Symbolic Performance Prediction of Data-Dependent Parallel Programs

Hasyim Gautama and Arjan J.C. van Gemund

Dept. of Information Technology and Systems
Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft, The Netherlands
{H.Gautama, A.J.C.vanGemund}@ITS.TUDelft.NL

Abstract. Analytically predicting the performance of data-dependent programs is an extremely challenging problem. Even for a fixed problem size the variety of typical input data sets may cause a considerable execution time variance. Especially for time-critical applications, merely predicting the mean execution time does not suffice and knowledge of the execution time distribution is essential. In this paper we present a compositional analytic method to approximate the first four statistical moments of the program execution time in terms of the first four moments of the loop bounds, the branch conditions, and the execution time delays of the constituent tasks. The approach applies to sequential and parallel programs of which the associated DAG has a series-parallel structure. For each binary or N-ary sequential, parallel, or conditional composition the solution complexity is merely $O(1)$. The method is exact for sequential and conditional composition. Furthermore, for N-ary parallel compositions experimental results of synthetic and real workloads show that the actual prediction error of the moments method is in the percent range. The analytic method is implemented in terms of the performance modeling language Pamela. Apart from the theory, in this paper we also present a symbolic Pamela compiler. Provided with a Pamela process model of the (parallel) program, the Pamela compiler symbolically translates this source to closed-form expressions that express the first four moments of the program execution time.

1 Introduction

Static approaches to program performance prediction aim to provide an analytic, compositional model that predicts program execution time in terms of program parameters such as loop bounds, branching probabilities, and basic instruction workload models in order to maximize diagnostic insight in program performance. For instance, consider the following loop

```
for (i=1; i<=N; i++)
    if (C)
        S;
```

T. Field et al. (Eds.): TOOLS 2002, LNCS 2324, pp. 259–278, 2002.
© Springer-Verlag Berlin Heidelberg 2002

that contains a branch and a statement (block). Assuming that the truth probability of the branch condition is p, and the average workload of statement `S` is X, both profiled over a representative set of input data vectors, the program execution time Y is predicted by

$$Y = \sum_{i=1}^{N} pX = NpX \tag{1}$$

While Eq. (1) is correct in terms of its mean value $\mathsf{E}[Y]$, the model is inadequate to predict the execution time *distribution* of Y over the whole spectrum of input data sets. However, for highly data-dependent programs, such as sorting programs and simulation programs, knowledge about the distribution of Y can be crucial, for instance in time-critical (e.g., real-time) applications where only some percentage of runs (or none) is allowed to exceed some execution time threshold. In the above example, the variance of Y, being determined by N, p, and X, can be quite significant. Hence program performance prediction using stochastic parameters is more effective and realistic than using deterministic parameters [23].

There are a number of probabilistic approaches to program execution time analysis, the specific trade-off between analysis cost and prediction accuracy depending on how workload distributions are represented. For instance, consider the sequential composition of two statements with execution time X_1 and X_2, respectively, the total execution time being $Y = X_1 + X_2$. If the probability density function (pdf) of X_1 and X_2 is given, solving Y requires a convolution of pdf(X_1) and pdf(X_2). While the solution is exact, the complexity involved with program-wide convolution of anything but the simplest of distributions prohibits practical application. In addition, the fact that over the whole *spectrum* of input data sets loop bounds and branching behavior also typically exhibit *stochastic* behavior, considerably complicates the analysis [1,22].

Aimed at balancing prediction accuracy and solution complexity, many approaches have been proposed to characterize probability densities, ranging from the use of specific distributions or series approximations to the use of mean and variance. Recently, an analytic approach has been presented where the distributions are represented in terms of the first four moments [4,5,6]. The method permits the first four moments of the execution time $\mathsf{E}[Y^r]$ to be analytically expressed at $O(1)$ solution complexity in terms of the first four moments of the basic block execution times, sequential loop bounds, and branch truth probability values. For parallel compositions the analysis is approximate, though equally low-cost. This low-cost, parametric approach provides rapid, diagnostic insight in execution time distribution of (parallel) applications during the first design stages where low cost still outweighes accuracy.

While previous papers have focused on particular compositions (e.g., sequential [4], parallel [5,6]), in the current paper we give an overall presentation of the moments analysis technique. In particular, we present an implementation of the methodology in terms of a process-algebraic modeling language and compiler. The language is called PAMELA (PerformAnce ModEling LAnguage [8]) of which

a prototype version based on the use of deterministic variables has been presented at the 1995 instance of this Conference [9]. In contrast however, the current language version fully supports the use of *stochastic* variables. Most notably, the methodology is now also supported by a public-domain research *compiler* that automatically translates a Pamela process model into an analytic model that predicts the first four moments of the execution time at minimum cost.

The paper is organized as follows. In Section 2 we review related approaches towards predicting the execution time distribution of sequential, conditional, and parallel task compositions. In Section 3 we summarize our moments analysis for sequential, conditional, and parallel compositions, and report on the accuracy and complexity of our analysis. In Section 4 we present a number of case studies involving the execution time prediction of real (parallel) programs in order to show the accuracy of our technique in practice. In Section 5 we present the language and compiler implementation, including two example applications. Finally, in Section 6 we draw our conclusions.

2 Related Work

In this section we review related approaches towards analytically predicting the execution time distribution of sequential, conditional, and parallel task compositions. An approach using the pdf is introduced by Gelenbe [7] to determine the completion times of block-structured parallel programs (SP graphs). However, the high cost numerical integration prohibits practical use. Lester [15] uses the z-transform to approximate the pdf. While the real pdf can be approximated well, the solution complexity of the underlying numeric process is still high.

To decrease solution complexity, there have been a number of approaches based on the use of representations other than the pdf, where generality is traded for cost. Some approaches introduce restrictions to specific distributions which are characterized by a limited number of parameters. Sahner and Trivedi [21] use exponomial distributions. While proven a powerful tool for reliability modeling and analytical performance analysis, exponomial workloads are hardly measured in real programs. Aimed at analyzing parallelism (order statistics) Kruskal and Weiss [14] use increasing failure rate (IFR) distributions while Axelrod [2], Gumbel [11], Madala and Sinclair [17], and Robinson [20] use symmetric distributions. Although the approximation error is quite reasonable, only the first moment can be obtained. As mean and variance are required it is impossible to analyze applications with nested parallelism. Sötz [26] uses an exponential distribution combined with a deterministic offset. While the analysis is straightforward, the approach may introduce significant errors. Sarkar [22] determines the mean and variance for sequential compositions only. Schopf and Berman [24] use normal distributions. While the application to sequential programs is straightforward, binary parallelism is approximated heuristically so that the application to N-ary parallelism would cause severe errors.

