Fast and Effective Procedure Inlining*

Oscar Waddell and R. Kent Dybvig

Indiana University Computer Science Department
Bloomington, IN 47405, USA

Abstract. The effectiveness of an inlining algorithm is determined not only by its ability to recognize inlining opportunities but also by its discretion in exercising those opportunities. This paper presents a new inlining algorithm for higher-order languages that combines simple analysis techniques with demand-driven online transformation to achieve consistent and often dramatic performance gains in fast linear time. The algorithm is shown to be as effective as and significantly faster than offline, analysis-intensive algorithms recently described in the literature.

1 Introduction

Inlining is an important optimization for programs that use procedural abstraction. By replacing a call with an inline expansion of the procedure body, inlining eliminates procedure linkage overhead and may uncover opportunities for further optimization. Unrestrained inlining, however, can lead to excessive code growth and excessive compile time. The effectiveness of an inlining algorithm is therefore determined not only by its ability to recognize inlining opportunities but also by its discretion in exercising those opportunities.

Recent research on procedure inlining for higher-order languages has focused on the use of offline polyvariant flow analyses to identify appropriate inlining opportunities [8,4]. Although reported performance gains are impressive, the cost and volatility of such analyses renders the method impractical for use in production compilers. This paper presents a new inlining algorithm for higher-order languages that combines simple analysis techniques with online transformation to achieve comparable or better performance gains in fast linear time.

Several factors contribute to the speed and effectiveness of our algorithm: First, it is polyvariant. Information about the arguments at a call site is used to decide whether inlining that call is cost-effective. Second, it is online. Inlining decisions are based, in part, on the effects of other optimizations performed by the algorithm: constant folding, copy propagation, and elimination of useless or unreachable code. Third, it is context-sensitive. The context in which an expression occurs is used to identify legal transformations. Finally, it is demand-driven. The operands of binding constructs and calls are processed as soon as but not sooner than they are needed, at which point the context in which they are used has been determined.

* This material is based on work supported in part by the National Science Foundation under grant number CDA-9312614.

Polyvariance permits aggressive inlining without excessive code growth. Code growth is usually limited by inlining only procedures whose size is less than a given threshold. Our polyvariant algorithm instead considers the size of the procedure when optimized for the arguments at the call site. For example, given the definition of `f` below, the algorithm may choose to inline `e2` in place of the call `(f 'key2)` even if the size of `f` as a whole exceeds the threshold.

```
(define f
  (lambda (x)
    (case x [(key1) e1] [(key2) e2] ... [(keyn) en])))
```

Online transformation provides our algorithm with more accurate information on which to base its inlining decisions. Offline algorithms, which separate analysis, inlining, and subsequent optimization, must estimate the effects of these optimizations. When their estimate is too conservative, inlining opportunities are missed; when their estimate is too liberal, code size increases. Our algorithm inlines aggressively while maintaining tighter control over code growth because it uses the actual optimized code to decide when inlining a call is cost-effective. In some cases, online transformation also provides more accurate control-flow information, which helps to identify additional inlining opportunities.

Contextual information allows the algorithm to determine when an expression can be discarded or processed in a more restricted mode. For example, in the expression `(seq e1 e2)`, `e1` is processed only for its effects. When processed for effect, certain provably terminating effect-free expressions are discarded. In languages such as C and Scheme it may be possible to determine the boolean value of an expression even when its actual value cannot be determined. This is sufficient, however, when processing the guard of a conditional expression.

Demand-driven processing of call operands allows our algorithm to defer processing until more contextual information is available. For example, suppose `f` is `(lambda (x y) (if (< x 5) x y))`. When attempting to inline the call `(f e1 e2)`, `e1` is not processed until it is needed to evaluate the test `(< x 5)`. If `e1` reduces to `3`, the reference to `y` is unreachable and the useless expression `e2` is processed for effect only. Eliminating useless or unreachable code saves processing time and also improves the accuracy of the information collected by the algorithm. For example, if variables originally referenced only within `e2` no longer appear referenced, their bindings can be processed for effect only.

The inlining algorithm is a source-to-source transformation that operates by a demand-driven traversal of the source program. The algorithm collects information about the size of the residual program and the use of variables in the residual program as it is produced. This information supports inlining decisions and elimination of useless code. Termination is achieved by detecting cycles in the call graph as it is computed and by bounding the amount of effort expended on inlining at each call site in the input program. Code growth is limited by aborting an inlining attempt as soon as the size of the residual code produced in that attempt exceeds a given threshold. Polyvariance is obtained by attempting to inline all calls, while relying on the effort bound and the size threshold to abort an attempt when it becomes too costly. Applicative order and sequencing of

$$
\begin{array}{rl}
e ::= & \texttt{(const } c\texttt{)} \mid \texttt{(ref } x\texttt{)} \mid \texttt{(primref } p\texttt{)} \mid \texttt{(call } e_0\ e_1\texttt{)} \\
\mid & \texttt{(if } e_1\ e_2\ e_3\texttt{)} \mid \texttt{(seq } e_1\ e_2\texttt{)} \mid \texttt{(assign } x\ e\texttt{)} \\
\mid & \texttt{(lambda (}x\texttt{) } e\texttt{)} \mid \texttt{(letrec ((}x_1\ e_1\texttt{)} \ldots \texttt{(}x_n\ e_n\texttt{)) } e\texttt{)}
\end{array}
$$

$$c \in \mathit{Const},\ p \in \mathit{Prim},\ x \in \mathit{Var},\ e \in \mathit{Exp}$$

Fig. 1. The core language.

effects are preserved by optimizing inlined calls as if they were equivalent `let` expressions. For example, suppose `f` is `(lambda (x y) e1)`. When inlining, the call `(f e2 e3)` is treated as the semantic equivalent of `(let ((x e2) (y e3)) e1)`. Values obtained by processing `e2` and `e3` are used when specializing `e1`, but effects in `e2` and `e3` are residualized at the point of call.

Inlining is copy propagation extended to `lambda` expressions. Just as copy propagation often enables constant folding, inlining enables β-reduction. The next section formally describes an inlining algorithm that exploits this observation. This algorithm is simplified in that it concentrates on recognizing inlining opportunities without attempting to select appropriately among them. Section 3 shows how the simplified algorithm is restrained to obtain the actual inlining algorithm. Section 4 discusses inlining of recursive procedures. Section 5 provides detailed measurements of the effect of the algorithm on execution time, code size, and compile time in a production compiler. Section 6 describes related work and compares our results with those obtained by offline, flow-analysis-based inlining algorithms. Section 7 summarizes our work.

2 Unrestrained Inlining Algorithm

The inlining algorithm is described as a continuation-passing style (CPS) interpreter with explicit store. The actual implementation is in direct style but otherwise follows the structure of the algorithm described here with the restraints and extensions described in Sects. 3 and 4.

The inlining algorithm, I, takes an input expression e, a context γ, an environment ρ, a continuation κ, and a store σ. The input expression is in the core language shown in Fig. 1. The algorithm returns a residual expression in the same core language. Domain equations are given in Fig. 2. Figure 3 presents the algorithm driver. Figure 4 shows the functions that implement the core of the algorithm. To simplify the presentation, `lambda` abstractions are restricted to a single formal parameter, and applications (`call`) are restricted to a single operand. Generalization to multiple-argument and variable-arity procedures is straightforward.

