Implementation of Speculative Parallelism in Functional Languages

P. V. R. Murthy and V. Rajaraman

Abstract— **A compile-time analysis technique is developed to derive the probability with which a user-defined function or a supercombinator requires each one of its arguments. This provides a basis for identifying useful speculative parallelism in a program. The performance of speculative evaluation is compared with that of lazy evaluation, and the necessary conditions under which speculative evaluation performs better are identified.**

Index Terms— **Conservative parallelism, speculative parallelism, lazy evaluation, branch speculation, argument speculation, strictness analysis**

I. Introduction

CONCURRENCY in lazy implementations of functional languages is increased by scheduling tasks for the evaluation of the strict arguments along with the function before they are actually demanded [2]. A speculative evaluator is one that initiates the computation of some expressions before a lazy evaluator would have, i.e., before they are known to be necessary or unnecessary. If they later turn out to be necessary, performance can improve because they were initiated earlier, and so can complete earlier (subject to the availability of spare processors). If they turn out to be unnecessary, performance can decline because the resources they used were wasted. This introduces a trade-off. Burton [1] and Osborne [7] have addressed the problem of speculative computation, mainly in problems involving parallel search, multiple-approach speculative computation and order-based speculative computation. Osborne shows how the above speculative computation approaches may be incorporated into Multilisp and points out that speculative computation does yield benefits.

In this paper, we are mainly concerned with speculative evaluation that arises during the reduction of an application of a function (supercombinator). A promising expression is one whose value is required for a program's evaluation with a high probability. During the reduction of a function application, promising nonstrict arguments may be reduced in parallel with the strict arguments and the function body. This evaluation model is referred to as argument speculation. Assume that (equal $L1$ $L2$) and Palindrome (L) evaluate to true with low probabilities (Fig. 1). During the reduction of an application (f $E1$ $E2$ $E3$ $E4$) (Fig. 1(a)), if the reducer has the knowledge that $L4$ is required with a higher probability than $L3$, then the average reduction time can be reduced by reducing the nonstrict argument $E4$ in parallel with the strict arguments $E1$ and $E2$.

Another speculative evaluation model we discuss in this paper is referred to as branch speculation. In branch speculation, no speculation is done during the reduction of a function application, but during the reduction of a conditional expression, the more promising branch in a conditional expression is reduced in parallel with the predicate. During the reduction of the conditional expression (Fig. 1(b)), if the reducer has the knowledge that Ef is required with a higher probability, the average reduction time can be reduced by reducing Ef in parallel with Palindrome (L).

This paper is organized as follows. Section II suggests that branch probabilities be used directly to identify promising branches in the case of branch speculation, and that, indirectly, a compiler may use branch probabilities to identify promising nonstrict arguments for use in argument speculation. Section III provides a compile-time analysis technique to compute the probability with which each function in a program requires each one of its nonstrict parameters. Section IV shows how argument probabilities derived at compile-time may be used at run-time in speculatively evaluating promising nonstrict arguments of a function. It also discusses about how branch probabilities may be used in branch speculation. Section V discusses the problems in speculative evaluation and proposes some solutions to them. Section VI discusses the termination of programs under speculative evaluation. Section VII provides a further discussion on speculative evaluation models.

II. Branch Probabilities

To know the more promising branch in a conditional expression, we need to know the probability with which the predicate in it evaluates to true, which is referred to as the branch probability. In another context, branch probabilities have been used to obtain closed-form expressions for the execution times of Lisp programs [10]. To identify the more useful nonstrict arguments, it is necessary to identify those nonstrict arguments occurring in paths that have a greater likelihood of being selected during a function's evaluation. Branch probabilities may be obtained through measurement or supplied by the user based on experience.

In functional programs, most functions are defined recursively. Since the recursive branches are selected most often, it is reasonable for the user to specify a large probability of selection in favor of the recursive branch. Branch probability may be specified by the user as part of the conditional expression as follows. In "if Ec then $*q*$ Et else Ef" q

Manuscript received October 1, 1992; revised October 1, 1993.

The authors are with the Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560 012 India; e-mail: murthy@serc.iisc.ernet.in.

IEEE Log Number 9403101.

```
f(L1,L2,L3,L4)==if (equal L1 L2) then
                             length L3
                        else  length L4;
```
(a)

```
if Palindrome(L) then
     Et
else Ef;
```
(b)

Fig. 1. Examples of speculative evaluation.

is the probability with which the predicate Ec evaluates to true. The branch probability q follows the key word "then" and is enclosed within $*$'s. Consider "$f(L1, L2, L3) ==$ if (Palindrome $L1$) then $*0.1*$ (h $L2$) else (h $L3$)" where function h is strict in its argument. It is easy to see that f requires $L2$ with probability 0.1 and $L3$ with probability 0.9 for its evaluation. We would like a compiler to derive these argument probabilities, which is the topic addressed by the next section.

Care must be taken, as far as possible, to see that branch probabilities specified in a program do not contradict each other. A compiler cannot statically detect contradicting branch probabilities in all cases [5]. Also, in the case of nested conditionals, the truth value of an inner predicate may depend upon the truth value of an outer predicate, and thus it is necessary that conditional branch probabilities are specified.

III. Computing Argument Probabilities at Compile-Time

We first develop rules that a compiler uses to compute the probability with which a primitive expression requires a variable for its evaluation. The compiler uses these rules to compute the probability with which a function requires its nonstrict parameters for its evaluation. Recursive functions pose a problem, which we point out in Section III-B. Also, we provide a solution based on successive approximation. In this paper, we confine ourselves to developing the method for a first-order functional language with nonflat domains. The extensions necessary to deal with higher-order functions can be found in [5].

A. Probability with Which a Primitive Expression Requires a Given Variable

Let E be an expression, and let x be a variable. Let E_x and E_x^c denote the events that "E requires x for its evaluation," and "E does not require x for its evaluation," respectively. Let P be the probability function that maps E_x into a real number in the interval [0.0, 1.0]. Note that all the variables defined through lets and letrecs are graphically substituted. We now specify P for each form of expression.