Lüthi *et al.* [16] characterize parameter variabilities in terms of histograms. In contrast to program (task graph) analysis they address the problem of solv-

ing queuing models with load variabilities. Also Schopf and Berman [24] use histograms with a limited number of intervals. However, the analysis complexity grows rapidly with the number of histogram intervals needed to accurately characterize a distribution.

Another way of characterizing the pdf is based on series approximation, for example the Gram-Charlier series of type A [5]. While the analysis is asymptotically exact, the number of Gram-Charlier terms needed for a sufficiently accurate approximation is prohibitive.

In summary, the above approaches deal with sequential and parallel compositions. Some impose restrictions on the allowable distributions. Some allow a mere characterization in terms of mean and variance. None of the approaches addresses the effect of stochastic loop bounds or branching, which are essential in stochastic program modeling. An exception is the approach taken by Adve and Vernon [1] who allow a sequential loop bound to be stochastic.

Table 1 summarizes the related work in terms of the distribution type used, and whether the approach addresses sequential composition (SC), and stochastic loop bounds (LB), condition probabilities (CP), binary parallel composition (BP) and N-ary parallel composition (NP). Our approach is included for reference.

Table 1. Summary of related work from program analysis perspective.

First author	Distr. type	SC	LB	CP	BP	NP
Adve [1]	Mean & Var	-	+	-	-	-
Axelrod [2]	Normal	-	-	-	-	+
Gautama [5]	Series	-	-	-	+	+
Gautama (this paper)	Moments (4)	+	+	+	+	+
Gelenbe [7]	PDF	+	-	-	+	+
Gumbel [11]	Normal	-	-	-	-	+
Kruskal [14]	IFR	-	-	-	-	+
Lester [15]	Z-transform	+	-	-	-	+
Madala [17]	Normal	-	-	-	-	+
Robinson [20]	Normal	-	-	-	-	+
Sahner [21]	Exponomial	+	-	-	+	+
Sarkar [22]	Mean & Var	+	-	-	-	-
Schopf [24]	Histogram	+	-	-	+	-
Schopf [24]	Normal	+	-	-	+	-
Sötz [26]	Det & Exp	+	-	-	+	+
Lüthi [16]	Histogram	Queuing network				

3 Analysis

In order to provide a proper understanding of our performance prediction approach, in this section we summarize the moments analysis corresponding to sequential, conditional and parallel compositions. Due to space limitations only the main results are presented while the interested reader is referred to [3,4,5,6].

The moments analysis represents a stochastic variable X by a 4-tuple denoting the first four central moments $X = (\mathsf{E}[X], \mathrm{Var}[X], \mathrm{Skw}[X], \mathrm{Kur}[X])$, i.e., the mean, variance, skewness and kurtosis of X, respectively. When X is deterministic ($\mathrm{Var}[X] = 0$), the tuple simply becomes $X = (\mathsf{E}[X], 0, 0, 3)$. A limited number of moments are incorporated in the analysis since higher moments are less important to the accuracy of X. Moreover, evaluation cost is significantly reduced.

The four moment values can distinguish between well-known standard distributions. For example an exponentially distributed random variable X with parameter μ is expressed by $X = (\mu, \mu^2, 2, 9)$, a uniformly distributed X with sample space $[a, b]$ is expressed by $X = ((a + b)/2, (b - a)^2/12, 0, 1.8)$, and a normally distributed X with parameters $[\mu, \sigma]$ is expressed by $X = (\mu, \sigma^2, 0, 3)$. Other distribution types can be distinguished from the skewness and kurtosis values. For any valid empirical workload evaluating the four moment values is straightforward without *a priori* assumption to a certain distribution.

In the following we summarize the analysis of the moments of composition's execution time Y in terms of its constituent parts X_i. For readability the formulae are presented in terms of rth moments ($\mathsf{E}[X^r]$) rather than the central moments and independence assumptions are made wherever required.

3.1 Sequential Composition

Consider an addition of two random variables $Y = X_1 + X_2$. The moments of Y are given by [4]

$$\mathsf{E}[Y^r] = \sum_{j}^{r} \binom{r}{j} \mathsf{E}[X_1^{r-j}] \mathsf{E}[X_2^j] \tag{2}$$

For N additions of a random variable X given by $Y = \sum_{i=1}^{N} X$, it holds [4]

$$\mathsf{E}[Y^r] = \mathsf{E}\left[\frac{\mathrm{d}^r}{\mathrm{d}t^r} \left(\sum_{j=0}^{r} \frac{t^j \mathsf{E}[X^j]}{j!} \right)^N \Bigg|_{t=0} \right] \tag{3}$$

where N may be stochastic. In particular, the mean and variance of Y in Eq. (3) are given by

$$\mathsf{E}[Y] = \mathsf{E}[N]\mathsf{E}[X] \tag{4}$$

$$\mathrm{Var}[Y] = \mathsf{E}[N]\mathrm{Var}[X] + \mathsf{E}[X]^2\mathrm{Var}[N] \tag{5}$$

while the skewness and kurtosis of Y are according to

$$\begin{aligned}
\mathrm{Skw}[Y] = {} & (\mathsf{E}[N]\mathrm{Skw}[X]\mathrm{Std}[X]^3 + \mathsf{E}[X]^3\mathrm{Skw}[N]\mathrm{Std}[N]^3 + \\
& 3\mathsf{E}[X]\mathrm{Var}[X]\mathrm{Var}[N])/\mathrm{Std}[Y]^3 && (6)\\
\mathrm{Kur}[Y] = {} & (\mathsf{E}[N]\mathrm{Var}[X]^2(\mathrm{Kur}[X]-3) + \mathsf{E}[X]^4\mathrm{Kur}[N]\mathrm{Var}[N]^2 + \\
& 6\mathsf{E}[X]^2\mathrm{Var}[X](\mathrm{Skw}[N]\mathrm{Std}[N]^3 + \ \mathsf{E}[N]\mathrm{Var}[N]) + \\
& \mathsf{E}[N]\mathrm{Var}[N]) + 4\mathsf{E}[X]\mathrm{Var}[N]\mathrm{Skw}[X]\mathrm{Std}[X]^3 + \\
& 3\mathrm{Var}[X]^2(\mathsf{E}[N]^2 + \mathrm{Var}[N]))/\mathrm{Std}[Y]^4 && (7)
\end{aligned}$$

where $\mathrm{Std}[X]$ denotes the standard deviation of X. Since Eqs. (2) and (3) are exact, we present no experimental results. More details can be found in [4].