Contextual information (γ) is used to identify situations in which certain transformations are legal. Four contexts are distinguished: *test*, used when processing an expression for its boolean value, *effect*, used when processing an expression for side effects only, *value*, used when the actual value is needed, and

$$
\begin{array}{rcl}
\rho \in \mathit{Env} & = & \mathit{Var} \to \mathit{Var} \\
\sigma \in \mathit{Store} & = & (\mathit{Loc}_x \to \mathit{VarFlags}) \times (\mathit{Loc}_\gamma \to \mathit{CtxtFlags}) \times (\mathit{Loc}_e \to \mathit{Resid}) \\
\kappa \in \mathit{Cont} & = & \mathit{Exp} \to \mathit{Store} \to \mathit{Exp} \\
\mathit{Resid} & = & \mathit{Exp} \cup \{\mathsf{unvisited}\}) \\
s, r \in \mathit{VarFlags} & \subseteq & \{\mathsf{ref}, \mathsf{assign}\} \\
\mathit{CtxtFlags} & \subseteq & \{\mathsf{inlined}\} \\
op \in \mathit{Operand} & ::= & \mathsf{Opnd}(\mathit{Exp}, \mathit{Env}, \mathit{Loc}_e) \\
\mathit{Var} & ::= & \mathsf{Var}(\mathit{Identifier}, \mathit{Operand} \cup \{\mathsf{null}\}, \mathit{VarFlags}, \mathit{Loc}_x) \\
\gamma \in \mathit{Context} & ::= & \mathsf{Test} \mid \mathsf{Effect} \mid \mathsf{Value} \mid \mathsf{App}(\mathit{Operand}, \mathit{Context}, \mathit{Loc}\gamma)
\end{array}
$$

$$
l_e \in \mathit{Loc}_e, l_x \in \mathit{Loc}_x, l_\gamma \in \mathit{Loc}_\gamma \qquad \text{Locations}
$$

Fig. 2. Domain equations.

call, used when processing an expression in the operator position of an application. Naturally, side effects are conserved in all contexts. Test, effect, and value contexts are represented by the nullary constructors Test, Effect, and Value. A call context γ is represented by the constructor $\mathsf{App}(op, \gamma_1, l_\gamma)$ where op is an Opnd structure (described below) for the actual parameter at the call site, γ_1 is the context of the call itself, and $\sigma(l_\gamma)$ contains a flag that is set if this call is inlined. This flag is used when residualizing a call.

The environment ρ maps source-program variables to residual-program variables and is used to rename (α-convert) bound variables to prevent inadvertent capture when code is replicated by inlining. The initial environment ρ_0 is the identity function on variables. The store σ is a triple of functions that respectively map locations l_x to subsets of *VarFlags*, locations l_γ to subsets of *CtxtFlags*, and locations l_e to residual expressions in the core language or to the flag unvisited. As a notational convenience, we use subscripts on location labels to select the appropriate component of the store. Thus $\sigma(l_x)$ abbreviates $(\sigma \downarrow 1)(l_x)$ and $\sigma[l_\gamma \mapsto \emptyset]$ abbreviates $\langle \sigma \downarrow 1, (\sigma \downarrow 2)[l_\gamma \mapsto \emptyset], \sigma \downarrow 3 \rangle$.

Source- and residual-program variables are represented using the constructor $\mathsf{Var}(x, op, s, l_x)$, where x is the lexical identifier corresponding to the variable, op is an operand structure, and s and $\sigma(l_x)$ are subsets of *VarFlags* that describe uses of the variable in the source and residual program, respectively. The ref flag supports elimination of useless assignments and useless bindings. The assign flag is used to prevent copy propagation and inlining through assigned variables. The parser introduces Var structures for variables and initializes the flags in the initial store, σ_0. The final store contains flags for the residual program variables and may be used in place of σ_0 in a subsequent application of I. Where unambiguous we use x as a notational convenience for $\mathsf{Var}(x, op, s, l_x)$.

When a variable x is bound as the formal parameter of an inlined procedure or bound by a `letrec` expression, the op field of $\mathsf{Var}(x, op, s, l_x)$ contains an operand structure $\mathsf{Opnd}(e, \rho, l_e)$, where e is the source expression corresponding to the actual parameter or bound value, ρ is an environment binding free variables of

e, and l_e contains the residual code for e or unvisited if the operand has not yet been processed.

The remainder of this section explains how the algorithm handles the core forms of the language.

Constants. Handling constants is straightforward. When processed for boolean value or side effects, constant values are reduced to `true`, `false`, or `void`. This simplifies processing of conditional expressions and simplifies the construction of residual sequence expressions.

Sequences. The expression `(seq e1 e2)` discards the value of `e1` and returns the value of `e2`. Therefore, `e1` is processed in an Effect context, and `e2` is processed in the current context, γ. When processed for effect, constants, primitive references, `lambda` expressions, variable references, and effect-free primitive calls reduce to the constant `void`. Residual sequences (constructed by *seq*, Fig. 4) are nested only in the first position. This permits constant-time access (via *result*) to the result expression of a residual sequencing form.

Conditionals. The guard of a conditional expression is processed for its boolean value using a Test context. When processed for boolean value, non-false constants, primitive references, `lambda` expressions, and certain effect-free primitive calls reduce to the constant `true`. If the *result* of the residual guard expression is a constant, the flow of control is known at compile time, so a sequence of the guard (which may contain effects) and the selected branch is residualized.

Variable Assignments. If a local variable `x` is unreferenced, then an assignment `(assign x e1)` is useless and `e1` can be processed for effect. Because the algorithm may still be in the process of computing referenced flags for residual-program variables ($\sigma(l_{x'})$) when it encounters the assignment, it bases this decision on the conservative source-program referenced flags (s). If `x` is referenced in the source program, the expression `e1` is processed in Value context, the assign flag is set in $\sigma(l_{x'})$, and an assignment to the α-converted variable x' is residualized.

Procedure Calls. The treatment of `call` illustrates the demand-driven nature of our algorithm. The argument at a call site is not processed before the body of the called procedure. Rather, when the algorithm encounters a call, it constructs an App context that contains the current context (the context of the call itself) and an operand structure for the argument at the call site. The expression in operator position is then processed using the newly constructed call context.

The operand structure captures the current environment so that the argument can be processed on demand. Operand structures also play a role in cycle detection described in Sect. 3. The residual code is cached within the operand so that an argument is processed at most once. The *visit* procedure returns the cached residual code if present. Otherwise it processes the argument in the given context, caching the result. Operands are visited in Value context unless they are associated with unreferenced variables, in which case they are visited for effect.