Rule 1—Probability that a Constant Requires a Variable x: Let E be a constant. $P(E_x) = 0$. E.g., $P(5_x) = 0$, $P((5 : 2 : [\,])_x) = 0$.

Rule 2—Probability that a Variable Requires a Variable x: Let E be a variable. If E is the same as x, $P(E_x) = 1$, and is 0 otherwise. E.g., $P(y_x) = 0$, $P(x_x) = 1$.

Rule 3—Probability that $(e1+e2)$ Requires a Variable x: Let E be $(e1 + e2)$. Assuming that $e1_x^c$ and $e2_x$ are independent events, we have $P(E_x) = P(e1_x) + (1 - P(e1_x))P(e2_x)$. Note that the same rule holds for all strict binary functions like $*, /$, div, mod, and so forth. E.g., $P(((x * x) + z)_x) = P((x * x)_x) + (1 - P((x * x)_x)) * P(z_x) = 1$.

Rule 4—Probability that a Conditional Requires a Variable x: Let E be the expression "if Ec then $*q*$ Et else Ef" where Ec, Et and Ef are some expressions. Let q be the probability with which Ec evaluates to true. Let Ec^t, Ec^f be the events that Ec evaluates to true and that Ec evaluates to false, respectively.

Assuming (see Section III-E) the independence of the following:

1) the events Ec_x^c and $((Ec^t \cap Et_x) \cup (Ec^f \cap Ef_x))$, and
2) the events Ec^t, Et_x and that of the events Ec^f, Ef_x,

we have $P(E_x) = P(Ec_x) + (1 - P(Ec_x))(qP(Et_x) + (1 - q)P(Ef_x))$. E.g., $P($(if z then $*0.1*$ y else $x)_x) = P(z_x) + (1 - P(z_x)) * (0.1 * P(y_x) + 0.9 * P(x_x)) = 0.9$. The binary operations **AND-OR** are not strict. The user may annotate a binary operation containing **AND-OR** with the probability (q) with which the first operand evaluates to true. These expressions are converted into conditional expressions, and the above rule is applied.

Rule 5—Probability that $(E1 : E2)$ Requires a Variable x: Let E be $(E1 : E2)$, where ":" is the lazy list constructor that constructs a Cons cell with $E1$ as the head and $E2$ as the tail, but does not evaluate them. $P(E_x) = 0$. E.g., $P((x : [\,])_x) = 0$.

Rule 6—Probability that $(\langle unop \rangle e)$ Requires a Variable x: If E is $(\langle \text{unop} \rangle e)$, where $\langle \text{unop} \rangle$ is any unary function such as negate, not, atom, but not a selector function such as hd or tl, and if e is some expression, then we have $P(E_x) = P(e_x)$. (We specify $P(E_x)$ if $\langle \text{unop} \rangle$ is a selector function in Rule 8.) E.g., $P((\text{atom}x)_x) = 1, P((\text{not } b)_c) = 0$.

Rule 7—Probability that a User-Defined Function Application Requires a Variable x: E is a user-defined function application (or a supercombinator application) f $e, \cdots e_n$, where f is defined as $f(x_1, \cdots x_n)$ $=e.$ $e_1, \cdots, e_n$, and e are some expressions. $P(E_x) = P(E'_x)$, where $E' = e[e_1/x_1, \cdots, e_n/x_n]$. E.g., $f(a, b) = a + b$, $P((f(x * x)(z * z))_x) = P(((x * x) + (z * z))_x) = 1$.

Rule 8—Probability that (Selector e) Requires a Variable x: Let E be (selector e), where selector is hd or tl, and e is some expression. Note that the evaluation of (hd e) consists of first reducing e to weak head normal form (WHNF) [8], and then reducing the head of the resulting Cons cell to WHNF, which is the result of E. Similarly, the evaluation of (tl e) consists of first reducing e to WHNF, and then reducing the tail of the resulting Cons cell to WHNF. Before we specify Rule 8, we define (e. selector). The expression (selector e) is transformed into the expression (e. selector), and the probability function P is applied on (e. selector).

Definition 1: Given an expression e (of the forms mentioned below), we define the expression (e. selector), where selector is either hd or tl, as follows.

Case 1: e is $e1 : e2$, where $e1$ and $e2$ are some expressions.

$$\begin{aligned} e.\ \text{selector} &= e1, \quad \text{if selector is } hd, \\ &= e2, \quad \text{if selector is } tl. \end{aligned}$$

Case 2: e is "if Ec then $*q*$ Et else Ef."

e. selector =

$$\text{if } Ec \text{ then } *\, q * (Et.\ \text{selector}) \text{ else } (Ef.\ \text{selector})$$

Case 3: e is $(fe_1 \cdots e_n)$, where $e_1 \cdots e_n$ are some expressions, and f is a supercombinator defined as $f(x_1, \cdots, x_n) =$ Expr. e. selector $= (Expr[e_1/x_1, \cdots, e_n/x_n])$. selector.

Case 4: e is (selector 'e'), where selector$'$ is either hd or tl and e' is some expression.

$$e.\ \text{selector} = (e'.\ \text{selector}').\ \text{selector}.$$

Definition 2: Now we define $P(E_x)$, where E is (selector e).

$$\begin{aligned} P((\text{ selector } e)_x) &= P((e.\ \text{selector})_x), \text{ if } e \text{ is not a parameter,} \\ &= 1, \text{if } e \text{ is a parameter and same as } x, \\ &= 0, \text{ if } e \text{ is a parameter and not the same as } x. \end{aligned}$$

Note that if e is a parameter, it has to be a list-type one, and to evaluate (selector e), whatever expression is bound to the parameter at run-time has to be evaluated. E.g., let $f(z) == 1 : z$; $P((hd\ (f\ x))_x) = P((hd\ (1 : x))_x) = 0$. $P((tl(f\ x))_x) = P((tl(1 : x))_x) = P(x_x) = 1$.