3.2 Conditional Composition

Consider a random variable Y which is defined by $Y = X$ if $C = \mathrm{true}$ and 0 otherwise, where C has the truth probability P. Then the moments of Y are given by [4]

$$\mathsf{E}[Y^r] = \mathsf{E}\left[\frac{\mathrm{d}^r}{\mathrm{d}t^r} \left(\sum_{j=0}^{r} \frac{t^j \mathsf{E}[X^j]}{j!} \right)^P \Bigg|_{t=0} \right] \tag{8}$$

Similar to Eq. (3), Eq. (8) can also be expressed in terms of $\mathsf{E}[P^r]$ as given in Eqs. (4) to (7) when N is replaced by P.

Modeling stochastic branch behavior focuses on modeling (measuring) P. Conventional branch modeling implicitly assumes P to be a Bernoulli trial with a *deterministic* parameter p that does not model input data variability. As the memoryless Bernoulli process does not model typical branch behavior, we therefore use an alternative statistical model based on the use of Alternating Renewal Processes (ARP). The ARP model allows branching probability to have a wide range of distributions across the space of input data sets. Instead of measuring P in terms of a deterministic Bernoulli parameter p ("average truth probability"), we measure P in terms of U and D which are random variables denoting the length of consecutive true and false branch invocations, respectively. In our ARP approach we model the branch truth probability P by beta distributions whose moments are given by

$$\mathsf{E}[P^r] = \frac{\Gamma(a+b)\Gamma(a+r)}{\Gamma(a)\Gamma(a+b+r)} \tag{9}$$

where Γ denotes the gamma function, and

$$a = \frac{(\mathsf{E}[D]^2(\mathsf{E}[U] - \mathrm{Var}[U]) + \mathsf{E}[U]^2(\mathsf{E}[D] - \mathrm{Var}[D]))\mathsf{E}[U]}{(\mathsf{E}[D] + \mathsf{E}[U])(\mathsf{E}[D]^2\mathrm{Var}[U] + \mathsf{E}[U]^2\mathrm{Var}[D])}, \quad b = \frac{\mathsf{E}[D]}{\mathsf{E}[U]}\, a. \tag{10}$$

This approach is completely compatible with our moment analysis. Experiments reported in [3] show an accuracy improvement by an order of magnitude compared to Bernoulli modeling approach. More details can be found in [3].

3.3 Parallel Composition

Consider the N-ary parallel composition corresponding to $Y = \max_{i=1}^{N} X_i$ where X_i are N identical and independent distributed (iid) variates of X. Due to integration problems, however, there is no exact explicit expression for $\mathsf{E}[Y^r]$ in terms of $\mathsf{E}[X^r]$. To obtain an explicit solution in terms of $\mathsf{E}[X^r]$ we first express X in terms of the lambda distribution defined by the percentile function R as function of the cummulative distribution function (cdf), $0 \le F \le 1$, according to [12]

$$X = R_X(F) = \lambda_1 + \frac{F^{\lambda_3} - (1-F)^{\lambda_4}}{\lambda_2} \tag{11}$$

where λ_1 is a location parameter, λ_2 is a scale parameter and λ_3 and λ_4 are shape parameters. The λ values are a simple function of $\mathsf{E}[X^r]$. Without loss of generality let $\lambda_1 = 0$. Due to the inverse formulation of F in terms of Eq. (11), $\mathsf{E}[Y^r]$ can now be explicitly derived in terms of $\mathsf{E}[X^r]$ (through λ_j) according to [5]

$$\mathsf{E}[Y^r] = \frac{N}{\lambda_2^r} \sum_{i=0}^{r} \binom{r}{i} (-1)^i \mathrm{B}(\lambda_3(r-i) + N, \lambda_4 i + 1) \tag{12}$$

where B denotes the beta function. More details can be found in [5].

The binary parallel composition, corresponding to $Y = \max(X_1, X_2)$, can also be analytically treated by first modeling X_1 and X_2 in terms of lambda distributions. In this case the moments of Y are approximated by [6]

$$\mathsf{E}[Y^r] = \int_0^1 (\max(R_{X_1}(F), R_{X_2}(F)) + \Delta x)^r \mathrm{d}F \tag{13}$$

where Δx denotes the difference between $R_Y(F)$ and $\max(R_{X_1}(F), R_{X_2}(F))$. The worst case relative error of $\mathsf{E}[Y]$ in Eq. (13) is 35% which occurs for equal workloads, while sharply decreasing for diverging workloads. Furthermore, measurements using real programs suggest that the worst case error may actually be much less (approximately 5%). Note that Eq. (13) only requires integration on a fixed bound of interval $[0, 1]$. For N-ary parallel composition experimental results on synthetic as well as empirical iid distributions show that the relative error is in the percent range, even for large N (e.g., 4% for $N = 10^4$). More details can be found in [5,6].

4 Experiments

To illustrate the application of our moment analysis approach, in this section we describe the results of the moment analysis approach for three small example codes. The first code features a simple branch condition which is memoryless. The second code features a branch condition that has memory. While the above codes are sequential examples, the third code features two data parallel loops which exhibit various degrees of workload correlation. For all codes, the measured distributions Y_m (moments, measured execution times) are based on running

the program for 6,000 input data sets while the execution time is determined through counter-based program instrumentation. The predicted execution time Y_p is directly based on Eq. (2) to Eq. (13). The prediction error for the rth moment is defined by

$$\varepsilon_r = |Y_m^r - Y_p^r|/Y_m^r \tag{14}$$

The results show that ε_r is in the percent range as long as branch and workload correlation effects are small.

4.1 Sparse Vector Scaling

In this section we apply the moment method to the example loop which has been described in Section 1, instantiated to a sparse vector scaling routine where `C` equals to `x[i] != 0` and `S` is equal to `x[i] *= alpha` where `alpha` is a scaling factor. Let the execution time of interest be given by the multiplication time between `x[i]` and `alpha` which is assumed to take unit execution time while ignoring other program contributions.

