1, v_2, \ldots, v_n)$ and $\mathbf{s} = (s_1, s_2, \ldots, s_n)$. From the definition of $\mathbf{M}$, v and $\mathbf{s}$, we have $v = v \times \mathbf{M} + \mathbf{s}$ and hence

$$\mathrm{v} = s \times (\mathbf{I} - M)^{-1}. \tag{1}$$

Refer to Fig. 3 for an example, where

$$\mathrm{v} = \mathrm{v} \times \begin{pmatrix} 0.75 & 1.5 & 0 \\ 0.165779 & 0 & 2 \\ 0 & 0 & 0 \end{pmatrix} + (1,\ 2,\ 0),\ \text{and its solution is}\ \mathrm{v} = (1000,\ 1502,\ 3004).$$

It is easy to see that $v = \mathbf{s} \times (\mathbf{I} + \mathbf{M} + \mathbf{M}^2 + \cdots)$. Clearly, the infinite sum in the above equation converges whenever the program execution terminates. We say that the matrix $\mathbf{U} = (\mathbf{I} - \mathbf{M})^{-1} = \Sigma_{k=0}^{\infty} \mathbf{M}^k$ is the *indirect call* matrix of the program: the $(i, j)^{\text{th}}$ entry u_{ij}

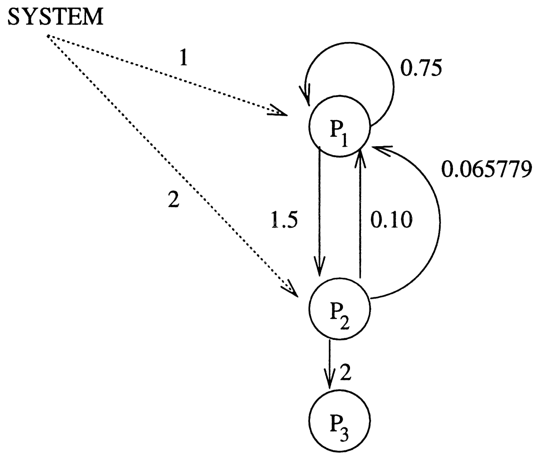

Fig. 3. Top: Call graph for a program with three procedures. Dotted arcs represent indirect calls from the system, made the indicated number of times. Solid arcs represent direct calls with the shown ρ values.

represents the expected number of times P_j will be invoked (directly or indirectly) for every invocation of P_i from SYSTEM.

4.2. Estimating the effect of an inlining operation

When a copy of a call site is created by inlining, the model assumes that the new call site maintains the expected execution frequency with respect to the new (outermost) nested block that is introduced into the block nesting diagram. More formally, consider inlining at the call site a such that *caller* $(a) = i$ and *callee*$(a) = j$. For each call site b in P_j, a new call site c is now introduced in the new version of P_i, with $\rho_c = \rho_a \rho_b$. For instance, in Fig. 1, if all call sites a in the original program had $\rho_a = 0.5$, we now have $\rho_4 = \rho_5 = \rho_9 = 0.5$, $\rho_1 = \rho_6 = \rho_2 = \rho_3 = \rho_8 = 0.\ 25$, $\rho_7 = 0.125$. Call sites 1–4 are not part of the program, since they are not within the current version of procedure `p`.

The benefit due to an inlining operation is the number of calls it saves. At any point while inlining, there may be several potential inlining operations possible, and one must be selected as the next operation. Using a greedy strategy, at each step we select the inlining operation (the arc to be inlined, and the version to be used) that results in the largest number of calls saved per unit increase in code space. For the inlining process to be fast, efficient selection of the best inlining operation is crucial. Therefore, direct use of (1) is precluded, and we next derive an efficient procedure to determine the best operation and its effect, under both *ov*-inlining and *cv*-inlining policies.