Rule 9—Probability with Which a Function Requires a Parameter: Let f be a supercombinator with $x1, \cdots, xn$ as formal parameters and defined as $f(x1, \cdots, xn) = e$, where e is some expression. Let (f_{xi}) denote the event that the supercombinator f requires xi for its evaluation. $P(f_{xi}) = P(e_{xi})$. E.g., $f(x, y) ==$ if x then $*0.6*$ y else 0; $P(f_x) = 1, P(f_y) = 0.6$.

To determine the probabilities with which a user-defined function (or a supercombinator) requires its arguments, the compiler, after generating the graphs for the functions in a program(see Section IV-A), uses *Rule 1* to *Rule 8*.

B. Probability with Which a Recursive Function Requires an Argument

$P(f_z)$ for the function f (Fig. 2) is computed below. $P(f_z) = (1-q)*P((f(x-1)\ (y+z)\ z)_z)$ (from Rules 4, 3, 2, and 1) $= (1-q)*(q+(1-q)*P((f(x-1-1)\ (y+z+z)\ z)_z))$ (*from Rule 7*) $= (1-q)*(q+(1-q)*(q+(1-q)*(\cdots)))$. It can be seen that $P(f_z) = (1-q)$. However, while computing $P(f_z)$, the compiler may get into an infinite loop. We now present a successive approximation method to overcome this problem. We first state the solution informally, and some terms used in this explanation are defined formally later in this section.

```
f(x,y,z) == if (x=0) then *q* y else f (x-1) (y+z) z ;
```

Fig. 2. Example for illustrating refinement.

Let $f^0 =$ if $(x = 0)$; then $*q*y$; else $f\ (x-1)\ (y+z)\ z$. By expanding the recursive function call in f^0, say, we obtain f^1. In general, obtain f^k from $f^{(k-1)}$, $k \geq 1$, by expanding the recursive function call present in $f^{(k-1)}$.

Let f_z^k be the event that f requires the value of z within the first k expansions of f. We prove later in this section that $f_z^{(k-1)}$ is a subset of f_z^k, i.e., $P(f_z^{(k-1)}) \leq P(f_z^k)$. The compiler computes $P(f_z^k)$, assuming that the probability with which a recursive function call in f^k requires z to be 0 for $k \geq 0$. (f^k may contain outermost nonrecursive function calls, and the compiler uses Rule 7 to compute the probabilities with which they require z.) We show later in this section that the sequence $P(f_z^0)$, $P(f_z^1), \cdots$ converges and that the limit is $P(f_z)$.

The process of obtaining f^k from $f^{(k-1)}, k \geq 0$, is called a refinement. In the case of most recursive expressions, a refinement simply consists of expanding the outermost recursive calls present in them once. The purpose of a refinement of a recursive expression is to bring the variables present in the arguments in recursive calls into the nonrecursive portion of the resulting expression, so that for probability computation purposes, the rules developed for primitive expressions may be employed. Thus, a refinement may not simply be a single expansion of each one of the outermost recursive function calls in a recursive expression. Repeated expansions of recursive function calls may take place until such a stage that the variables present in the arguments of the outermost recursive calls appear in the nonrecursive portion of the expression. The above process is a single refinement. With each refinement, the probability value being computed increases. If it does not increase, the computation has converged.

Definition 3 (Outermost Expression, Outermost Application): Let e be a subexpression of an expression E. The subexpression e is said to be outermost in E if there exists at least one occurrence of e in E such that e is not a subexpression of an argument in any supercombinator application in E. If e is a supercombinator application, then we refer to it as an outermost supercombinator application in E. Consider the following expressions.

1) Say, E is $(x*y) + (x+y)$; then $x, y, (x+y), (x*y)$ and $(x*y) + (x+y)$ are outermost in E.
2) E is $(f\quad e1\quad e2)\ +g\ (f\quad e1\quad e2)\ e3$

Here $(f\quad e1\quad e2)$ occurs twice within E, once not as an argument in any supercombinator application and once as an argument in the application of the function g. The first occurrence of the application $(f\quad e1\quad e2)$ is outermost in E. Also, the application $(g(f\quad e1\quad e2)\quad e3)$ is outermost in E.

Definition 4 (Outermost-Exprs(E, S)): Let E be an expression, and let S be a set containing some or all subexpressions of E. Outermost-exprs(E, S) = true, if each and every member of S that is not a constant expression is outermost in E; otherwise, it is false.

Definition 5 (Instance of a Supercombinator): Let A be $(g\ e_1 \cdots e_n)$, an application of some supercombinator g, defined as $g(x_1, \cdots, x_n) =$ expr, where $e_1, \cdots, e_n, n \geq 1$, and expr are some expressions. Instance (A) is defined to be expr$[e_1/x_1, \cdots, e_n/x_n]$.

Definition 6: Given an n-ary function $f\ x_1 \cdots x_n$, parameter x_i is said to be a relevant parameter [6] of $f\ x_1 \cdots x_n$ iff there exist values $a_1 \cdots a_n$ such that the following condition holds:

$$f a_1 \cdots a_{(i-1)} \perp a_{(i+1)} \cdots a_n \neq f\ a_1 \cdots a_i \cdots a_n.$$

For instance, consider the following:

$$g(x, y, z, p) == \text{ if } (x = 0) \text{ then } y \text{ else } g(x-1)\ (y+z)\ z\ p.$$

The parameter p is not relevant, and is referred to as an irrelevant parameter of g.

In the following, we assume that the parameters of all supercombinators are relevant. However, we point out the necessary modifications in the analysis in the presence of irrelevant parameters later in Section III-C.

Definition 7 (Surface-Exprs(E, S)): Let E be an expression, and let S be a set containing some subexpressions of E.

Surface-Exprs(E, S) =if Outermost-exprs(E, S) then E
else Surface-Exprs(E', S)

where $E' = E$[Instance $(fa_1)/fa_1, \cdots,$ Instance $(fa_n)/fa_n]$, where $\{fa_1, \cdots, fa_n\}, n \geq 1$, is the set of all the outermost supercombinator applications in E that are not constant expressions. (Here Instance$(fa_i)/fa_i$ means that Instance (fa_i) is substituted for the supercombinator application fa_i.)