In the experiment the elements of vector `x` are generated according to a Bernoulli distribution with probability $p = 0.1$ to resemble data sets with average density $p = 0.1$. The moments of N and P are given in Table 2. Applying Eqs. (8) and (3) yields an execution time prediction Y_p which is compared to the measured execution time Y_m in Table 3. The table shows that $\mathsf{E}[Y_p]$ is approximately exact while the relative error of $\mathrm{Var}[Y_p]$ is 1% due to measurement inaccuracies.

Table 2. $\mathsf{E}[N^r]$ and $\mathsf{E}[P^r]$ for vector scaling.

X	$\mathsf{E}[X]$	$\mathrm{Var}[X]$	$\mathrm{Skw}[X]$	$\mathrm{Kur}[X]$
N	10,000	0	0	3
P	0.1	$9\ 10^{-2}$	2.7	8.1

Table 3. $\mathsf{E}[Y^r]$ for vector scaling.

Y	$\mathsf{E}[Y]$	$\mathrm{Var}[Y]$	$\mathrm{Skw}[Y]$	$\mathrm{Kur}[Y]$
Y_m	999	891	0.0	3.0
Y_p	1,000	901	0.0	3.0

4.2 Straight Selection Sort

Straight Selection Sort sorts an N elements vector `x` according to

```
for (i=N; i>=2; i--) {
    k = i;
    for (j=i-1; j>=1; j--)
        if (x[j]>x[k])
            k = j;
    swap(x[i],x[k]);
}
```

where `swap` exchanges the value between two elements. To define program performance we choose the statement `k = j` to take unit execution time while the

rest of the constructs are assumed to take zero execution time. Although in practice the statement `k = j` will not represent any significant execution time, this choice provides the most interesting (i.e., stochastic) scenario.

For this experiment we let $N = 1,000$ and the elements of array `x` are generated uniformly with sample space $[0, 1]$. The measured input parameters are shown in Table 4 while the predicted and measured execution times are shown in Table 5. The Y_p value is determined by evaluating Eqs. (8) and (3). While ε_1 is very small, the relative variance error is 28%, which is caused by correlation of the branch condition between different invocations. While traditional branch models such as a Bernoulli trial would yield variance errors of 74%, our ARP model provides much better estimates [3].

Table 4. $\mathsf{E}[N^r]$ and $\mathsf{E}[P^r]$ for selection sort.

X	$\mathsf{E}[X]$	Var$[X]$	Skw$[X]$	Kur$[X]$
N	1000	0	0	3
P	$1.1\ 10^{-2}$	$3.0\ 10^{-2}$	8.3	60

Table 5. $\mathsf{E}[Y^r]$ for selection sort.

Y	$\mathsf{E}[Y]$	Var$[Y]$	Skw$[Y]$	Kur$[Y]$
Y_m	5,492	21,143	-0.1	3.0
Y_p	5,477	15,126	0.0	3.0

4.3 PSRS

PSRS (Parallel Sorting by Regular Sampling [25]) sorts an N elements vector `x` into a vector `y`. A pseudo code of PSRS for P processors is as follows

```
forall (p=0;p<P;p++)
    sort1(x[p*N/P;(p+1)*N/P-1]);
<select pivots>
<disjoin>
forall (p=0;p<N;p++)
    sort2(y[p*N/P;(p+1)*N/P-1]);
```

The algorithm comprises two data parallel sorting loops. The first parallel section (using Quicksort), called `sort1`, divides the array in P equal subarrays, and sorts each partition in parallel, after which a number of global pivots are determined. Based in these pivots, the subarray is redivided in P parts (`disjoin`), after which P new partitions are created based on the pivots index. The second section (using Mergesort), called `sort2`, sorts the new partitions, after which a sorted array is obtained. The working of `sort1` implies that each task has an iid workload, as we generate the input array with a uniform random variable using a sample space $[0, 1]$. In contrast, the task workloads in `sort2` are correlated, since their workload depends on the common global pivot values. As workload X_i we choose the number of floating point operations per task.

In the experiments we assign each processor a constant array length $N/P = 1,000$ while P ranges from 2 to 128. In Figs. 1 and 2 we show ε_r for `sort1` and `sort2`, respectively. The results of the Gumbel method [11] are included for

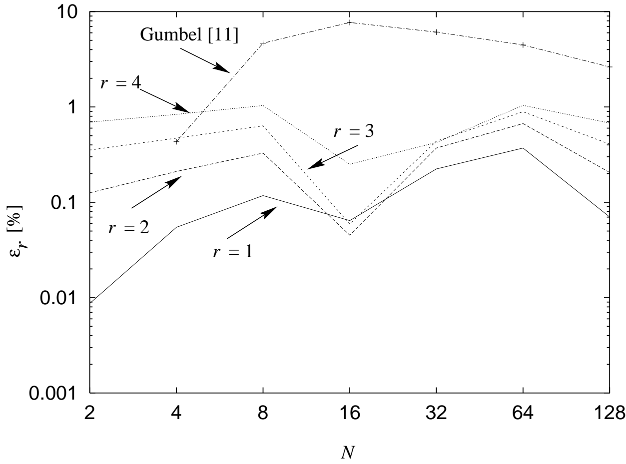

Fig. 1. Error [%] for `sort1`.

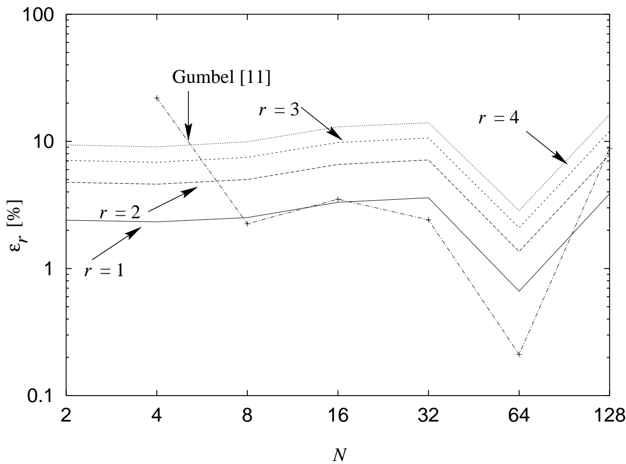

Fig. 2. Error [%] for `sort2`.

comparison. While `sort1` produces excellent accuracy, the correlation effects in `sort2` clearly degrade ε_r.