Processing of arguments is deferred until the algorithm encounters references to the corresponding formal parameters or decides to residualize the call. If a call

$$I : \mathit{Exp} \to \mathit{Context} \to \mathit{Env} \to \mathit{Cont} \to \mathit{Store} \to \mathit{Exp}$$

$$I(\texttt{const}\ c)\gamma\rho\kappa = \begin{cases} \kappa(\texttt{const void}) \text{ if } \gamma = \mathsf{Effect} \\ \kappa(\texttt{const true}) \text{ if } \gamma = \mathsf{Test} \text{ and } c \neq \texttt{false} \\ \kappa(\texttt{const}\ c) \quad\ \text{otherwise} \end{cases}$$

$$I(\texttt{seq}\ e_1\ e_2)\gamma\rho\kappa = Ie_1\ \mathsf{Effect}\ \rho\lambda e_1'.\ Ie_2\gamma\rho\lambda e_2'.\ \kappa\ \mathit{seq}(e_1', e_2')$$

$$I(\texttt{if}\ e_1\ e_2\ e_3)\gamma\rho\kappa = Ie_1\ \mathsf{Test}\ \rho\kappa_1$$

$$\text{where}\ \ \kappa_1 e_1' = \begin{cases} Ie_2\gamma_1\rho\lambda e_2'.\ \kappa\ \mathit{seq}(e_1', e_2') \text{ if } \mathit{result}(e_1') = (\texttt{const true}) \\ Ie_3\gamma_1\rho\lambda e_3'.\ \kappa\ \mathit{seq}(e_1', e_3') \text{ if } \mathit{result}(e_1') = (\texttt{const false}) \\ Ie_2\gamma_1\rho\lambda e_2'.\ Ie_3\gamma_1\rho\kappa_2 \quad \text{otherwise, where} \\ \qquad \kappa_2 e_3' = \begin{cases} \kappa\ \mathit{seq}(e_1', e_2') \quad \text{if } e_2' = e_3' = (\texttt{const}\ c) \\ \kappa(\texttt{if}\ e_1'\ e_2'\ e_3') \text{ otherwise} \end{cases} \end{cases}$$

$$\gamma_1 = \begin{cases} \mathsf{Value} \text{ if } \gamma = \mathsf{App}(op, \gamma_1, l_\gamma) \\ \gamma \quad\ \text{otherwise} \end{cases}$$

$$I(\texttt{assign}\ x\ e)\gamma\rho\kappa\sigma = \begin{cases} Ie\ \mathsf{Effect}\ \gamma\rho\kappa\sigma \ \ \text{if } \mathsf{ref} \notin s, \text{ where } \rho(x) = \mathsf{Var}(x', op, s, l_{x'}) \\ Ie\ \mathsf{Value}\ \gamma\rho\kappa_1\sigma \text{ if } \mathsf{ref} \in s, \text{ where } \rho(x) = \mathsf{Var}(x', op, s, l_{x'}) \\ \qquad \kappa_1 e'\sigma_1 = \kappa\ \mathit{seq}((\texttt{assign}\ x'\ e'), (\texttt{const}\ c))\ \sigma_2 \\ \qquad \sigma_2 = \sigma_1[l_{x'} \mapsto \{\mathsf{assign}\} \cup \sigma_1(l_{x'})] \\ \qquad c = \texttt{true} \text{ if } \gamma = \mathsf{Test} \text{ else } c = \texttt{void} \end{cases}$$

$$I(\texttt{call}\ e_1\ e_2)\gamma\rho\kappa\sigma = Ie_1\gamma_1\rho\kappa_1\sigma_1, \text{ where}$$

$$\begin{aligned} op &= \mathsf{Opnd}(e_2, \rho, l_{e_2}) && l_{e_2} \text{ fresh} \\ \gamma_1 &= \mathsf{App}(op, \gamma, l_{\gamma_1}) && l_{\gamma_1} \text{ fresh} \\ \sigma_1 &= \sigma[l_{e_2} \mapsto \mathsf{unvisited},\ l_{\gamma_1} \mapsto \emptyset] \end{aligned}$$

$$\kappa_1 e_1'\sigma_2 = \begin{cases} \kappa e_1'\sigma_2 \qquad\qquad\qquad \text{if } \mathsf{inlined} \in \sigma_2(l_{\gamma_1}) \\ \mathit{visit}(op, \mathsf{Value}, \kappa_2, \sigma_2) \text{ otherwise, where} \\ \qquad \kappa_2 = \lambda e_2'.\ \kappa(\texttt{call}\ e_1'\ e_2') \end{cases}$$

$$I(\texttt{primref}\ p)\gamma\rho\kappa = \begin{cases} \kappa(\texttt{const true}) \qquad \text{if } \gamma = \mathsf{Test} \\ \kappa(\texttt{const void}) \qquad \text{if } \gamma = \mathsf{Effect} \\ \kappa(\texttt{primref}\ p) \qquad \text{if } \gamma = \mathsf{Value} \\ \mathit{fold}(\texttt{primref}\ p)\gamma\rho\kappa \text{ if } \gamma = \mathsf{App}(op, \gamma_1, l_\gamma) \end{cases}$$

$$I(\texttt{lambda}\ x\ e)\gamma\rho\kappa\sigma$$

$$= \begin{cases} \kappa(\texttt{const true})\sigma \qquad \text{if } \gamma = \mathsf{Test} \\ \kappa(\texttt{const void})\sigma \qquad \text{if } \gamma = \mathsf{Effect} \\ Ie\ \mathsf{Value}\ \rho_1\kappa_1\sigma_1 \qquad \text{if } \gamma = \mathsf{Value}, \text{ where } x \text{ abbreviates } \mathsf{Var}(x, \mathsf{null}, s, l_x) \\ \qquad x' \text{ abbreviates } \mathsf{Var}(x', \mathsf{null}, \sigma(l_x), l_{x'})\ x',\ l_{x'} \text{ fresh} \\ \qquad \rho_1 = \rho[x \mapsto x'] \\ \qquad \sigma_1 = \sigma[l_{x'} \mapsto \emptyset] \\ \qquad \kappa_1 = \lambda e'.\ \kappa(\texttt{lambda}\ x'\ e') \\ \mathit{fold}(\texttt{lambda}\ x\ e)\gamma\rho\kappa\sigma \text{ if } \gamma = \mathsf{App}(op, \gamma_1, l_\gamma) \end{cases}$$

Fig. 3. The inlining algorithm.