Definition 8 (Arg_to_basic_exprs(Arg)): Let Arg be an expression, and let k be the nesting depth of calls to supercombinators in Arg. (For instance, k for the expression E, $(hd(p : z) + f(g\ (h\ x\ y\ z)) + f(g\ y))$, is 3. If, in an expression, there do not exist any applications of supercombinators at all, then $k = 0$.) If Ap is an application of a supercombinator, let vars(Ap) denote the set of all variables present in the arguments in Ap. Let Applications$_k$(Arg) be the set of all the applications in Arg at the kth nesting level. (In the above example, Applications$_3$ $(E) = \{(h\ x\ y\ z)\}$.)

Arg-to-basic-exprs(Arg) = Bring-vars-to-surface(Arg, k), where Bring-vars-to-surface(Arg,k) = if $k = 0$ then Arg else Bring-vars-to-surface(Arg$'$, $k-1$), where Arg$'$ =Arg[Surface-Exprs$(Ap_1$, vars$(Ap_1))/Ap_1$, $\cdots$, Surface-Exprs$(Ap_n$, vars$(Ap_n))/Ap_n]$, where Applications$_k$(Arg) $= \{Ap_1, \cdots, Ap_n\}, n \geq 1$.

Definition 9 (Rep-Ins-fn(RA)): Let RA be the application $(fe_1 \cdots e_n)$, where f is a supercombinator. $e_1, \cdots, e_n$ are some expressions.

Rep-Ins-fn (RA) =
Surface-Exprs(Instance($f\ e'_1 \cdots e'_n$), $\{e'_1, \cdots, e'_n\}$)
where e'_1 = Arg-to-basic-exprs(e_1),
$\vdots$
e'_n = Arg-to-basic-exprs(e_n).

Definition 10 Refine(E): Let E be an expression. Let $\{a_1, \cdots, a_n\}, n \geq 0$, be the set of all the outermost supercombinator applications in E that are not constant expressions.

refine$(E) = E$[Rep-Ins-fn$(a_1)/a_1, \cdots,$ Rep-Ins-fn$(a_n)/a_n]$.

(Here Rep-Ins-fn$(a_i)/a_i$ means that the supercombinator application a_i is substituted by Rep-Ins-fn(a_i).)

Definition 11: Let f be defined as $f\ (x_1, \cdots, x_n) = E$, with E being some expression. We define the expression $f^k, k \geq 0$, as follows:

$$\begin{aligned} f^k &= E, \text{ if } k = 0, \\ &= \text{refine}(f^{(k-1)}), \text{ if } k > 0. \end{aligned}$$

f^0 is said to be obtained by performing zero refinements on f. $f^k, k \geq 0$, is said to be obtained by performing k successive refinements on f. It may be noted that if E is not a recursive expression, applications of only nonrecursive functions may be present in E, which would all be expanded into primitive expressions to obtain f^1, and the set of function applications in f^1 would be empty. In this case, each $f^k, k >= 2$, would be identical to the expression f^1.

Lemma 1: Let f be a supercombinator with x as a parameter. Then we have $f_x^k \subseteq f_x^{(k+1)}, k \geq 0$.

Proof: f_x^k denotes the event that the evaluation of the expression obtained by performing k successive refinements on f requires x, without having to evaluate (or before evaluating) the recursive calls present in f^k.

Let the set of the outermost supercombinator applications present in f^k that are not constant expressions be A^f.

Let $A^f = \{a_1, \cdots, a_n\}, n \geq 0$. Then we have:

$$f^{(k+1)} = f^k[\text{Rep-Ins-fn}(a_1)/a_1, \cdots, \text{Rep-Ins-fn}(a_n)/a_n]$$

Case 1: Note that if A^f does not contain any recursive expressions, we have $f_x^k = f_x^{(k+1)}$.

Case 2: Let R^f be the set of recursive expressions in A^f that need to be refined to obtain $f^{(k+1)}$ from f^k; i.e., $R^f \subseteq A^f$. Let R_x^f denote the event that the evaluation of one or more of the nonrecursive expressions obtained by refining members of R^f to form $f^{(k+1)}$ requires x.

$f_x^{(k+1)}$ occurs if either f_x^k occurs or R_x^f occurs; i.e., $f_x^{(k+1)} = f_x^k\ U\ R_x^f$.

Hence, $f_x^k \subseteq f_x^{(k+1)}, k >= 0$. □

Theorem 1: Let f be a supercombinator with x as a parameter. Then we have:

$$\lim_{k \to \infty} P(f_x^k) = P(f_x).$$

Proof: The evaluation of f requires x if the evaluation of any one of the expressions $f^0, f^1, \cdots, f^k, \cdots, k \geq 0$, requires x, without having to evaluate (or before evaluating) the recursive calls present in each one of them.

$$\text{That is, } f_x = \bigcup_{k=1}^{\infty} f_x^k \qquad (1)$$

From Lemma 1, we know that the sequence $\{f_x^k\}$ is montonically increasing with k. Hence, we have the following:

$$\lim_{k\to\infty} f_x^k = \bigcup_{k=1}^{\infty} f_x^k \qquad (2)$$

Thus, $\{f_x^k\}$ is a convergent sequence of events.