Consider inlining an arc $a = (P_i, P_j)$. The effect on $\mathbf{M}$ is that the i^{th} row of $\mathbf{M}$ changes to reflect the addition of any new call sites and the removal of the old call site. Specifically, under the *cv*-inlining policy, we have

$$\mathbf{M}_{\text{new}} = \mathbf{M} + \rho_a \mathbf{1}_{i,j} \times \mathbf{M} - \rho_a \mathbf{1}_{i,j}$$

where $\mathbf{1}_{i,j}$ is a matrix that is 1 at the $(i, j)^{\text{th}}$ element and zero elsewhere.

Under the *ov*-inlining policy, we have:

$$\mathbf{M}_{\text{new}} = \mathbf{M} + \rho_a \mathbf{1}_{i,j} \times \mathbf{M}_{\text{orig}} - \rho_a \mathbf{1}_{i,j} \tag{2}$$

where $\mathbf{M}_{\text{orig}}$ denotes the direct call matrix of the original program (before any inlining). From (2) and the univariant $\mathbf{U}^{-1} = (\mathbf{I} - \mathbf{M})$ we obtain

$$\mathbf{U}^{-1}_{\text{new}} = \mathbf{U}^{-1} + \rho_a \mathbf{1}_{i,j} \times \mathbf{U}^{-1}_{\text{orig}}.$$

Since inlining operations do not affect the termination characteristics of programs, the matrix $\mathbf{U}_{\text{new}} = (\mathbf{I} - \mathbf{M}_{\text{new}})^{-1}$ exists. Next, let $\mathbf{U}_{\text{new}} = \mathbf{U} \times \mathbf{B}$. The matrix $\mathbf{B}$ essentially captures the change in $\mathbf{U}$ due to the inlining step. Note that $\mathbf{B} = \mathbf{U}^{-1} \times \mathbf{U}_{\text{new}}$ and hence $\mathbf{B}$ is invertible. Since $\mathbf{U}^{-1}_{\text{new}} = \mathbf{B}^{-1} \times \mathbf{U}^{-1} = \mathbf{U}^{-1} + \rho_a 1_{i,\,j} \times \mathbf{U}^{-1}_{\text{orig}}$, we have

$$\mathbf{B}^{-1} = \mathbf{I} + \rho_a \mathbf{1}_{i,j} \times \mathbf{U}^{-1}_{\text{orig}} \times \mathbf{U}$$

Now, let $\mathbf{A} = \mathbf{U}^{-1}_{\text{orig}} \times \mathbf{U}$. Hence

$$\mathbf{B}^{-1} = (\mathbf{I} + \rho_a \mathbf{1}_{i,j} \times \mathbf{A}). \tag{3}$$

and

$$\mathbf{B} = (\mathbf{I} + \rho_a \mathbf{1}_{i,j} \times \mathbf{A})^{-1}. \tag{4}$$

We will now use the identity that

$$(\mathbf{I} + \mathbf{1}_{k,l} \times \mathbf{D})^{-1} = (\mathbf{I} - \left(\frac{1}{1 + d_{lk}}\right)\mathbf{1}_{k,l} \times \mathbf{D})$$

to get

$$\mathbf{I} - \mathbf{B} = \left(\frac{\rho_a}{1 + \rho_a a_{ji}}\right)\mathbf{1}_{i,j} \times \mathbf{A}.$$

Note that $(1 + \rho_a a_{ji})$ cannot be zero, otherwise the entire i^{th} column of $\mathbf{B}^{-1}$ would have been zero (see Eq. (3)) and hence $\mathbf{B}^{-1}$ would not have been invertible.

Let $|v| = \Sigma_i v_i$. The benefit of an inlining step is simply $|v| - |v_{\text{new}}|$. Let $\mathbf{e}$ be a column-vector of length n such that $\mathbf{e}_i = 1$ for all i.