In summary, the above 3 experiments show that the moments analysis yields quite an acceptable accuracy in view of the $O(1)$ solution complexity of our symbolic analysis technique, while outperforming comparable low-cost analysis methods. More example codes and results can be found in [5,6].

5 Tool Implementation

The moments analysis method has been implemented in terms of a performance modeling language called PAMELA (PerformAnce ModEling LAnguage). The language allows an application to be described in terms of a process-algebraic, performance simulation model which is subsequently compiled to an analytic execution time model. An initial version of the language and associated analysis technique has been presented in [9], which was restricted to deterministic variables only. In this section we present the new version that supports the statistical moments analysis technique, featuring a compiler that implements the symbolic analysis technique. The compiler is available in the public domain [18].

5.1 Modeling Language

PAMELA is a process-algebraic language that allows a parallel program-machine combination to be modeled in terms of a sequential, conditional, and parallel composition of processes, modeling workload, condition synchronization, and resource contention. Work is described by the `use` *process*. The construct `use(r,t)` exclusively acquires service from *resource* `r` for `t` units time (excluding possible queuing delay). The scheduling policy that is currently supported is FCFS with non-deterministic conflict arbitration. The service time `t` may be deterministic or stochastic. In the latter case, its distribution is expressed in terms of the first four moments. A resource `r` has a multiplicity that may be larger than 1. As in queuing networks, it is convenient to define an infinite-server resource called `rho` that has infinite multiplicity. Instead of writing `use(rho,t)` we will simply write `delay(t)`.

PAMELA features the following process composition operators:

- `;` for binary sequential composition,
- `seq (<index> = <lb>, <ub>)` for N-ary sequential composition,
- `||` for binary parallel composition,
- `par (<index> = <lb>, <ub>)` for N-ary parallel composition,
- `if (<cond>) [else]` for conditional composition.

PAMELA is a strongly typed language. Variables can be of three types: `process`, `resource`, or `numeric`. The latter type is used for time expressions, parameters, indices, and loop bounds. Each lhs variable can have a formal parameter list. The scope of these parameters is limited to the rhs expression.

The following PAMELA equations model a Machine Repair Model (MRM) in which `P` clients either spend a mean time `t_l` on local processing, or request service from a server `s` with service time `t_s` with a total cycle count of `N` iterations (unlike steady-state analysis, in our approach we require models to terminate; however `N` may be kept symbolic).

```
%------------------------------------------------------------------
% mrm.pam -- Machine Repair Model
%------------------------------------------------------------------
numeric exponential(mu) = moments(mu,mu*mu,2,9)  % appr exp distr

numeric P = 1000                                % # clients
numeric N = 1000000                             % # iterations
numeric t_l = exponential(10)                   % exp distr (mu = 10)
numeric t_s = exponential(0.1)                  % exp distr (mu = 0.1)

resource s = fcfs(0,1)    % fcfs is a predefined FCFS resource array
                          % 1st arg index, 2nd arg its multiplicity

process main =   par (p = 1, P)                 % fork P clients
                     seq (i = 1, N) {           % each loops N times
                         delay(t_l) ;
                         use(s,t_s)
                     }
```

The example illustrates the top-down, *material-oriented* modeling approach [13] taken in PAMELA, in which the server is modeled by a *passive* resource. In a *machine-oriented* approach (the dual modeling paradigm), the server would have been modeled by a separate *process* that synchronizes through explicit message-passing. Unlike our approach however, the latter paradigm is not amenable to a mechanic analysis process where a PAMELA model is *automatically compiled* to analytic execution time expressions. This is due to the inherent difference in modeling condition synchronization and mutual exclusion. In message-passing paradigms both forms of synchronizations are implicitly expressed through the *same* constructs, i.e., synchronous communication, combined with a non-deterministic choice operator (selective communications). This makes it impossible to symbolically and mechanically express the separate timing effects of both synchronization types, which is the basis of our analytic approach. In the PAMELA top-down modeling approach, problem parallelism (including condition synchronization) is modeled explicitly in terms of `par` (or `;`) operators, explicitly constrained by mutual exclusion (`use`) as a result of scheduling when the processes need to share resources such as, e.g., software locks, file servers, processors, communication links, memories, and/or I/O disk handlers, The PAMELA modeling paradigm is further discussed in [10].

Due to the ultra-low solution complexity of the performance model high parameter values can be specified as the resulting models typically evaluate within a second. When no numeric value is specified for a variable, this implies that the parameter will also occur in the symbolic performance model. This language feature is described in the next section.

As in ordinary mathematics, the semantics of a PAMELA model is based on expression substitution. Although for readability a model may be coded in terms of many equations, internally each expression is evaluated by recursively substituting every global variable by its corresponding rhs expression. Consequently,

the above MRM model is internally rewritten to the equivalent normal-form model below (relevant equations shown only).

```
numeric t_l = moments(10,100,2,9)
numeric t_s = moments(0.1,0.01,2,9)

process main =   par (p = 1, 1000)
                     seq (i = 1, 100000) {
                         delay(moments(10,100,2,9)) ;
                         use(fcfs(0,1),moments(0.1,0.01,2,9))
                     }
```

Apart from the process operators mentioned above, Pamela includes the usual unary and binary numeric operators such as `+`, `*`, `/`, `mod`, `div`, `==`, `<`, `max`, etc., as well as the reduction operators `sum (<index> = <lb>, <ub>)` and `max (<index> = <lb>, <ub>)`. Conditional numeric expressions are described using `if-then` just like conditional process expressions. Furthermore, as parts of the analysis result is expressed in terms of vectors, the `numeric` abstract data-type includes vectors as well as scalars, implying that all numeric operators are overloaded. A vector is denoted `[<scalar>, ..., <scalar>]`. Hence, the expression `[1,2,3] * 4` is legal and, incidentally, will be compiled to `[4,8,12]`. In order to generate unbounded, symbolic vectors Pamela features the `unitvec` operator which returns a unit vector in the dimension (base 0) given by its argument. For instance, the expression `10 * unitvec(3)` will be compiled to `[0,0,0,10]`.