$$I(\texttt{ref}\ x)\gamma\rho\kappa\sigma = \begin{cases} \kappa(\texttt{const void})\sigma & \text{if } \gamma = \mathsf{Effect}, \text{ else} \\ \kappa(\texttt{ref}\ x')\sigma_1 & \text{if } op = \mathsf{null} \text{ or } \mathsf{assign} \in s \\ & \text{where } \rho(x) = \mathsf{Var}(x', op, s, l_{x'}) \\ & \qquad \sigma_1 = \sigma[l_{x'} \mapsto \{\mathsf{ref}\} \cup \sigma(l_{x'})] \\ \mathit{visit}(op, \mathsf{Value}, \kappa_1, \sigma) & \text{otherwise, where } \rho(x) = \mathsf{Var}(x', op, s, l_{x'}) \\ & \text{and } \kappa_1 = \lambda e\sigma_2.\ \mathit{copy}(\rho(x), \mathit{result}(e), \gamma, \kappa, \sigma_2) \end{cases}$$

$$\mathit{fold}(\texttt{primref}\ p)\mathsf{App}(op, \gamma_1, l_\gamma)\rho\kappa\sigma = \mathit{visit}(op, \mathsf{Value}, \kappa_1, \sigma)$$
$$\text{where } \kappa_1 e_1' \sigma_1 = \begin{cases} \kappa(\texttt{const}\ c')\sigma_2 & \text{if } \mathit{result}(e_1') = (\texttt{const}\ c) \text{ and } p(c) = c' \\ & \text{where } \sigma_2 = \sigma_1[l_\gamma \mapsto \{\mathsf{inlined}\} \cup \sigma_1(l_\gamma)] \\ \kappa(\texttt{primref}\ p)\sigma_1 & \text{otherwise} \end{cases}$$

$$\mathit{fold}(\texttt{lambda}\ x\ e)\mathsf{App}(op, \gamma_1, l_\gamma)\rho\kappa\sigma = Ie\gamma_1\rho_1\kappa_1\sigma_1$$
$$\begin{aligned} \text{where} \quad & x \text{ abbreviates } \mathsf{Var}(x, \mathsf{null}, s, l_x) \\ & x' \text{ abbreviates } \mathsf{Var}(x', op, \sigma(l_x), l_{x'}) \quad x', l_{x'} \text{ fresh} \\ & \rho_1 = \rho[x \mapsto x'] \\ & \sigma_1 = \sigma[l_{x'} \mapsto \emptyset] \\ & \kappa_1 e' \sigma_2 = \begin{cases} \mathit{visit}(op, \mathsf{Effect}, \kappa_2, \sigma_2) \text{ if } \mathsf{ref} \notin \sigma_2(l_{x'}) \text{ and } \mathsf{assign} \notin \sigma_2(l_{x'}) \\ \mathit{visit}(op, \mathsf{Effect}, \kappa_3, \sigma_2) \text{ if } \mathsf{ref} \notin \sigma_2(l_{x'}) \text{ and } \mathsf{assign} \in \sigma_2(l_{x'}) \\ \mathit{visit}(op, \mathsf{Value}, \kappa_3, \sigma_2) \text{ otherwise} \end{cases} \\ & \kappa_2 = \lambda e_1'\sigma_3.\ \kappa\ \mathit{seq}(e_1', e')\ \sigma_3[l_\gamma \mapsto \{\mathsf{inlined}\} \cup \sigma_3(l_\gamma)] \\ & \kappa_3 = \lambda e_1'\sigma_3.\ \kappa\ (\texttt{call}\ (\texttt{lambda}\ x'\ e')\ e_1')\ \sigma_3[l_\gamma \mapsto \{\mathsf{inlined}\} \cup \sigma_3(l_\gamma)] \end{aligned}$$

$$\mathit{copy}(\mathsf{Var}(x', op, s, l_{x'}), e, \gamma, \kappa, \sigma)$$
$$= \begin{cases} I(\texttt{const}\ c)\gamma\rho_0\kappa\sigma & \text{if } e = (\texttt{const}\ c) \\ \kappa(\texttt{ref}\ x_1)\sigma & \text{if } e = (\texttt{ref}\ x_1) \text{ and } \mathsf{assign} \notin s_1 \\ & \text{where } x_1 \text{ abbreviates } \mathsf{Var}(x_1, op_1, s_1, l_{x_1}) \\ \mathit{fold}\ e\gamma\rho_0\kappa\sigma & \text{if } \gamma = \mathsf{App}(op_1, \gamma_1, l_\gamma) \\ & \quad \text{and } e = (\texttt{primref}\ p) \\ & \quad \text{or } e = (\texttt{lambda}\ x_1\ e_1) \\ \kappa(\texttt{primref}\ p)\sigma & \text{if } \gamma = \mathsf{Value} \text{ and } e = (\texttt{primref}\ p) \\ \kappa(\texttt{const true})\sigma & \text{if } \gamma = \mathsf{Test} \text{ and } e = (\texttt{primref}\ p) \\ & \qquad \text{or } e = (\texttt{assign}\ x_1\ e_1) \\ & \qquad \text{or } e = (\texttt{lambda}\ x_1\ e_1) \\ \kappa(\texttt{ref}\ x')\sigma_1 & \text{otherwise, where } \sigma_1 = \sigma[x' \mapsto \{\mathsf{ref}\} \cup \sigma(l_{x'})] \end{cases}$$

$$\mathit{visit}(\mathsf{Opnd}(e, \rho, l_e), \gamma, \kappa, \sigma) = \begin{cases} Ie\gamma\rho\kappa_1\sigma & \text{if } \sigma(l_e) = \mathsf{unvisited} \\ & \text{where } \kappa_1 = \lambda e'\sigma_1.\ \kappa e'\sigma_1[l_e \mapsto e'] \\ \kappa e'\sigma & \text{otherwise, where } e' = \sigma(l_e) \end{cases}$$

$$\mathit{seq}(e_1, e_2) = \begin{cases} e_2 & \text{if } e_1 = (\texttt{const void}) \text{ else} \\ (\texttt{seq}\ (\texttt{seq}\ e_1\ e_3)\ e_4) & \text{if } e_2 = (\texttt{seq}\ e_3\ e_4) \\ (\texttt{seq}\ e_1\ e_2) & \text{otherwise} \end{cases}$$

$$\mathit{result}(e) = \begin{cases} e_2 \text{ if } e = (\texttt{seq}\ e_1\ e_2) \\ e \text{ otherwise} \end{cases}$$

Fig. 4. The inlining algorithm (continued).

is inlined, the residual code returned is a version of the integrated procedure's body specialized, via recursive application of the inliner, to its argument. If the call has not been inlined, the operand is processed for value using *visit*, and the call is residualized.

Processing on demand allows arguments bound to unreferenced formal parameters of an inlined procedure to be processed for effect rather than value. This often reduces the size and execution time of the residual program and may improve the accuracy of the information collected for variables in the residual program. For example, if `f` is `(lambda (x y z) (if (> z 0) (+ z y) x))`, the inlined call `(f e1 e2 3)` reduces to `(seq e1 ((lambda (y) (+ 3 y)) e2))`. Because `e1` is processed for effect, the residual code for the call may actually be smaller when inlined. If variable references in the source of `e1` no longer appear in the residual code, expressions bound to those variables may be processed for effect as well.

Primitive References. The current context determines how the algorithm treats primitive references. In Test or Effect contexts it residualizes `true` or `void` because a primitive reference cannot signal an error and always has a non-false value. In a Value context it simply residualizes the primitive reference. In a call context, the algorithm processes the operand for its value and attempts constant folding when the result is a constant. If successful, the call is marked inlined and the new constant is residualized; otherwise the primitive reference is residualized.

Lambda Expressions. In Test and Effect contexts `lambda` expressions reduce to the constants `true` and `void`, respectively. In a Value context the body is processed in an environment extended with a fresh variable for the formal parameter, and the `lambda` expression is reconstructed with the fresh variable and the optimized body.