Then, using the definitions of $\mathbf{U}$ and $\mathbf{U}_{\text{new}}$, we have

$$\begin{aligned} |\mathrm{v}| - |\mathrm{v}_{\text{new}}| &= \mathrm{v} \times \mathbf{e} - \mathrm{v}_{\text{new}} \times \mathbf{e} \\ &= \mathrm{v} \times (\mathbf{I} - \mathbf{B}) \times \mathbf{e}. \end{aligned}$$

Now, by definition of $(\mathbf{I} - \mathbf{B})$ and $\mathbf{e}$, $(\mathbf{I} - \mathbf{B}) \times \mathbf{e}$ is a column-vector $\mathbf{f}$ of length n such that

$$f_l = \begin{cases} \dfrac{\rho_a \sum_k a_{jk}}{1 + \rho_a a_{ji}} & \text{if } l = i \\ 0 & \text{otherwise.} \end{cases}$$

Continuing, the number of calls saved

$$\begin{aligned}|\nu| - |\nu_{\text{new}} &= \nu \times (\mathbf{I} - \mathbf{B}) \times \mathbf{e} \\ &= f_i \nu_i \\ &= \left(\frac{\rho_a \sum_k a_{jk}}{1 + \rho_a a_{ji}} \right) \nu_i.\end{aligned}$$

Note that the matrix $\mathbf{A} = \mathbf{U} \times \mathbf{U}_{\text{orig}}^{-1}$ need be computed only once per inlining step. At that time, we can also precompute $\Sigma_k a_{jk}$ for each value of j. Thus, at every inlining step, the number of calls saved by inlining each arc can be computed in time proportional to the number of arcs. Moreover, we need compute only $\mathbf{A}$ at each step, and $\mathbf{A}_{\text{new}}$ can be computed directly from $\mathbf{B}$ and $\mathbf{A}$. In fact, using Eq. (4) and the definitions of $\mathbf{U}$ and $\mathbf{U}_{\text{new}}$, $\mathbf{A}_{\text{new}} = \mathbf{A} - \mathbf{A} \times \rho_a/(1 + \rho_a a_{ji})\mathbf{1}_{i,\,j} \times \mathbf{A}$ and can be computed in $O(n^2)$ time, since its $(k, k')^{\text{th}}$ entry is $a_{kk'} - \rho_a a_{jk'} a_{ki}/(1 + \rho_a a_{ji})$. The updated value of v can likewise be calculated as $v_{\text{new}} = v - v \times \rho_a/(1 + \rho_a a_{ji})$; note that these matrix products are special cases that can be quickly calculated. Thus the choice of arc to be inlined can be made in time O(no. of arcs) + $O(n^2)$.

Simplification for the special case of cv*-inlining:* Note that for *cv*-inlining, the above procedure still holds with $A = \mathbf{U} \times \mathbf{U}^{-1} = \mathbf{I}$. So $\Sigma_k a_{jk} = 1$. If we are inlining a non-recursive arc (i.e., $i \neq j$) then $a_{ij} = 0$ and hence the number of calls saved is $\rho_a\ v_i$. If the arc inlined is recursive $(i = j)$, then $a_{ij} = 1$ and hence the number of calls saved is $[\rho_a/(1 + \rho_a)]v_i$. Thus we get Scheifler's equations.

4.3. A hybrid inlining strategy

It is straightforward to obtain a greedy strategy that considers only *ov*-inlining steps, based on the above procedure to estimate the effect of an inlining operation. There is an additional difficulty, however. A *cv*-inlining operation always leads to fewer calls at runtime. This does not hold for *ov*-inlining, where an *ov*-inlining operation can replace a call to a highly optimized procedure by its unoptimized code. In many such cases, an *ov*-inlining operation will actually lead to more calls at runtime—though further *ov*-inlining would lead to a superior result. These small, local moves in the state space can result in the greedy strategy becoming trapped at a local maximum. In such cases [15] it helps to add some lookahead capability, so the strategy can greedily choose a small sequence of operations.