Every probability measure P [11] is sequentially continuous in the sense that if $\{A_n\}$ is a convergent sequence of events, then $\lim_{n\to\infty} P(A_n) = P(\lim_{n\to\infty} A_n)$ Since $\{f_x^k\}$ is a convergent sequence of events, we have the following:

$$\lim_{k\to\infty} P(f_x^k) = P(\lim_{k\to\infty} f_x^k).$$

From (1) and (2), we have:

$$\lim_{k\to\infty} P(f_x^k) = P(f_x). \qquad \square$$

For the function f given in Fig. 2, the sequence $\{P(f_z^0), P(f_z^1), \cdots\}$ has a limit, and the limit is $P(f_z)$. Here the above sequence is a monotonically increasing sequence. $(P(f_z^{(k+2)}) - P(f_z^{(k+1)})) \leq (P(f_z^{(k+1)}) - P(f_z^k)), k \geq 0$. The sequence of successive differences in probability values follows a geometric progression and is a monotonically decreasing sequence. The user may specify that when two successive probability values differ by less than or equal to a chosen tolerance value (ϵ), the compiler may stop performing further refinements on f. For instance, if $\epsilon = 0.01$, then 14 refinements on f are needed to reach the desired accuracy. $P(f_z^{13}) = 0.7560195$, $P(f_z^{14}) = 0.7648156$. The actual value of $P(f_z)$ is 0.8 .

Consider

```
member(a, x) ==
    if (x = []) then * 0.1 * False
    else
      if (a = (hd x)) then *0.5* True
      else member a(tl x);
```

$P(\text{member}_a^0) = 0.9$. $P(\text{member}_a^1) = 0.9$. For computing $P(\text{member}_a)$, only one refinement on member needs to be performed, because $P(\text{member}_a^1) - P(\text{member}_a^0) = 0$.

C. Refinement in the Presence of Irrelevant Parameters

We now point out the additions, required in the presence of irrelevant parameters, to the refinement process already described. To detect the formal parameters of a supercombinator that are not relevant, an analysis called Relevance Information Analysis [6] may be performed on a given program. Definitions 7 and 8 need to be modified suitably in view of the fact that an irrelevant argument in a function application cannot be made outermost in the expression obtained by instantiation.

D. Optimizations in Deriving Argument Probabilities

Consider $f(..) == \cdots g(\cdots) \cdots$ and $g(..) == \cdots g(\cdots) \cdots$. In this case, the compiler may first compute the probability with which g requires its arguments and subsequently use this information while computing the probabilities with which f requires its arguments. This would avoid repeatedly scanning the body of g twice. In order to know the order in which functions may be traversed, a static call graph may be constructed and traversed backward in a breadth-first manner. However, the above optimization cannot always be used, for instance, when f and g are mutually recursive.

E. Comments on the Independence Assumption

The independence assumptions made in Rule 4 in Section III-A are not theoretically sound, but are made for pragmatic reasons. Errors due to the independence assumption can be minimized by specifying conditional branch probabilities in the case of nested conditional expressions. Despite the independence assumptions, the probability with which a function requires a nonstrict argument is never incorrectly reported to be 1, and is always reported to be less than 1 [5]. Also, the probability with which a function requires a strict argument is always reported to be 1 [5].

F. Relationship of Our Method to Strictness Analysis

This method is in fact a generalization of traditional strictness analysis, because it not only can determine strict arguments but also can quantify how strict a function is in each of its nonstrict arguments. However, just to perform strictness analysis, a conventional strictness analyzer [2] would be more efficient. Consider $f(x, y) ==$ if$(x = 0)$ then y else $f(x-1)\ y$. $P(f_x)$ would be computed to be 1 in just one refinement of f. However, if the branch probability with which $(x = 0)$ evaluates to true is quite low, then, in order for $P(f_y)$ to converge to 1, several refinements on f are needed, whereas a traditional strictness analyzer would detect that f is strict in y faster.

The problem is that in order for an argument to be declared strict by our method, a probability value of 1 should be reached. A value of 0.0 for ϵ has to be specified to achieve this. The number of refinements performed depends upon the desired accuracy. But for nonstrict arguments, we are not interested in their actual probability values. We are interested only in their relative promise. To determine the relative utility of nonstrict arguments, a value 0.0 for ϵ need not be specified. For these reasons, we suggest that strict arguments of functions be determined by a traditional strictness analyzer first, and then the method proposed in this section may be employed to determine the relative utility of nonstrict arguments.

IV. Speculative Evaluation

The compile-time technique suggested in the previous section aids in identifying promising nonstrict arguments of functions, thereby identifying useful speculative parallelism in a program. During the speculative evaluation of a functional program, we need to distinguish between three types of tasks, namely, the following:

1) mandatory tasks,
2) promising speculative tasks, and
3) speculative tasks reducing expressions with low probabilities of requirement.

For the sake of convenience, and also to be able to obtain a fine distinction about the utility of various speculative tasks, argument probabilities are mapped into the priority interval $[0 \cdots 10]$ at compile-time and stored in a function table in the case of argument speculation. In the case of branch speculation, branch probabilities are mapped into the interval $[0 \cdots 10]$ at compile-time and stored within the graphs of conditional expressions. Priority queues corresponding to levels 0 to 10 may be maintained at each processing element. A task to reduce the root expression of the program may be placed in priority queue 10 at processing element 1.

Definition 12—Net Priority: Let a program require the result of an expression E with probability p. Let E require, for its evaluation, the result of an expression e with probability q. Then the (parent) task reducing E may spark a (child) task with a priority proportional to pq. This priority is referred to as the net priority with which the (child) task runs. The following are issues that arise during the creation and progress of tasks.

1) Should speculative tasks with low priorities be created? If indeed they are created, they may seldom get the attention of processors if speculative tasks with higher priorities exist. On what basis should a (threshold) priority value t be chosen so that only those speculative tasks with net priorities greater than or equal to t may be created, so that there are net gains over several runs of a program?
2) If a task is scheduled speculatively and later is found to be useful by its parent, should its priority be upgraded to that of its parent, and should the corresponding changes in priorities be propagated transitively to all its descendants? Similarly, should a task sparked speculatively, but later found to be unnecessary by its parent, be deleted from the system so that the processors engaged by it and its descendants are put to better use? Can these operations be achieved at reasonable costs?

A. Implementation

In this section, we describe the graphical representation of functions, the parallel system simulated, the parallel graph reduction scheme employed, and the scheduling method used by us.

Graph Representation: We have designed our own functional language [5] to experiment with the ideas presented in this paper. The language supports all essential features of a functional language. Graphs as described in [8] are generated for programs written in this language by our compiler.