In a call context $\mathsf{App}(op, \gamma, l_\gamma)$ the body is processed using the outer context of the call, γ, in an environment mapping the formal parameter x to a fresh residual-program variable x' bound to the operand op from the call site. The s field of x' is initialized with the contents of $\sigma(l_x)$ for the corresponding source-program variable x, i.e., the source flags of x' inherit the residual flags of x.

If no reference to x' is residualized while processing the body, then $\mathsf{ref} \notin \sigma_2(l_{x'})$ and the operand can be processed for effect. If x' is also unassigned, then a sequence is constructed from the residualized operand and the specialized body to ensure applicative-order sequencing of effects. Otherwise, a `let` binding (direct `lambda` call) is constructed for variables referenced or assigned in the residual code of the body. These transformations effectively inline a version of the procedure specialized to the call site for which the call context was originally constructed.

Letrec Expressions. (not shown) Handling a `letrec` expression is similar to handling a `lambda` expression in a call context: the body is processed in an extended (circular) environment and useless bindings are pruned. A binding can be pruned only after all expressions in its scope have been processed. Therefore, after processing the body of a `letrec` expression, the algorithm must pro-

```
(letrec ((f (lambda (x) (if (zero? x) 1 (* x (f (- x 1))))))
         (g (lambda () (display "hello")))
         (h (assign z (lambda () g)))
         (a (lambda (x) (b x)))
         (b (lambda (x) (a x))))
  (f 5))
```

Fig. 5. Sample `letrec` expression.

cess operands of as-yet unreferenced variables, such as `g`, `h`, `a`, and `b` in Fig. 5. Operands of variables not referenced in the source program are visited for effect. Processing of trivially effect-free operands, such as the `lambda` expression bound to `g`, or operands of already referenced variables such as `f`, is left for the pruning phase. Other operands are conservatively processed for value. The binding for an unassigned and unreferenced variable can then be eliminated provided its operand is effect-free, that is, it does not diverge, perform side effects, or capture a continuation. The binding for `g` is retained because a reference to `g` is residualized when the operand of `h` is conservatively processed for value and the binding for `h` is retained because its operand performs a side effect. The bindings for `a` and `b` are discarded. By visiting operands on demand, the algorithm generally obtains more complete information than would be obtained with any arbitrary processing order.

Variable References. Variable references are the focal points for copy propagation and inlining. In Effect context, a variable reference reduces to the constant `void`. In Value, Test and call contexts, the variable is renamed, and the operand bound to that variable, if any, is processed for value as a prelude to copy propagation and inlining. The initial processing of the operand conveniently biases the algorithm in favor of inlining at the leaves of the call graph.

Copy propagation and inlining are performed by *copy* (Fig. 4), which takes the Var structure for the renamed variable, the *result* e of the processed operand, and the current context, continuation, and store. Copies are propagated in cases one, four, and five of the definition. Inlining is attempted in the third case when e is a `lambda` expression processed in a call context.

3 Restraining the Algorithm

We now describe the mechanisms that bound the run time of the algorithm, limit code growth due to inlining, and detect cycles.

A linear bound is enforced on the run time of the algorithm by maintaining an effort counter whenever an expression is processed more than once, i.e., for each attempt to inline an operand. The effort counter advances each time I is called. If the effort bound is exceeded, the inlining attempt is aborted and the call is residualized instead. Because the inlining attempt may be aborted, *fold* does not set the inlined flag until processing is complete. The effort counter is not reset for nested integrations, as this could lead to nonlinear processing time. Because the number of call sites in the original source program is fixed and because we bound the amount of effort expended processing each of these sites, the run time of the algorithm is linear in the size of the input program.

The effort counter is an intuitive, no-nonsense mechanism that permits general optimization and allows the linear-time constant of proportionality to be scaled. Other methods for bounding the run time of inlining algorithms may abandon entire classes of useful optimizations. For example, if only calls to leaf routines are inlined, then programs containing η-expansions are not fully optimized.

Code growth is limited via a size counter that tracks the size of the residual code as it is produced. This counter is incremented whenever a form is residualized in the output. If the size threshold is exceeded while attempting to integrate a procedure, the inlining attempt is abandoned and the call is residualized instead. When inlining an operand bound to a variable referenced only once in the source program, no limit is imposed on the size of the residual code. Inlining procedures called only once does not directly increase the size of the residual program and, through specialization, may decrease code size. While the effort counter alone prevents indefinite inline expansion, the size counter prevents excessive growth in any one part of the program.

The effort and size counters are passed as additional arguments to I. When constructing a call context, the current effort counter is stored in each operand structure so that the cost of processing the operand may be charged to the original call site. In particular, this allows source-program operands to be processed without an effort counter. The size counter is not stored in the operand structure. Rather, a new counter with no limit is used when the operand is first processed for value. The resulting size count is cached along with the residual code in the operand structure. The cached size is added to the current size counter if a binding to this operand is residualized.

Cycles are detected by consulting additional flags within operand structures to recognize when the value of an operand is requested during the initial processing of that operand. An "outer-pending" flag, tested and set by *copy*, is used to detect recursive references encountered while inlining a procedural operand, as in the expression `((lambda (x) (x x)) (lambda (x) (x x)))`. An "inner-pending" flag, tested and set by *visit*, is used to detect recursive references within an operand as in `(letrec ((f (lambda () (f)))) (f))`. These flags can be extended to govern loop unrolling within a recursive procedure and unfolding at the call site. When a cycle is discovered while processing the operand for a variable, a variable reference is residualized.

The cycle-detection mechanism generally interrupts recursive inlining long before the effort counter is exceeded. This is important because it prevents the algorithm from wasting effort that could otherwise be applied to inlining. To illustrate, suppose that while inlining a call to `f`, the algorithm encounters a call to `g`, where `g` is bound to a recursive procedure. If cycle-detection were disabled, then recursively visiting the operand of `g` would eventually exceed the effort limit and the entire inlining attempt for `f` would be aborted. By detecting the cycle involving `g` early, the algorithm residualizes the call to `g` and resumes its effort to inline `f`.

Polyvariance is achieved by attempting to integrate a procedure at each of its call sites while relying on the size counter to abort any attempts for which the residual code produced would exceed the size limit. Thus, the algorithm bases the inlining decision at each call site on the size of the procedure when specialized to its arguments. In contrast, a monovariant approach bases the decision solely on the size of the procedure without taking into account the arguments. Polyvariance thus permits more aggressive inlining without an increase in the size threshold, resulting in better performance with tighter control over code growth.

4 Inlining Recursive Procedures

Using the cycle detection mechanisms described in Sect. 3, the algorithm recognizes that a variable is bound to a recursive operand when that operand is initially processed for value. When attempting to inline a call to a recursive procedure bound to `f`, the environment is extended to map `f` to a fresh variable, `f′`, with no operand, and the procedure body is processed in a call context. Because `f′` has no operand, recursive calls within the body are forced to residualize (see second case of I(`ref` x), Fig. 4). If, after processing the body, `f′` is unreferenced, then no recursive calls were residualized, i.e., the call folded completely and the inlining attempt succeeds.