We can exploit the fact that *cv*-inlining always leads to fewer calls (in reality, and in predictions). The greedy strategy now considers both *ov*- and *cv*-inlining at every call site. Recall that the proof of Theorem 1 shows how the *cv*-inlining of an arc a can be simulated by a sequence of *ov*-inlining operations, beginning with the *ov*-inlining of a. Therefore, by considering *cv*-inlining operations as representing sequences of *ov*-inlining operations, our solution gives the greedy strategy some lookahead along certain chosen paths through the state space. As long as there is sufficient code space to simulate the *cv*-inlining of some arc $a = (P_i,$

```
$\mathbf{A}$ := $\mathbf{I}$
for j := 1 to n
  Arow[j] := $\sum_{k=1}^{n} a_{jk}$
currentExpansion := 0
repeat
  /* find best inlining operation under $\mu$ */
  best := 0
  for each arc $a = (P_i, P_j)$ in the current call graph
    ovBenefit :=
      $\begin{cases} 0 & \text{if } \sigma_{\mathrm{ov}}(P_j) + \texttt{currentExpansion} > \texttt{MaxExpansion} \\ \rho_a \texttt{Arow[j]} \nu_i/(1+\rho_a a_{ji}) & \text{otherwise} \end{cases}$
    ovCost := $\sigma_{\mathrm{ov}}(P_j)$; ovRatio := ovBenefit / ovCost

    cvBenefit :=
      $\begin{cases} 0 & \text{if } \sigma_{\mathrm{cv}}(P_j) + \texttt{currentExpansion} > \texttt{MaxExpansion} \\ \rho_a \nu_i/(1+\rho_a) & \text{otherwise, if } i = j \\ \rho_a \nu_i & \text{otherwise, if } i \neq j \end{cases}$
    cvCost := $\sigma_{\mathrm{cv}}(P_j)$; cvRatio := cvBenefit / cvCost

    if best $<$ cvRatio then
      best := cvRatio; bestArc := $a$; bestKind := cv
    if best $<$ ovRatio then
      best := ovRatio; bestArc := $a$; bestKind := ov
  if best = 0 then exit program

  if bestKind = ov then
    ovSequence := $\langle$ bestArc $\rangle$
    currentExpansion := currentExpansion + ovCost
  else /* get appropriate ov sequence, sketched in Theorem 1 */
    ovSequence := simulateCvOperation(blockNestingDiagram,bestArc)
    currentExpansion := currentExpansion + cvCost

  for each arc $a = (P_i, P_y)$ in ovSequence
    perform indicated ov-inlining on program
    update blockNestingDiagram and current call graph
    $\nu$ := $\nu - \nu \times \frac{\rho_a}{1+\rho_a a_{yi}} \mathbf{1}_{i,y} \times \mathbf{A}$
    $\sigma_{\mathrm{cv}}(P_i)$ := $\sigma_{\mathrm{cv}}(P_i) + \sigma_{\mathrm{ov}}(P_y)$
    /* update $\mathbf{A}$, making $\mathbf{A}'$ with entries $a'_{ij}$ as follows */
    for k = 1 to n
      for x = 1 to n
        $a'_{kx}$ := $a_{kx} - (\rho_a a_{yx} a_{ki})/(1+\rho_a a_{yi})$
    $\mathbf{A}$ := $\mathbf{A}'$
    recalculate Arow entries, as done on second and third lines
forever
```

Fig. 4. Pseudocode for hybrid inlining using a greedy strategy and a fixed code-size expansion. $\sigma_{ov}(p)$ and $\sigma_{cv}(p)$ denote, respectively, the original and current sizes of procedure p.

P_j) for which $\rho_a\ v_i > 0$, we cannot become trapped at a local maximum. We refer to this form of inlining as *hybrid* inlining.

Fig. 4 provides pseudocode for hybrid inlining, summarizing the preceding discussion.