5.2 Symbolic Compilation

A Pamela model is translated to a time-domain performance model by substituting every `process` equation by a `numeric` equation that models the execution time associated with the original process. The lhs is derived from the original lhs by prefixing `T_`. Thus the cost model of a process expression `main` is denoted `T_main`. The result is a Pamela model that only comprises `numeric` equations as the original `process` and `resource` equations are no longer relevant. The fact that the cost model is again a Pamela model is for reasons of convenience as explained later on.

The analytic approach underlying the translation process is based on a combination of critical path analysis of the delays due to condition synchronization ("task synchronization"), and a lower bound approximation of the delays due to mutual exclusion synchronization ("queuing delay") as a result of resource contention. Per `process` equation four `numeric` equations are generated, whose lhs identifiers are derived from the original process variable by prefixing specific strings. Let `L` denote the lhs of a process equation. The first equation generated is `phi_L` which computes the effect of condition synchronization. The second equation generated is `delta_L` which computes the aggregate workload per resource (index). The combined mutual exclusion delay is computed by the third equation denoted `omega_L`. The three above equations are combined into the fourth equation `T_L` which denotes the execution time. A more detailed background can be found in [9].

Returning to the MRM example, the PAMELA model of the MRM is compiled to the following time domain model (T_main shown only):

```
numeric T_main = max(max (p = 1, 1000) {
                          sum (i = 1, 1000000) {
                              (moments(10,100,2,9) +
                               moments(0.1,0.01,2,9))
                          }
                     },max(sum (p = 1, 1000) {
                                sum (i = 1, 1000000) {
                                    moments(0.1,0.01,2,9) * unitvec(0)
                                }
                     })
```

Since all parameters are numerically bound, this intermediate result is automatically reduced (evaluated) to

```
numeric T_main = moments(1.01+07,4.11+03,4.69-01,3.23+00)
```

During this evaluation process Eqs. (2), (3), and (12) are used.

5.3 Parameterization

By virtue of the symbolic nature of the analysis process, PAMELA process models are typically parameterized, thus preserving parameters into the resulting time domain model. In order to enable parameterization PAMELA supports the `parameter` modifier, which blocks the substitution process from attempting to substitute a rhs expression for the particular variable that is intended to become a parameter. Consider the MRM model presented earlier. As all variables were bound to numeric values the compiler evaluates the model for the given values of P, N, t_l, and t_c. In order to investigate the effect of, say, P and N, we would redefine these variables according to

```
numeric parameter P
numeric parameter N
```

Symbolic compilation now yields the intermediate result (T_main shown only)

```
numeric T_main = max(max (p = 1, P) {
                          sum (i = 1, N) {
                              (moments(10,100,2,9) +
                               moments(0.1,0.01,2,9))
                          }
                     },max(sum (p = 1, P) {
                                sum (i = 1, N) {
                                    moments(0.1,0.01,2,9) * unitvec(0)
                                }
                     }))
```

Although parameterized, this model is automatically further reduced as a result of simple transformation rules such as

```
sum (i = a, b) scalar = (b-a+1) * scalar
```

yielding the following result symbolic performance model:

```
numeric T_main = max(max (p = 1, P) {
                             N * moments(10.1,100.01,2,9)
                        },(P * N * (moments(0.1,0.01,2,9))))
```

This model can be evaluated for different values of P and N, possibly using mathematical tools other than the PAMELA compiler. In PAMELA further evaluation is achieved as described earlier, by recompiling the above result after removing one or both `parameter` modifiers and providing a numeric rhs expression.

In order to assess the prediction quality of the above result, we derive the mean cycle time of the MRM as function of P by dividing `T_main` by N. The resulting prediction T_{MOM} = `T_main/N` is compared with the result T_{MVA} of Mean Value Analysis [19] (as `t_l` and `t_s` are approximately exponential) as shown in Table 6.

Table 6. MRM mean cycle time.

P	1	2	5	10	20	50	100	200	500
T_{MVA}	10.10	10.10	10.11	10.12	10.12	10.19	10.82	20	50
T_{MOM}	10.10	10.11	10.12	10.12	10.12	10.12	10.13	20	50

Table 6 illustrates the effects of the low-cost, lower bound approximation in PAMELA of the effects of mutual exclusion (queuing). While the prediction error ε_1 for $P = 0$ and $P \to \infty$ is zero, near to the saturation point ($P = 100$) the error is around 8%. Experiments [10] indicate that for very large systems ($O(1000)$ resources) the worst case average error is limited to 50%. However, these situations seldom occur as typically systems are either dominated by condition synchronization or mutual exclusion, in which case the approximation error is in the percent range.

Note that in this particular example the introduction of variance in `t_l` and `t_s` has a negligible effect on the prediction. For small P the prediction values are slightly higher than the deterministic lower bound $T = t_l + t_s$ which is caused by the order statistics effect computed by Eq. (12).

Given the ultra-low solution complexity, the accuracy provided by the compiler is quite acceptable in scenarios where a user conducts, e.g., application scalability studies as a function of various machine parameters, to obtain an initial assessment of the parameter sensitivities of the application. In particular, note that on a Pentium II 350 MHz the symbolic performance model of the MRM merely requires 0.9 CPU s per point in Table 6 (irrespective of N and P) where most of the CPU time is required to evaluate Eqs. (12) and (13). In contrast, the evaluation of the intermediate model would take approximately 22,500 CPU s where most of the CPU time is required to repeatedly evaluate Eq. (2) for `P*N` times.

5.4 Modeling Example

We end this section with an example of how PSRS can be modeled using PAMELA when run on a shared-memory architecture. In contrast to the exposition in Section 4.3, which merely focused on the prediction error of the two parallel sorting sections (`sort1` and `sort2`), we now model the entire algorithm, including the effect of communication delays.

In the PAMELA model of PSRS, given in Fig. 3, some numeric variables are declared in the first seven lines of the model, i.e., the array length `N`, the processor number `P`. The moments of `c1`, `c2` and `c3` are obtained from profiling. Note that `c` in `sort` must be profiled separately for each function call. The next lines declare the CPU resources and the shared memory resource. Although in the current model the workload of each parallel process is mapped onto a unique CPU (see the `flop` model), the explicit use of CPU resources allows the modeler to also investigate the effects of multithreading. Note that the `shmem` multiplicity is currently set to a value such that no contention will ever occur. The evaluation of the model starts from the `main` process which consists of 3 parallel and 2 sequential sections. In order to simplify the modeling example, we have chosen straight selection sort (described in Section 4.2) instead of quicksort (which requires a much larger sequential model). The `main` process calls the other 3 processes, i.e., the program (sub)model `sort` (straight selection sort), and the machine models `flop` and `move` which model the floating point operation, and the shared memory load/store operation, respectively. The last 2 processes represent workloads with exponential time delay `t_f` and `t_m`. Note, that the `move` interface allows the possibility of modeling, e.g., memory contention (e.g., `use(mem,t_m)` for a single memory bank), without requiring further model modification.