If, after processing the body, `f′` is referenced but the fresh formal parameters are not referenced, then copy propagation or inlining successfully replaced references to the formals within the body. This often means the operands of recursive calls to `f` are known. For example, if `f` is the factorial function defined below, processing the call `(f 5)` in this manner yields `(* 5 (f′ 4))`.

```
(define f
  (lambda (x)
    (if (zero? x)
        1
        (* x (f (- x 1))))))
```

If some of the operands have constant residual expressions, a size counter is set along with modest effort and unfolding counters and we attempt to completely unfold the call. If these limits are not exceeded and no recursive calls remain in the resulting residual code, the inlining attempt succeeds. For example, the call `(f 5)` reduces to `120`.

When the preceding conditions do not hold, a call to a recursive procedure may still be inlined if the procedure body can be specialized to the arguments at the call site. For example, consider the definition of `fold` below.

Table 1. Performance increase due to inlining with different size and effort limits. Numbers less than 1 indicate a decrease in performance.

	Effort limit 100						Effort limit 1000					
Size limit:	10	20	30	40	50	60	10	20	30	40	50	60
boyer	1.03	1.04	1.04	1.04	1.04	1.04	1.03	1.04	1.06	1.06	1.07	1.06
dynamic	1.05	1.07	1.07	1.08	1.06	1.05	1.08	1.10	1.05	1.05	1.10	1.07
graphs	1.79	1.78	1.78	1.78	1.78	1.77	1.76	2.25	2.24	2.25	2.25	1.59
lattice	4.30	4.37	4.39	4.40	4.39	4.39	4.30	4.57	4.48	4.57	4.56	4.57
matrix	1.29	1.27	1.27	1.29	1.30	1.29	1.31	1.28	1.30	1.31	1.32	1.30
maze	1.45	1.38	1.40	1.35	1.40	1.42	1.46	1.39	1.40	1.42	1.41	1.39
nbody	.95	.97	.98	1.04	1.04	1.05	.92	1.00	.98	.93	1.04	1.03
splay	1.26	1.29	1.30	1.29	1.28	1.28	1.32	1.20	1.33	1.31	1.38	1.35
conform	2.41	2.50	2.62	2.70	2.69	2.66	2.44	2.52	2.47	2.68	2.67	2.69
earley	1.10	1.10	1.08	1.09	1.09	1.10	1.09	1.11	1.12	1.12	1.12	1.12
em-fun	1.15	1.18	1.05	1.02	1.09	1.02	1.20	1.15	1.07	1.15	1.15	1.14
em-imp	1.16	1.10	1.06	1.11	1.15	1.06	1.16	1.09	1.06	1.13	1.05	1.06
interpret	1.07	1.03	1.09	1.09	1.10	1.10	1.08	.99	1.10	1.07	1.07	1.07
peval	1.22	1.07	1.05	1.07	1.07	1.06	1.07	1.21	1.23	1.22	1.23	1.24
simplex	1.32	1.32	1.30	1.32	1.32	1.32	1.31	1.32	1.31	1.32	1.30	1.30

```
(define fold
  (lambda (f x b null? car cdr)
    (if (null? x)
        b
        (f (car x) (fold f (cdr x) b null? car cdr)))))
```

When the operand bound to `fold` is initially processed, the algorithm records the residual code for the operands at each recursive call. Specialization is attempted if some formal parameters are trivially invariant (in the above example, all parameters except `x`) and the residual expressions for their operands at the original call site are not variable references. This prevents trivial specialization. Specialization is accomplished by processing the procedure body in a call context that supplies operands only for the trivially invariant formal parameters. Additional details are provided in [9]. Specializing the call `(fold * x 1 zero? (lambda (x) x) (lambda (x) (- x 1)))` yields code identical to the earlier definition of `f`, modulo renaming.

5 Performance

We have implemented the inlining algorithm as an additional pass within *Chez Scheme*, a commercial optimizing compiler for Scheme. Although described in Sect. 2 as an interpreter written in continuation-passing style (CPS), the inlining algorithm is implemented in direct style, relying on the applicative-order semantics of Scheme to sequence effects. We do not CPS-convert input programs; doing so would not materially simplify the algorithm and could exaggerate the benefits obtained by inlining [1].

Table 2. Ratio of code size to original code size with different size and effort limits. Numbers less than 1 indicate a decrease in code size.

	Effort limit 100						Effort limit 1000					
Size limit:	10	20	30	40	50	60	10	20	30	40	50	60
boyer	.97	.97	.99	1.00	1.00	1.00	.97	.99	.99	1.01	1.01	1.01
dynamic	.87	1.18	1.20	1.20	1.31	1.31	.86	1.26	1.30	1.37	1.55	1.57
graphs	.72	.71	.71	.71	.71	.71	.80	.85	.85	.85	.85	.90
lattice	1.13	.91	.91	.91	.91	.91	1.13	.89	.89	.89	.89	.89
matrix	.96	1.01	1.03	1.03	1.03	1.03	.94	.99	1.01	1.00	1.00	1.00
maze	.74	.74	.78	.78	.78	.78	.74	.74	.78	.78	.78	.78
nbody	1.20	1.22	1.25	1.25	1.25	1.25	1.24	1.27	1.33	1.34	1.40	1.46
splay	1.00	1.00	1.02	1.06	1.06	1.06	.99	1.01	1.12	1.12	1.10	1.12
conform	.73	.80	.82	.86	.86	.86	.72	.75	.76	.80	.80	.80
earley	.91	.92	.96	.96	.96	.96	.82	.85	.89	.89	.91	.94
em-fun	.88	.92	1.07	1.09	1.09	1.09	.88	.91	1.15	1.17	1.13	1.12
em-imp	.87	.89	.89	.92	.92	.92	.85	.89	.88	.90	.90	.90
interpret	1.19	1.26	1.31	1.31	1.31	1.31	1.17	1.23	1.28	1.29	1.29	1.35
peval	1.01	1.04	1.04	1.04	1.04	1.04	1.00	1.03	1.03	1.03	1.03	1.03
simplex	.86	.86	.86	.86	.86	.86	.86	.86	.86	.86	.90	.90

The inlining algorithm was evaluated using a suite of common Scheme benchmark programs [8,4,9]. The benchmark programs provided by Jagannathan and Wright [8] had already been processed by their local simplification pass, which performs many of the same optimizations as our inliner. To facilitate comparison with their results, performance data was collected on a 150 MHz SGI MIPS R4400 workstation.

Table 1 shows the performance increase obtained when the benchmarks are optimized using different size and effort limits. With the exception of nbody, all programs show consistent and significant speedups. Profiling reveals that nbody spends most of its time indexing three levels deep into a structure of nested arrays. Thus, the variability we observe is likely due to cache effects. When run on the Intel Pentium Pro, nbody shows consistent speedups ranging from 1.06 to 1.11 (6–11%). As is to be expected, increasing the effort limit generally improves performance somewhat. Both graphs and peval improve substantially at higher effort limits.