4.4. Experimental evaluation

We now continue our case study by measuring the effect of the additional flexibility offered by an *ov*-inlining policy. We compare the performance of the hybrid inlining strategy described in the previous section with that of the greedy *ov*-strategy (i.e., without lookahead), the greedy *cv*-strategy due to Scheifler, the profile-guided technique presented in [4] and the static technique used in GCC. It should be re-stated that none of the above techniques make inlining decisions based on secondary effects such as additional specialization opportunities.

For this case study, recall that our goal is to minimise the number of dynamic calls predicted by the probabilistic model, without exceeding a fixed code-size increase prediction. Success toward this goal is thus measured. The accuracy of the predictions has been examined in [6, 13]; while there are some inaccuracies with the predictions, they do not affect our conclusions, as explained below. In [13] we also examine the greedy strategy with *cv*-, *ov*-, and hybrid inlining, using exact call information instead of relying on predictions made using the probabilistic model. We observe a typical improvement of 1–3% over the probabilistic model for all policies, with a few instances where the more accurate information *decreased* overall performance. (This is not really surprising since the greedy strategy is suboptimal.)

To gather data, an inliner was designed for the C language. The GNU C compiler [7] was modified to emit a call graph for each separately compiled module and generate code to permit suitable profiling. The inliner then profiled the program, built a program-wide call graph, and emitted information on the desired inlining operations and the predicted number of calls executed. Inter-module inlining was permitted, as in [4].

We measured the effect of the various inlining techniques on both programmer-written and machine-generated C programs.[4] Programs in the first category include *Bison* (version 1.24), processing the grammar for GCC 2.7.2; *Flex* (version 2.5.2), creating the lexical analyzer for EQUALS [16]; and *Cccp*, the macro pre-processor for GCC 2.5.8, processing `cccp.c`.

Programs in the second category were produced by the EQUALS compiler for the following functional programs: *Event* and *Ida* adapted from Hartel's benchmark suite [17]; two theorem provers, *ODProv* [18] and *PCProv*; *Pascal*, an interpreter for a subset of Pascal; and *FFT*, which implements a Fast Fourier Transform algorithm. All the functional programs except *FFT* make extensive use of lazy evaluation [19]. This reduces the number of calls that procedures make directly to one another; instead, procedures are often invoked by the runtime system when closures are entered. Since such calls cannot be inlined by any of the above techniques, in order to obtain meaningful results, we measure the reduction in the number of *inlinable* calls.

[4] Recursion is not likely to predominate in the standard benchmark suites; as our results in this paper show, the difference between *cv*- and *ov*-inlining policies becomes most important with recursion.

Table 1
Inlinable calls removed after 5% code size expansion

Benchmark	GCC	IMPACT	CV	OV	Hybrid
Bison	17.7	31.7	**57.1**	55.8	**57.1**
Cccp	0.3	39.9	**50.2**	47.1	**50.2**
Event	0.0	45.2	**76.3**	**76.3**	**76.3**
FFT	0.0	1.7	64.4	**69.2**	**69.2**
Flex	1.0	39.1	**59.3**	59.0	**59.3**
Ida	0.0	73.4	**96.9**	76.9	**96.9**
ODProv	0.0	28.2	38.2	**39.8**	**39.8**
Pascal	5.5	89.0	**94.5**	90.8	90.8
PCProv	0.0	13.0	32.7	**36.7**	**36.7**
Geom. mean	—	27.5	59.6	58.8	**60.9**
Wins	0	0	6	4	8

The percentage of inlinable calls removed by the various inlining techniques with 5% and 20% increase in code size are given in Tables 1 and 2 respectively. In the tables, GCC denotes a static (not profile-based) inlining technique similar to the one used by GCC; IMPACT represents a technique, described later, that is similar to that presented in [4]; CV, OV and Hybrid denote the greedy strategy with *cv* policy (due to Scheifler), the greedy strategy with *ov* policy (without lookahead) and the hybrid strategy with *ov* policy and *cv* lookahead respectively. For each example program, the best results obtained are highlighted.