Parallel Graph Reduction: We have implemented parallel graph reduction (with a global address space) using lazy evaluation and speculative evaluation. The graph reduction model is based on template instantiation [8]. A task is represented by a pointer to the graph being reduced by it. We have used notification model for blocking and resumption [9]. Techniques used by us for task synchronization and other aspects of parallel graph reduction can be found in [3].

Parallel System: We have simulated a shared memory multiprocessor system with N processors and N memories connected by a communication network. Copies of the graphs of functions in a program and a copy of the parallel graph reducer (lazy or speculative) are available with each processing element (PE). The graph reducer programs are written in the C language. Time is accounted for various operations like traveling down the spine, checking the availability of arguments, and instantiation. The graph reducer programs are viewed as a sequence of machine instructions to account for the time to execute instructions. The timings for instruction fetch, data access, and ALU operations correspond to those of Intel 80386 [4]. Delays due to simultaneous requests to a memory unit are taken care of by queuing the requests.

We discuss the relative performance of speculative evaluation and lazy evaluation schemes on a given number of processors in Section IV-B. Time for garbage collection is not accounted in both lazy evaluation and speculative evaluation.

Choice of a Threshold Priority Value: It is desirable that only those expressions whose probability of requirement for the program's evaluation is good are speculatively scheduled, for the following reasons:

1) To ensure net gains due to speculative evaluation of a functional program over several runs;
2) To limit the speculative parallelism generated, particularly when we have only a limited number of processors; and
3) Priority change operations are expensive [5], [9]; hence, expressions with low probabilities of requirement may not be speculatively sparked, because if priority upgradation on demand for tasks evaluating such expressions is not performed, they may not get the attention of processors until all high-priority tasks are completed.

In speculative evaluation with static priorities, only those speculative tasks with net priorities greater than or equal to a chosen threshold priority value are sparked. An expression with a net priority less than the threshold value is reduced only on demand with the priority of the parent requesting its value. A threshold priority value greater than 5 is, in general, good enough for most programs, but may not always be so. The choice of a suitable threshold value is program-specific [5] and depends on the following parameters:

1) Number of processors in the system,
2) Conservative parallelism that may be generated by a program that is data-dependent, and
3) Speculative parallelism that may be generated by expressions that are required for the program's evaluation with different priorities; this also is data-dependent.

Although it is usually the case that speculative evaluation of a promising expression yields benefits, it may not always be so, particularly when the average complexity of the expression when it is not required is much greater than that when it is required.

Overheads Due to Speculation: In speculative evaluation with static priorities, the main overheads are in terms of computing the net priority of a child task. The net priority with which a task is running is stored in a field called the priority field at the task node.

Scheduling: Mandatory tasks run at priority level 10, and speculative tasks run at lower priorities. Each processing element distributes the tasks generated by it onto other processing elements, and also onto itself, in a round-robin manner. Eleven priority queues exist $(0 \cdots 10)$. Tasks are placed in appropriate priority queues as dictated by the net priorities assigned to them. A task to reduce the root expression of the program is placed in priority queue 10 at PE_1 to start with.

Whenever a processing element wants to pick a task for evaluation, it scans the priority queues in order from queue 10 down to queue 1. A lower-priority queue is scanned to pick a task only if no task is available in higher-priority queues. Tasks with priority 0 are not runnable. A nonpreemptive scheduling scheme is used.

The problem of a mandatory task starving for the attention of a processor in the event of a speculative task getting stuck in an infinite loop does not arise, for the following reasons. Nontermination during the evaluation of well-written functional programs arises essentially as a result of an attempt to evaluate the components of an infinite list structure speculatively. However, this problem does not arise in our models of speculative evaluation, because they evaluate the components of a list structure lazily.

Consider the processing elements $PE_1 \cdots PE_n$. Imagine that PE_1 is at the left end and that PE_n is at the right end. As mandatory tasks are picked up in preference to speculative tasks by processors, in order that speculative tasks may get the attention of processors at least initially, mandatory tasks are distributed in a round-robin manner from PE_1 to PE_n, and speculative tasks are distributed from PE_n down to PE_1.

B. Comparisons with the Performance of Lazy Evaluation

To illustrate the use of speculative evaluation, we now present some programs. The programs reported below are run on the simulated multiprocessor mentioned in Section IV-A, using both lazy evaluation and speculative evaluation, and using static priorities with different inputs chosen randomly conforming to branch probabilities. In the case of list inputs, lists with varying lengths and different elements are fed as inputs. Specevalt and Lazyevalt are the average speculative evaluation time and lazy evaluation time for 30 runs of each program, respectively, on a given number of processors. The performance results reported below are the cases in which both lazy evaluation and speculative evaluation exploit parallelism at the fine-grain level.

Branch Speculation:

Program 1

```
h(x, y) == if $x > 0$ then $(h(x-1)y) + 2$
      else
            if $x = 0$ then 0
            else $1 + h\ y\ (-x)$;
psum(low, high) == let
                  mid = ( low + high ) div2
               in
               if low = high then low
            else (psum low mid) + (psum (mid + 1) high)
sum(n) == psum 1 n;
g(x, y, z) == if $(h\ x\ x) = 0$ then *0.1 * sum y else sum z;
```

TABLE I
PERFORMANCE OF BRANCH SPECULATION

Number of PE's	4	8	16	32	64
(Specevalt/Lazyevalt)	0.92	0.9	0.83	0.79	0.77

The above program is contrived, but nevertheless suggests the situations in which branch speculation may be of use. The root expression is an application of the function g. Only the more promising branch (sum z) in g is speculatively evaluated in parallel with the predicate $((h\ x\ x) = 0)$. The time complexity of the predicate is significant for large values of $abs(x)$, but little parallelism is generated during its evaluation. Speculative evaluation of (sum z) results in a progress of its evaluation by the time it is selected. The threshold priority value used to run the above program is 9.

Average speculation overheads per processor for 30 runs are about 3% with four processors, and they decrease with an increase in the number of processors. Because the branch (sum z) is selected in most of the runs of the above program, the average wasted work per processor is negligible.