Inlining more than doubles the performance of lattice, graphs, and conform. A large percentage of runtime calls made by lattice (73%) and graphs (43%) are to a user-defined `memv` procedure that searches a list for a given key. With a size limit of 4 our algorithm completely unfolds calls to `memv` in both programs because the target list happens to be a short constant at each call site for `memv`. The speedup of 4.57 for lattice at size limit 20 represents an additional speedup of 1.15 beyond the speedup of 3.97 due to unfolding calls to `memv` at size limit 4. The speedup of 2.25 for graphs at size limit 20 represents an additional speedup of 1.43 beyond the speedup of 1.57 due to unfolding calls to `memv` at size limit 4. The algorithm does not unfold any of the recursive calls within conform. The large speedup is

Table 3. Run time of the algorithm (in seconds) with different size and effort limits.

	Effort limit 100						Effort limit 1000					
Size limit:	10	20	30	40	50	60	10	20	30	40	50	60
boyer	.03	.04	.03	.04	.04	.04	.02	.08	.07	.08	.09	.09
dynamic	.33	.48	.52	.50	.54	.53	.34	.56	.57	.68	.81	1.06
graphs	.08	.08	.09	.09	.08	.07	.11	.13	.14	.13	.15	.14
lattice	.08	.08	.09	.08	.07	.08	.09	.11	.12	.10	.11	.12
matrix	.11	.11	.11	.11	.13	.11	.14	.18	.20	.21	.21	.21
maze	.11	.11	.13	.12	.13	.12	.11	.12	.13	.13	.14	.13
nbody	.20	.20	.20	.22	.20	.22	.29	.47	.48	.51	.57	.57
splay	.10	.10	.10	.09	.10	.11	.13	.15	.13	.13	.14	.16
conform	.08	.12	.13	.14	.14	.15	.24	.27	.28	.32	.34	.31
earley	.13	.15	.15	.15	.15	.17	.26	.31	.40	.41	.44	.45
em-fun	.17	.19	.20	.21	.21	.22	.40	.45	.66	.70	.68	.64
em-imp	.16	.18	.18	.18	.19	.18	.34	.45	.46	.50	.54	.51
interpret	.28	.29	.33	.34	.34	.34	.29	.31	.47	.51	.56	.55
peval	.08	.08	.09	.12	.13	.13	.17	.19	.19	.19	.22	.19
simplex	.05	.06	.06	.06	.05	.05	.09	.10	.10	.14	.14	.13

due simply to the fact that conform contains many small procedures that can be inlined.

Table 2 shows the effect of different size and effort limits on the size of the resulting object code. The table reports the ratio of code size after optimization to the original code size. For many programs, inlining actually decreases code size. For most programs, code size increases gradually as the size limit increases. This confirms the intuition that the size counter is an effective mechanism for regulating code growth.

Table 3 shows the run time of the algorithm for different size and effort limits. The algorithm completes both analysis and transformation of each program (except dynamic) in less than one second. Increasing the effort limit by an order of magnitude typically only doubles the run time of the algorithm. When the size limit is varied, the run time of the algorithm increases roughly in proportion to the increase in code size.

6 Related Work

Jagannathan and Wright [8] use a polyvariant flow analysis to identify inlining opportunities and to estimate the size of a procedure when specialized for a particular call site. Their approach is offline: all inlining decisions are made prior to any transformation. Our algorithm combines a less accurate analysis (effectively sub-0CFA [3]) with online transformation and polyvariant specialization. The time required by their flow analysis varies widely and can be excessive—110 seconds to analyze dynamic—rendering their method impractical for use in a production compiler.

Table 4. Performance increase, processing time and range of code size ratios for our algorithm (POLI) and the polyvariant flow-directed inlining of Jagannathan and Wright (PCFA). The columns labeled "non-block" indicate results relative to a baseline that is not block compiled. All programs were compiled and run using *Chez Scheme*. All data was collected on the 150 MHz SGI MIPS R4400.

	Performance Increase			Processing Time (seconds)		Code Size Ratios		
Baseline	block	non-block				block	non-block	
Algorithm	POLI	POLI	PCFA	POLI	PCFA	POLI	POLI	PCFA
boyer	1.07	1.34	1.72	.09	1.4	.97 – 1.01	.82 – .86	.95 – 1.10
dynamic	1.10	1.20	1.20	.81	110.3	.86 – 1.57	.78 – 1.43	1.52 – 2.48
graphs	2.25	2.45	1.30	.15	.3	.71 – .90	.65 – .82	.79 – .87
lattice	4.56	6.38	1.27	.11	.2	.89 – 1.13	.72 – .92	1.02 – 3.09
matrix	1.32	1.35	1.11	.21	.4	.94 – 1.03	.86 – .94	.93 – 1.21
maze	1.41	1.82	1.67	.14	.5	.74 – .78	.48 – .51	.89 – .98
nbody	1.04	1.38	1.11	.57	2.6	1.20 – 1.46	1.08 – 1.31	1.25 – 2.62
splay	1.38	1.35	1.20	.14	4.0	.99 – 1.12	.94 – 1.06	.94 – .95

Table 4 provides a rough comparison of the best speedups obtained by the flow-directed inlining algorithm of Jagannathan and Wright with the speedups obtained by our algorithm (labeled POLI) given a size limit of 50 and an effort limit of 1000. The table also compares processing time and the ranges of object code size for the two methods. The comparison is complicated by the fact that their optimizer effectively block compiles the benchmarks, yet their baseline for comparison is not block-compiled. Enabling block compilation reduces baseline object-code size and improves baseline performance considerably. To compensate for the difference in benchmarking methodology we present both our absolute results and the results we obtain relative to a baseline that is not block compiled. It appears that we achieve equivalent or better speedups for all benchmarks except boyer and better code size ratios for all benchmarks except splay, yet our algorithm completes both analysis and transformation in far less time than their analysis alone.

Ashley [4] evaluated the effectiveness of four different flow analyses for inlining. The analyses range from a fast analysis less accurate than 0CFA to a polyvariant analysis similar to 1CFA. His inlining algorithm is based on that of Jagannathan and Wright but exposes more opportunities for inlining.

Table 5 compares the performance increase, processing time, and code size ratios for Ashley's inlining algorithm justified by sub-0CFA and 1CFA analyses with the results obtained by our algorithm with effort limits of 100 and 1000. All data was generated using the same compiler and benchmarking methodology. In particular, the baseline and optimized programs were block-compiled in all cases. The data was collected on practically identical hardware (our machine has half the physical memory and second-level cache). All results in the table reflect a size threshold of 30. Our algorithm achieves better or virtually equivalent

Table 5. Comparison of performance increase, processing time, and ratio of code size to original code size for two flow-directed algorithms and our algorithm run with effort limits of 100 and 1000. All data was gathered on the 200 MHz Intel Pentium Pro. All programs were compiled and run using *Chez Scheme*. Processing time for <0cfa and 1cfa cases includes analysis time only, i.e., it does not reflect the cost of inlining or subsequent simplification. A size threshold of 30 was used in all cases.