In our experiments we use random input vectors of size $\texttt{N} = 81,920$. Instead of comparing our predictions with the run times of the actually running parallel program (which would require a much more detailed PAMELA model) we use a counter-based measurement approach, in which we run a multithreaded version of PSRS that is instrumented with counters at the appropriate places in the program. The predicted execution time for $\texttt{P} \geq 2$ are obtained from profiling and running Fig. 3, respectively, while for $\texttt{P} = 1$ we choose straight selection sort given in Section 4.2 as the sequential program, instead of simply running PSRS for $\texttt{P} = 1$.

To observe the effect of memory communication on the execution time speedup we consider both $\texttt{t_m} = 0$ and `t_m = moments(10,100,2,9)`. In Fig. 4 the measured and predicted speedup are shown by solid and dashed lines, respectively, for $\texttt{P} \leq 128$. The speedup increases linearly with the number of processors for $\texttt{P} \leq 64$ and decreases for large `P` since the working of PSRS becomes inefficient for small array size. We measure a super linear speedup for $\texttt{t_m} = 0$ due to the choice of straight selection sort for the sequential version as well. For example the speedup is 10 for $\texttt{P} = 8$. Furthermore, assigning `t_m = moments(10,100,2,9)` degrades the speedup such that for $\texttt{P} = 2$ the speedup is less than 1. For both values of `t_m` the speedup prediction is approximately accurate while the error is mainly caused by the inaccuracy in profiling `c`. Note that the moments of `c`

```
%---------------------------------------------------------------------
% psrs.pam -- Parallel Sorting by Regular Sampling Model
%---------------------------------------------------------------------
numeric parameter N                              % array length
numeric parameter P                              % # processors
numeric t_f = moments(1,1,2,9)                   % computation time
numeric t_m = moments(10,100,2,9)                % communication time
numeric c1 = moments(5e-3, 2e+0, 4e+0, 2e+1)     % profiled for # runs
numeric c2 = moments(1e-2, 3e-2, 9e+0, 7e+1)     % profiled for # runs
numeric c3 = moments(4e-3, 5e+0, 2e+0, 6e+0)     % profiled for # runs
resource cpu(p) = fcfs(p,1)                      % the P CPUs
resource shmem = fcfs(P,P)                       % shared memory

process main = par (p=0, P-1) {                  % sort1:
                 sort(p*N/P,(p+1)*N/P-1,p,c1) ; % sort subarray
                 seq (i=0, P-1)                  % collect samples
                   move(p)
               } ;
               sort(0,P*P-1,p,c2) ;              % sort samples
               seq (i=0, P-2) {                  % choose pivots
                 move(p)
               } ;
               par (p=0, P-1) {                  % disjoin:
                 seq (i=0, P-1) {
                   seq (j=0, (N-N/P)/(P*P)-1) {
                     flop(p) ;
                     move(p)
                   }
                 }
               } ;
               par (p=0, P-1) {                  % sort2:
                 seq (i=0, P-1) {                % collect subarray
                   seq (j=0, (N-N/P)/(P*P)-1)
                     move(p)
                 } ;
                 sort(p*N/P,(p+1)*N/P-1,p,c3)   % sort subarray
               }

process sort(lb,ub,p,c) = seq (i=lb+1, ub) {     % sort
                            seq (j=lb, i-1) {
                              if (c)
                                flop(p)
                            } ;
                            move(p)              % swap
                          }

process flop(p) = use(cpu(p),t_f)                % computation
process move(p) = use(shmem,t_m)                 % memory communication
```

Fig. 3. Pamela model of PSRS

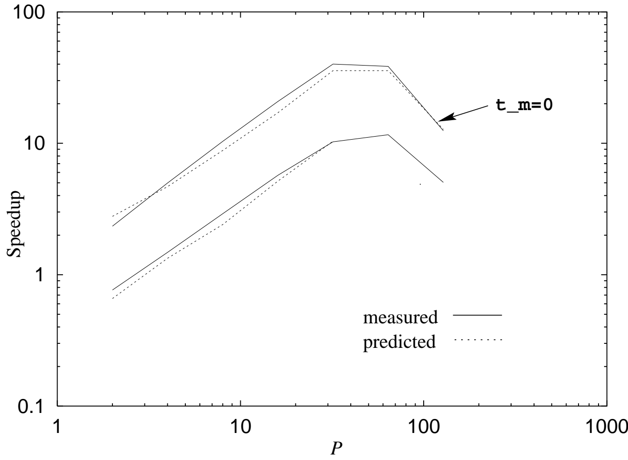

Fig. 4. Measured and predicted speedup.

are very sensitive to the program execution times since `c` is located in a loop nest with high invocation frequency.

The corresponding ε_1 of Fig. 4 is shown in Fig. 5. The highest error is $\varepsilon_1 =$ 18% at $\mathtt{P} = 16$ for $\mathtt{t_m} = 0$ and $\varepsilon_1 = 15\%$ at $\mathtt{P} = 2$ ε_1 decreases as `P` large since our measurements show that the correlation in `sort` is high for large array size. While the error is quite acceptable, the evaluation of the compiled version of above model only requires 1.7 CPU s.

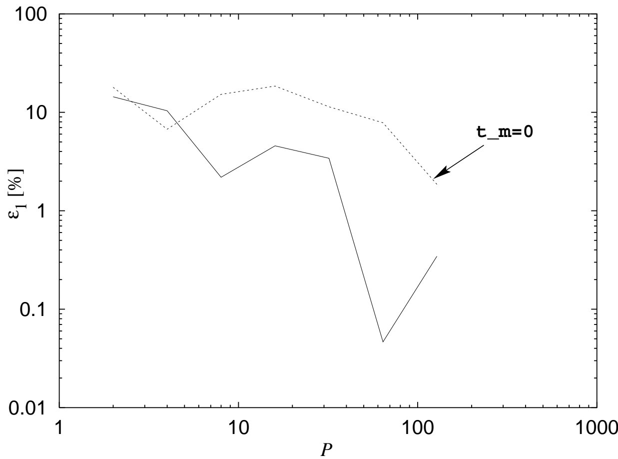

Fig. 5. ε_1 [%] for PSRS.