Note that GCC's inliner does not take profile information into account; rather, it either relies on programmer declarations, or inlines suitably small procedures within a module. We have followed the latter approach, and it is clear that profile information would have been helpful.

The inliner that was based on the technique discussed in [4] did not perform particularly well, especially on the recursive programs. This was anticipated, as the strategy does not inline recursive calls, which predominate in these benchmarks. An extension to their basic inliner, which preprocesses directly recursive calls, was sketched but apparently not used in [4]. Since it

Table 2
Inlinable calls removed after 20% code size expansion

Benchmark	GCC	IMPACT	CV	OV	Hybrid
Bison	17.7	58.4	**87.1**	75.2	**87.1**
Cccp	1.6	64.8	**72.3**	64.5	**72.3**
Event	10.2	80.9	**99.9**	**99.9**	**99.9**
FFT	0.0	3.5	84.0	**84.8**	**84.8**
Flex	1.4	58.2	**84.0**	76.7	**84.0**
Ida	0.0	74.0	**98.7**	77.0	**98.7**
ODProv	0.0	45.1	60.2	**62.4**	**62.4**
Pascal	5.5	89.0	**100.0**	**100.0**	**100.0**
PCProv	0.0	14.6	56.7	**63.6**	**63.6**
Geom. mean	—	40.1	81.0	77.1	**82.4**
Wins	0	0	6	5	9

is unclear how to deal with procedures containing several direct calls using this technique, we did not implement a preprocessing step. Instead, we implemented the algorithm sketched in [4, page 364]. This algorithm requires that we consider the procedures iteratively, from most to least frequently called. For each procedure, we inline all its call sites that invoke already-processed procedures, using the current versions of the invoked procedures. To enable comparison with other techniques, we do not permit any operation that causes the fixed code-size bound to be exceeded.

Examining the differences between the *ov*-, *cv*- and hybrid inliners, it can be seen that, despite the *ov* policy being more flexible than *cv*, suboptimal solutions and trapping at local maxima are serious problems unless lookahead is used. For instance, in *Ida* it did not matter much whether 5% or 20% code expansion was permitted with *ov*-inlining. In many other cases, *ov*-inlining received much less benefit than *cv*-inlining from a more generous code-size expansion.

Further, we see that hybrid inlining can improve on both *cv* and *ov*-inlining. For instance, Table 3 shows the performance of the different inlining strategies on small EQUALS programs from [13]. In that table, there are four examples on which hybrid inlining performed better than both *cv*- and *ov*-inlining. In these cases, a mixture of *cv*- and *ov*- steps has been used to achieve a result that would not have been found by either technique alone. On the other hand, we see in Table 1 a benchmark (*Pascal*) on which the greedy strategy did not benefit from the additional flexibility; rather, it lead to choices that, ultimately, were counterproductive. Additional lookahead would thus be useful, to further improve the hybrid inliner.

Discussion: The ultimate goal of a production inliner is typically to obtain faster code; for such inliners, a model will usually consider secondary effects—the ability to perform additional optimizations with the inlined code in a particular context, the effect of enlarging the body of a loop on cache performance [10], and so forth. While the overall reduction in running time may be the most appropriate measure for quantifying the *effectiveness* of an inlining technique in practice, the measure is architecture-dependent (due to the secondary effects themselves) and hence is inappropriate for *comparing* diverse inlining techniques. In contrast, the percentage reduction in function invocations is architecture-independent and repeatable, and hence is highly suited to compare different inlining techniques. In fact, the reduction in function calls often is a good predictor of the reduction in running time, as discussed below.