Performance of Argument Speculation:

Program 2 (Singletons program)

```
          member(a, x) == if (x = []) then *0.1* False
                          else
                              if (a = (hd x)) then *0.5* True
                              else member a (tl x);
            union(x, y) == let
                              z = union (tl x) y
                          in
                          if (x = []) then y
                      else
                              if (member (hd x) z) then z
                              else (hd x) : z;
difference(x, y) == let
                              z = difference (tl x) y
                          in
                          if (x = []) then *0.1* [ ]
                      else
                              if (member (hd x) y) then *0.5 * z
                              else (hd x) : z;
set(s) == if (atom s) then s : []
                      else union (set (hd s)) (set (tl s));
singletons(s) == if (atom s) then s : []
                      else
          union (difference (singletons (hd s)) (set (tl s)))
                (difference (singletons (tl s)) (set (hd s)))
```

Branch probabilities are shown in only those functions having nonstrict arguments. $P(\text{union}_x) = 1.0$; $P(\text{union}_y) = 1.0$; $P(\text{singletons}_s) = 1.0$; $P(\text{set}_s) = 1.0$; $P(\text{difference}_x) = 1.0$; $P(\text{difference}_y) = 0.9$.

In the call to union in the else-branch in the function singletons, both arguments are calls to function difference, and in argument speculation, they are reduced by tasks running with net priority 10. In each one of the calls to difference, the first argument runs with net priority 10, and the second

TABLE II
PERFORMANCE OF ARGUMENT SPECULATION

Number of PE's	4	8	16	32	64
(Specevalt/Lazyevalt)	0.87	0.85	0.84	0.82	0.81

argument runs with net priority 9. The threshold value used to run the program is 9.

Average speculation overheads per processor for 30 runs are about 4% with four processors, and they decrease with an increase in the number of processors. Interestingly, no wasted work is done during the evaluation of this program under speculative evaluation, because the call to the function difference in the body of singletons always needs to reduce its second argument y. However, a strictness analyzer would not be able to detect this.

Reduction in time using argument speculation is due to the fact that when an application of function difference is being reduced, a lazy evaluator spends considerable time evaluating the predicate ($x = [\,]$), where x may be bound to (singletons ($hd\ s$)) or (singletons ($tl\ s$)), before the value of the nonstrict argument y is demanded. We would expect that with an increase in the number of processors, there should be significant reduction in the ratio (specevalt/lazyevalt). But this is not the case, as shown by Table II. Considerable decrease in the ratio (specevalt/lazyevalt) with an increase in the number of processors is possible if the conservative parallelism generated by a program is not much, but the speculative parallelism generated is significant. For the singletons program, there is considerable parallelism generated, even under lazy evaluation.

Using a coarser grain size wherein tasks, whether mandatory or speculative, are sparked only for supercombinator applications, tasking overheads as well as speculation overheads are reduced. As a result, lower ratios of (specevalt/lazyevalt) are observed [5].

C. Priority Change Operations

The goal of minimizing the costs of priority upgradation or deletion of irrelevant tasks is not quite consistent with the necessary conditions that a program needs to satisfy for speculation to perform better than lazy evaluation [5]. This is because if the complexities of predicates in conditionals are significant, speculative evaluation of expressions can give rise to a huge tree of descendant tasks by the time it is known whether they are necessary. Transitively propagating priority change operations at this stage is costly, particularly when the chosen grain size is not coarse. With a coarse grain size, these costs can be reduced considerably.

V. PROBLEMS WITH SPECULATION

In the case of branch speculation, it appears as if nontermination may occur when a recursively defined function like factorial is reduced. However, nontermination does not occur, because the net priorities with which speculative tasks for the recursive branches are created keep decreasing with the depth of recursion, and ultimately they become 0. Note that nontermination occurs in branch speculation only when the (branch) probability, with which the recursive branch is selected in a recursive function, is specified to be 1. Tasks with priority 0 are not runnable.

Consider $f(a) == \text{if } a = 0 \text{ then } 0 \text{ else } 1/a$. Using branch speculation can result in an attempt to perform the operation (1/0). It is possible to allow speculation to be used in situations such as these if the system does not abort the program in error situations such as (1/0), but merely returns a special error value to the calling function. The program may be aborted only if the result computed by a speculative task that causes an error such as (1/0) is actually demanded for the program's evaluation. Similar problems arise in argument speculation, too; but they occur more often in branch speculation.

VI. TERMINATION OF PROGRAMS

As a consequence of the Church–Rosser theorem [8], it follows that the order of evaluation of subexpressions in an expression is irrelevant, because whatever order is chosen, if the evaluation terminates, the normal form reached would be the same. It can be ensured that the evaluators based on branch speculation and argument speculation would terminate on the same set of programs on which a lazy evaluator would. This can be achieved by ensuring that some progress is always made on the normal-order spine, even while evaluating speculative tasks, all of which may not be needed. Nontermination due to speculative evaluation of infinite lists does not arise in our speculation models, because lists are evaluated lazily.

VII. FURTHER DISCUSSION

In this paper, we have proposed two models of speculative evaluation, namely, branch speculation and argument speculation. Programs that satisfy the following properties can run faster using speculative evaluation.

1) The conservative parallelism generated by a program cannot keep all the processors in a system busy.
2) Evaluation of a program using branch speculation performs better than lazy evaluation if it has conditional expressions in some or all of which the complexities of predicates are significant, and in some or all of which one of the branches has a higher probability of selection than the other.

Evaluation of a program using argument speculation performs better if the program has at least some function applications that satisfy the following condition. If a lazy evaluator spends significant time before it may demand the result of a promising nonstrict argument during the reduction of a user-defined function application, and if the complexity of the argument is not trivial, it is worth speculatively evaluating the argument. The extent to which we gain as a result of branch speculation or argument speculation depends not only upon the availability of spare processors but also on the complexities of predicates in conditional expressions.