6 Conclusion

In this paper we have presented a compositional, analytic method to derive symbolic performance models of data-dependent parallel programs. Our approach approximates the first four statistical moments of program execution time in terms of the first four moments of the loop bounds, the branch conditions, and the execution time delays of the constituent tasks. The approach is intended to allow the use of stochastic workloads for modeling accuracy, while at the same time offering execution time expressions of $O(1)$ solution complexity.

Our moments method is implemented in terms of Pamela, offering modeling support by automatically generating the analytic performance models. Apart from a deterministic version, a first version of the compiler has been developed that completely supports stochastic workloads, automatically producing the first four moments of program execution time. Our experimental results show that the prediction error is in the percent range while the evaluation of the models only takes a few seconds. Future work is mainly directed to analyzing larger parallel programs.

Acknowledgements. The authors gratefully acknowledge the anonymous reviewers for their insightful comments.

References

1. V.S. Adve and M.K. Vernon, "The influence of random delays on parallel execution times," in *Proc. of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993, pp. 61–73.
2. T.S. Axelrod, "Effects of synchronization barriers on multiprocessor performance," *Parallel Computing*, vol. 3, May 1986, pp. 129–140.
3. H. Gautama, "Static branch performance prediction using alternating renewal processes," Tech. Rep. 1-68340-44(2001)04, Delft University of Technology, Delft, The Netherlands, Mar. 2001.
4. H. Gautama and A.J.C. van Gemund, "Static performance prediction of data-dependent programs," in *ACM Proc. on The Second Int. Workshop on Software and Performance (WOSP 2000)*, Sept. 2000, pp. 216–226. Ottawa, Canada.
5. H. Gautama and A.J.C. van Gemund, "Low-cost performance prediction of data-dependent data parallel programs," in *Proc. of the 9th Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2001)*, IEEE Computer Society Press, Aug. 2001, pp. 173–182. Cincinnati, Ohio.
6. H. Gautama and A.J.C. van Gemund, "Performance prediction of data-dependent task parallel programs," in *Proc. of the 7th Int. Conference on Parallel Processing (EuroPar 2001)*, Aug. 2001, pp. 106–116. Manchester, United Kingdom.
7. E. Gelenbe, E. Montagne, R. Suros and C.M. Woodside, "Performance of block-structured parallel programs," in *Parallel Algorithms and Architectures* (M. Cosnard *et al.*, eds.), Amsterdam: North-Holland, 1986, pp. 127–138.
8. A.J.C. van Gemund, "Performance prediction of parallel processing systems: The Pamela methodology," in *Proc. 7th ACM Int. Conf. on Supercomputing*, Tokyo, July 1993, pp. 318–327.

9. A.J.C. van Gemund, "Compile-time performance prediction of parallel systems," in *Proc. Computer Performance Evaluation: Modelling Techniques and Tools (Tools'95), LNCS 977*, Heidelberg, Sept. 1995, pp. 299–313.
10. A.J.C. van Gemund, *Performance Modeling of Parallel Systems.* Delft University Press, 1996.
11. E.J. Gumbel, "Statistical theory of extreme values (main results)," in *Contributions to Order Statistics* (A.E. Sarhan and B.G. Greenberg, eds.), New York: John Wiley & Sons, 1962, pp. 56–93.
12. E.J. Dudewicz J.S. Ramberg, P.R. Tadikamalla and F.M. Mykytka, "A probability distribution and its uses in fitting data," *Technometrics*, vol. 21, May 1979, pp. 201–214.
13. W. Kreutzer, *System simulation, programming styles and languages.* Addison-Wesley, 1986.
14. C.P. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," *IEEE Trans. on Software Engineering*, vol. 11, Oct. 1985, pp. 1001–1016.
15. B.P. Lester, "A system for the speedup of parallel programs," in *Proc. of the 1986 Int. Conference on Parallel Processing*, IEEE, Aug. 1986, pp. 145–152. Maharishi Int. U, Fairfield, Iowa.
16. J. Lüthi, S. Majumdar, G. Kotsis and G. Haring, "Performance bounds for distributed systems with workload variabilities and uncertainties," *Parallel Computing*, vol. 22, Feb. 1997, pp. 1789–1806.
17. S. Madala and J.B. Sinclair, "Performance of synchronous parallel algorithms with regular structures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, Jan. 1991, pp. 105–116.
18. Pamela Project Web Site, `http://ce.et.tudelft.nl/~hasyim/Pamela`.
19. M. Reiser and S.S. Lavenberg, "Mean value analysis of closed multichain queueing networks," *Journal of the ACM*, vol. 27, Apr. 1980, pp. 313–322.
20. J.T. Robinson, "Some analysis techniques for asynchronous multiprocessor algorithms," *IEEE Trans. on Software Engineering*, vol. 5, Jan. 1979, pp. 24–31.
21. R.A. Sahner and K.S. Trivedi, "Performance and reliability analysis using directed acyclic graphs," *IEEE Trans. on Software Engineering*, vol. 13, Oct. 1987, pp. 1105–1114.
22. V. Sarkar, "Determining average program execution times and their variance," in *Proc. of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, 1989, pp. 298–312.
23. J.M. Schopf and F. Berman, "Performance prediction in production environments," in *Proc. of the 1st Merged Int. Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, Los Alamitos, IEEE Computer Society, Mar. 30–Apr. 3 1998, pp. 647–653.
24. J.M. Schopf and F. Berman, "Using stochastic information to predict application behavior on contended resources," *IJFCS: Int. Journal of Foundations of Computer Science*, vol. 12, 2001.
25. H. Shi and J. Schaeffer, "Parallel sorting by regular sampling," *Journal of Parallel and Distributed Computing*, vol. 14, no. 4, 1992, pp. 361–372.
26. F. Sötz, "A method for performance prediction of parallel programs," in *Proc. CONPAR 90-VAPP IV (LNCS 457)* (H. Burkhart, ed.), Springer-Verlag, 1990, pp. 98–107.