Table 3
Inlinable calls removed after 200% code expansion

Benchmark	CV	OV	Hybrid
Euler	94.6	75.6	**95.5**
MatMult	94.7	74.9	**94.8**
MergeSort	68.6	70.2	**71.8**
Nfib	55.6	**65.4**	**65.4**
NumInt	77.8	74.3	**78.7**
Queens	**96.6**	75.9	**96.6**
Sieve	93.0	**93.2**	**93.2**
Tak	62.1	**68.4**	**68.4**

The relationship between execution times and the number of calls performed has been examined in [8]; in that study, an inliner that considered many factors still found that the most significant factor was due to the reduction in call and return operation time [8, page 94]. More interestingly, the paper showed that code-size expansion did not need to be avoided; the larger inlined programs and uninlined programs had similar virtual-memory behavior. However, it should be noted that their policy forbade inlining directly recursive calls, and for most programs their inliner usually did not even double code size [8, Fig. 15].

Despite this paper's focus on comparing techniques, for which measurement of inlining's direct effects was appropriate, indirect effects should nevertheless be considered in future production inliners. Running times may be affected by such indirect effects as changes in virtual memory and cache behavior, possible performance degradation due to compiler shortcomings (such as limitations in a compiler's register allocation technique [8]) as well as performance gains due to opportunities for aggressive specialization. Therefore, a simplified model ignoring these effects is not ideal for future production inliners. Moreover, such inliners *must* take advantage of profile information whenever available, since profile-based inlining techniques significantly outperform static techniques. It should be noted that profile-based inliners must be able to predict the profile behavior of new calls that are introduced as a result of the inlining operations themselves. In this article, we presented a profile-based inlining framework for any version inlining, with profile prediction, while modeling only direct effects. Incorporation of various indirect effects into our inlining model remains an interesting open problem.

5. Evaluating inlining techniques: a summary

Many inlining techniques have been proposed in the past literature, and their relative merits have been established empirically. In this paper, we propose a non-empiric scheme to compare inlining techniques, by decomposing a technique into its *policy*—rules that restrict the possible set of inlining operations—and its *strategy*—rules that specify which allowed operation, if any, should be performed next.

We define two measures to compare inlining techniques: the *flexibility* and the *power* of an inlining policy. While flexibility measures the set of all programs derivable using some policy, power is defined with respect to some abstract performance metric and reflects the *best* inlined programs that can be derived using that policy.

Observe that during inlining many semantically equivalent versions of a procedure are created. At every inlining step, a call to a procedure can be replaced with the code for any one of these equivalent versions. The policy used in currently known inlining techniques restricts attention to the most recent (i.e., *current*) version of a procedure's code, and is called the *cv* policy. We show that the above policy is too restrictive.

In particular, we consider two other policies: the *av* policy which allows *any* previously derived version of a procedure to be inlined, and the *ov* policy which allows only the *original* version of a procedure to be inlined. We show, using our notions of flexibility and power, that

- The *ov* policy is as powerful as the *av* policy, under any metric.

- On non-recursive programs, the *cv* policy is as powerful as the *av* policy, under any metric.
- There are recursive programs, and reasonable metrics, for which the *ov* policy is more powerful than the *cv* policy.

We then develop an inlining technique based on the *ov* policy and a greedy strategy previously used for *cv*-inlining. This greedy strategy requires profile information for the original program; as inlining changes the program, the profile behaviours of the inlined programs are estimated with a probabilistic model. We develop an efficient model for estimating the effect of an inlining step which naturally generalizes the model developed for *cv*-inlining.

It should be noted that, while every inlining step using the *cv* policy is guaranteed to improve the performance (according to the chosen metric), individual steps using the *ov* policy can actually lead to performance degradation. Hence, a greedy strategy with *ov* policy can be trapped at local maxima. To avoid this, we describe a *hybrid* policy, which considers both the original and current versions at each inlining step.

We experimentally evaluate the different techniques based on *cv*, *ov* and the hybrid policies, using the "function calls removed without code explosion" metric. Trapping at local maxima was frequently observed for pure *ov*-inlining (without lookahead), both on recursive and non-recursive programs. Pure *cv*-inlining, which corresponds to the techniques described in previous works, displays suboptimal behavior on many recursive programs. We find that hybrid inlining often outperforms either pure *ov*- or pure *cv*-inlining.