The overheads due to speculation using a static priority scheme are quite small, and, moreover, speculative tasks, however promising they are, do not compete with mandatory tasks. For these reasons, if there exist highly promising branches or nonstrict arguments in a program, speculative evaluation may be used even if there are only a limited number of processors.

The differences between branch speculation and argument speculation are as follows. Branch speculation can be used even if all functions in a program are strict in all their arguments, but contain conditionals. Also, in branch speculation, larger expressions are scheduled for speculative evaluation, since an entire branch in a conditional expression is scheduled for speculative evaluation. An argument usually is only a subexpression of a branch. Thus, more losses may be incurred in branch speculation, in general, if speculation is incorrect.

The compile-time analysis discussed in Section III is a generalization of traditional strictness analysis and aids in identifying useful speculative parallelism in the case of argument speculation. Recursive functions may need a number of refinements, as discussed in Section III; but since we are interested in only the relative utility of nonstrict arguments for a function's evaluation, usually not more than 10 to 15 refinements are necessary, and these are compile-time overheads. The following are some interesting extensions that are possible:

1) To develop a scheme to dynamically adjust the threshold value,
2) To incorporate speculative evaluation into the parallel G-machine, and
3) To study the possibility of using recurrence equations to derive argument probabilities.

ACKNOWLEDGMENT

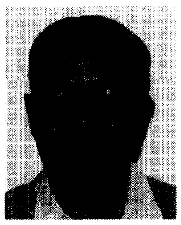
The authors thank K. Ravikanth and P. S. Sastry for their many helpful discussions.

REFERENCES

[1] F.W. Burton, "Speculative computation, parallelism, and functional programming," *IEEE Trans. Comput.*, vol. C-34, pp. 1190–1193, Dec. 1985.
[2] C. Clack and S.L. Peyton Jones, "A practical approach to strictness analysis," in *Functional Programming Languages and Computer Architecture*, in *Lecture Notes in Comput. Sci. 201*. New York: Springer-Verlag, 1985, pp. 190–203.
[3] ——, "The four-stroke reduction engine," *Proc. ACM Conf. Lisp Functional Programming*, 1986, pp. 220–232.
[4] Intel Corp., *Intel 80386 Programmer's Reference Manual.* Santa Clara, CA: 1986.
[5] P.V.R. Murthy, "A study of the implementation of speculative parallelism in functional languages," Ph.D. dissertation, Indian Inst. of Sci., Bangalore, India, 1991.
[6] S. Ono, "Relationships among strictness-related analyses for applicative languages," in K. Fuchi and L. Kott, Eds., *Programming of Future Generation Computers* , vol. II. Amsterdam, Netherlands: Elsevier (North-Holland), 1988, pp. 257–283.
[7] R.B. Osborne, "Speculative computation in Multilisp," in *Parallel Lisp: Languages and Systems,* in T. Ito and R.H. Halstead, Eds., *Lecture Notes in Comput. Sci. 441.* New York: Springer-Verlag, 1989, pp. 103–137.
[8] S.L. Peyton Jones, *The Implementation of Functional Programming Languages.* Englewood Cliffs, NJ: Prentice-Hall, 1987.
[9] ——, "Parallel implementations of functional programming languages,"*Comput. J.*, vol. 32, pp. 175–186, Feb. 1989.
[10] B. Wegbreit, "Mechanical program analysis," *Commun. ACM*, vol. 18, pp. 528–539, Sept. 1975.
[11] E. Wong and B. Hajek, *Stochastic Processes in Engineering Systems*, Prop. 4.1, Ch. 1. New York: Springer-Verlag, 1985.

P. V. R. Murthy received the M.Sc.(Tech.) degree in computer science in 1982 from Birla Institute of Technology and Science, Pilani, India, and the Ph.D. degree in computer science from the Indian Institute of Science, Bangalore, India, in 1991.

He is currently a Senior Scientific Officer at the Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India. His research interests are in parallel processing and programming languages.



V. Rajaraman received the B.Sc. (with honors) in physics from Delhi University, India, in 1952; the DIISC and AIISC degrees from the Indian Institute of Science, Bangalore, India, in 1955 and 1957, repsectively; the S.M. degree in electrical engineering from the Massachusetts Institute of Technology, Cambridge, MA, USA; and the Ph.D. degree from the University of Wisconsin, Madison, WI, USA, in 1961.

Currently, he is IBM Professor of Information Technology at the Jawaharlal Nehru Centre for Advanced Scientific Research Bangalore, India. He was an Assistant Professor of Statistics at the University of Wisconsin in 1961–62. He then joined the Indian Institute of Technology (IIT), Kanpur, India, as an Assistant Professor of Electrical Engineering in 1963, and in 1973 he became a Senior Professor of Electrical Engineering and Computer Science there. He headed the Computer Centre at IIT Kanpur from 1967 until 1972 and again from 1976 until 1979. He also initiated the computer science educational programme and guided its growth from 1966 to 1979. From 1982 to 1994, he was Professor of Computer Science and Chairman of Supercomputer Education and Research Centre at the Indian Institute of Science, Bangalore, India. During 1965–66, he was a Visiting Assistant Professor of Computer Science and Electrical Engineering at the University of California, Berkeley, CA, USA, and during 1972–73, he was a Visiting IBM Research Fellow at the Systems Development Institute, Canberra, Australia. He has also been active as a consultant to industry.

Dr. Rajaraman pioneered computer science education and research in India, and in recognition of this, he was awarded the Shanti Swarup Bhatnagar Prize in 1976 and the Fellowship of the Computer Society of India in 1981. He was awarded the Homi Bhabha Prize for research in applied sciences by the University Grants Commission, India, in 1986, and a National Award for Excellence in Computer Engineering by the Indian Society for Technical Education in 1988. He was a member of the Electronics Commission from 1979 until 1982. He has published many technical papers and is the author of 12 books on computer science. He is a Fellow of the Indian National Science Academy, the Indian Academy of Sciences, and the Indian National Academy of Engineering. He is an Honorary Professor at the Indian Institute of Science, Bangalore, India.