Performance Increase

	<0cfa	1cfa	100	1000
boyer	1.01	1.03	1.01	1.02
dynamic	1.11	1.05	1.10	1.10
graphs	1.20	1.27	1.70	2.18
lattice	1.03	1.33	3.92	4.06
matrix	1.05	1.09	1.22	1.21
maze	1.27	1.27	1.28	1.29
nbody	1.06	1.10	1.09	1.06
splay	1.09	1.11	1.23	1.26
conform	1.72	1.75	1.94	1.95
earley	1.10	1.11	1.07	1.10
interpret	1.12	—	1.14	1.15
peval	1.15	1.15	1.14	1.15
simplex	1.39	1.39	1.43	1.43

Processing Time (seconds)

	<0cfa	1cfa	100	1000
boyer	.01	.07	.02	.03
dynamic	.32	2.09	.20	.23
graphs	.05	.20	.03	.05
lattice	.01	.19	.03	.04
matrix	.05	3.02	.06	.09
maze	.05	.19	.03	.04
nbody	.13	1.98	.08	.19
splay	.06	.95	.02	.05
conform	.04	.31	.05	.12
earley	.07	.45	.06	.15
interpret	.39	—	.13	.19
peval	.07	.41	.04	.07
simplex	.03	.19	.03	.04

Code Size Ratio

	<0cfa	1cfa	100	1000
boyer	1.47	2.33	.98	1.01
dynamic	3.56	4.72	1.46	1.61
graphs	.73	.82	.68	.72
lattice	1.20	1.51	.94	.92
matrix	1.11	1.23	1.04	1.02
maze	.88	.89	.75	.75
nbody	1.29	1.37	1.19	1.25
splay	1.42	1.46	1.07	1.18
conform	1.06	1.19	.86	.81
earley	1.06	1.13	1.00	.94
interpret	2.06	—	1.47	1.45
peval	1.20	1.31	1.04	1.03
simplex	.89	.92	.81	.81

speedups for all of the benchmarks and typically completes both analysis and transformation in less time than is required for flow analysis alone. The volatility of the 1CFA analysis is highlighted by the fact that it exhausted virtual memory (128Mb core, 100Mb swap) while attempting to analyze interpret. Our code size ratios are consistently better than those obtained using either flow analysis. This supports our claim that online transformation provides more accurate size information on which to base inlining decisions.

The intuitive appeal of an offline flow-directed approach is that the results of flow analysis might be used to justify other optimizations in addition to inlining. In fact, Ashley found that inlining violates the soundness of flow information about procedures which could otherwise be used by procedure call optimizations in later passes of the compiler. This means that flow analysis must be repeated after inlining.

In the SML/NJ compiler, Appel uses an offline algorithm in which a data gathering pass is run alternately with a transformation pass that performs several optimizations including constant folding, β-contraction, uncurrying, and inlining of functions called only once [1]. This transformation phase is iterated until the number of contractions found is below some threshold. Next, inline expansion is performed using several heuristics to estimate the cost and benefit of inlining. To improve these estimates, this "round" of optimization is iterated beginning with the contraction phase. Although our algorithm can be iterated, we have found that few programs actually benefit from additional passes.

Appel and Jim recently described a linear-time algorithm that performs constant folding, dead-variable elimination, and inlining of functions called only once [2]. They have implemented an $O(n^2)$ variant of the algorithm to replace the contraction phase of the SML/NJ compiler described above. The new algorithm is online in that usage counts for variables are updated as optimizations are performed, and it is demand-driven in that bodies of binding forms are processed before processing the bindings. The new algorithm typically performs as many contractions in one pass as their earlier offline algorithm performs in three. They do not consider the more difficult problem of inlining functions called more than once.

Dean and Chambers [7] describe an online approach in which the cost and benefit of inlining at a call site are estimated by examining the code produced when the routine is tentatively inlined and optimized for the call site. To amortize the high cost of these inlining trials, they cache the results in a persistent database indexed by a description of the static information at the call site that justified optimizations that made inlining acceptable. By using inlining trials instead of offline scoring estimates, they were able to reduce overall compile time and code size with minimal loss of run-time performance. Our algorithm can be viewed as incorporating a lightweight form of inlining trials.

Wegman and Zadeck [10] present a fast global constant propagation algorithm and show how it can be extended to perform many of the specializations needed when procedures are inlined. Their algorithm propagates constants through the static single-assignment (SSA) graph [6] using a demand-driven traversal of the control-flow graph that facilitates elimination of unreachable code which, in turn, improves the accuracy of the information collected. The inlining algorithm they describe does not address key issues such as identifying inlining candidates, limiting code growth, and achieving termination in the presence of recursion. Their algorithm is not online, polyvariant, or context-sensitive in the sense we have described.

Ball describes an analysis that determines which parameters contribute to the value of the expressions in a procedure body [5]. When constant parameters are available at a call site he uses the parameter dependency information to guide inlining decisions with an estimate of code savings and performance gain that is based on predictions about the impact of subsequent optimizations. We suspect the speed of our algorithm could be improved by using the type-group analysis of Dean and Chambers, or the more direct parameter dependency analysis of Ball, to exploit polyvariance only where it is likely to be useful.

7 Conclusion

We have presented a procedure inlining algorithm that is both fast and effective. The algorithm improves performance dramatically for several benchmark programs without sacrificing performance on any. Moreover our algorithm is fast enough to process programs of several thousand lines in under one second on available hardware. The run time of our algorithm is linear in the size of the input program, which is important for scalable optimizations. Because it is both fast and effective, the algorithm is well-suited for use in a production compiler. The algorithm also provides a competent basis for comparison that can be used to isolate the additional benefit, if any, obtained through more involved analysis techniques. Finally, we have presented evidence that online techniques are faster than offline, analysis-intensive techniques, yet are equally effective for inlining.

Acknowledgements: Mike Ashley and Bob Burger contributed to the development of the inlining algorithm. Mike Ashley, Carl Bruggeman, Bob Burger, and Mark Leone provided helpful comments on drafts of this paper. We thank Suresh Jagannathan and Andrew Wright for providing us with details on their inlining algorithm and the benchmark code they used to test it, and Andy Hanson for allowing us to run benchmarks on his SGI workstation.

References

1. Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
2. Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. To appear in *Journal of Functional Programming.*
3. J. Michael Ashley. A practical and flexible flow analysis for higher-order languages. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 184–194, 1996.
4. J. Michael Ashley. The effectiveness of flow analysis for inlining. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, 1997. To appear.
5. J. Eugene Ball. Predicting the effects of optimization on a procedure body. *SIGPLAN Notices*, 14(8):214–220, August 1979. *Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction.*
6. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K Zadeck. Efficiently computing static single assignment form and the control dependence graph. Technical Report RC 14756, IBM, 1991.
7. Jeffrey Dean and Craig Chambers. Towards better inlining decisions using inlining trials. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 273–282, 1994.
8. Suresh Jagannathan and Andrew Wright. Flow-directed inlining. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 193–205, 1996.
9. Oscar Waddell and R. Kent Dybvig. Fast and effective procedure inlining. Technical Report 484, Indiana University Computer Science Department, Lindley Hall 101, Bloomington Indiana, 47405, 1997.
10. M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 3(2):181–210, 1991.