C.R. Ramakrishnan is currently an Assistant Professor in the Department of Computer Science, SUNY at Stony Brook. He holds an M.Sc. in Physics and an M.Sc. in Computer Science from Birla Institute of Technology and Science, Pilani, India, and a Ph.D. in Computer Science from SUNY at Stony Brook. His research interests are in the areas of logic and functional programming, optimizing compilers and formal verification.

Owen Kaser is currently an Assistant Professor in the Department of Mathematics, Statistics and Computer Science at the Saint John campus of the University of New Brunswick, Canada. He holds a BCSS from Acadia University, Wolfville, Canada, and M.S. and Ph.D. degrees in Computer Science from SUNY at Stony Brook. His research interests include parallel computing, algorithms and optimizing compilers.

Acknowledgements

This work was supported in part by NSF grants CCR9404921 and CDA 9303181 and NSERC grant OGP0155967.

References

[1] Appel AW. Compiling with Continuations, Cambridge: Cambridge University Press, UK, 1992.
[2] Bal HE, Tanenbaum AS. Language- and machine-independent global optimization on intermediate code. Journal of Computer Languages 1986;11:105–21.
[3] Ball JE. Program Improvement by the Selective Integration of Procedure Calls. PhD thesis. University of Rochester, 1982.

[4] Chang PP, Mahlke SA, Chen WY, Hwu Wen-mei W. Profile-guided automatic inline expansion for C programs. Software—Practice and Experience 1992;25:349–69.
[5] Peyton-Jones SL, Santos A. Compilation by transformation in the Glasgow Haskell compiler. In: Functional Programming, Glasgow '94. Workshops in Computing, Springer-Verlag, 1994:184–204.
[6] Scheifler RW. An analysis of inline substitution for a structured programming language. Communications of the ACM 1977;20(9):647–54.
[7] Stallman RM. 1993. Using and Porting GCC. Free Software Foundation. For version 2.5.
[8] Davidson JW, Holler AM. Subprogram inlining: A study of its effects on program execution time. IEEE Transactions on Software Engineering 1992;18:89–101.
[9] Cooper KD, Hall MW, Torczon L. An experiment with inline substitution. Software—Practice and Experience 1991;21:581–601.
[10] McFarling S. Procedure merging with instruction caches. In: Proc. of the SIGPLAN '91 Conference on Programming Language Design and Implementation, 1991:71–79.
[11] Kaser O, Ramakrishnan CR, Pawagi S. On the conversion of indirect to direct recursion. ACM Letters on Programming Languages and Systems 1993;2(14):151–64.
[12] Kaser O. Inlining to reduce stack space. In: International Symposium on Programming Language Implementation and Logis Programming (PLILP-93). Tallinn, Estonia, Springer-Verlag, LNCS 714, 1993.
[13] Kaser O. Computational Aspects of Inlining. PhD thesis. SUNY at Stony Brook, 1993.
[14] Deo N. Graph Theory with Applications to Engineering and Computer Science. Prentice Hall, 1974.
[15] Krishnamurthy B. An improved min-cut algorithm for partitioning VLSI networks. IEEE Transactions on Computers 1984;C-33:438–46.
[16] Kaser O, Ramakrishnan CR, Ramakrishnan IV, Sekar RC. EQUALS—a fast parallel implementation of a lazy language. Journal of Functional Programming 1997;7(2):183–215.
[17] Hartel PH, Langendoen KG. Benchmarking implementations of lazy functional languages. In: ACM Conference on Functional Programming Languages and Computer Architecture (FPCA). New York, ACM, 1993:341–349.
[18] O'Donnell MJ. Equational Logic as a Programming Language. In: Foundations of Computing. MIT Press, 1985.
[19] Peyton-Jones SL. The Implementation of Functional Programming Languages. Prentice-Hall, 1987.