Symbolic Performance Modeling of Parallel Systems

Arjan J.C. van Gemund, *Member*, *IEEE Computer Society*

Abstract—Performance prediction is an important engineering tool that provides valuable feedback on design choices in program synthesis and machine architecture development. We present an analytic performance modeling approach aimed to minimize prediction cost, while providing a prediction accuracy that is sufficient to enable major code and data mapping decisions. Our approach is based on a performance simulation language called PAMELA. Apart from simulation, PAMELA features a symbolic analysis technique that enables PAMELA models to be compiled into symbolic performance models that trade prediction accuracy for the lowest possible solution cost. We demonstrate our approach through a large number of theoretical and practical modeling case studies, including six parallel programs and two distributed-memory machines. The average prediction error of our approach is less than 10 percent, while the average worst-case error is limited to 50 percent. It is shown that this accuracy is sufficient to correctly select the best coding or partitioning strategy. For programs expressed in a high-level, structured programming model, such as data-parallel programs, symbolic performance modeling can be entirely automated. We report on experiments with a PAMELA model generator built within a data-parallel compiler for distributed-memory machines. Our results show that with negligible program annotation, symbolic performance models are automatically compiled in seconds, while their solution cost is in the order of milliseconds.

Index Terms—Performance prediction, parallel processing, analytic performance modeling.

1 Introduction

In high-performance computing, application performance is very sensitive to program features such as code and data partitioning, and machine computation and communication parameters. Performance prediction is an important engineering tool that provides timely feedback on design choices in program synthesis as well as in machine architecture development. Apart from prediction accuracy, prediction *cost* largely determines the utility of performance prediction tools. To enable a user to quickly find his/her way in the multidimensional design space, performance models often need to be used interactively. This implies that, e.g., parameterized plots be generated in seconds, despite the staggering dimensions in high-performance computing. To enable compile-time synthesis, an even more drastic cost reduction is required in view of the huge number of model evaluations required in automatic optimization. In this paper, we study to what extent prediction cost can be *minimized* while retaining *sufficient* prediction accuracy to allow a correct ranking of different design choices during the first stages of parallel algorithm and/or architecture development.

Prediction cost breaks down into *modeling cost*, associated with deriving the model, and *solution cost*, associated with evaluating the model for some parameter setting. Xu et al. rate the effectiveness of a performance prediction method in terms of the so called *prediction ratio* [37]. Loosely speaking, the prediction ratio is defined as the actual application execution time divided by the sum of the modeling cost and the solution cost. In their definition, the modeling cost equals the measurement time associated with determining program or machine-specific information needed in the performance model (program profiling, machine benchmarking). We will extend the definition of modeling cost to include the *construction* of the model itself. Depending on the particular performance modeling approach, model construction cost can be as complex as developing the program and/or machine itself. Similarly, solution cost is a critical factor. When a performance model is parameterized, solution cost may well dominate as modeling cost is typically amortized over many evaluations.

Performance prediction approaches take many shapes, the choice of underlying modeling formalism depending on the desired tradeoff between prediction cost and accuracy. Although potentially accurate, modeling formalisms such as stochastic Petri nets [3] or process algebras [16] are not attractive due to the exponential solution cost. Although approaches based on a combination of directed acyclic task graphs (DAGs) and queuing networks [1], [20], [23] pair comparably high modeling power with a high efficiency, the polynomial time complexity of the solution process still entails considerable cost for very large problem sizes. In *symbolic* approaches, the application is transformed into an explicit, algebraic performance expression. In contrast to the above *numeric* approaches, symbolic prediction techniques offer even lower solution cost that is less dependent on problem size. Typically, parallel programs and machines have some degree of regularity (replication). Being reflected in the symbolic performance model, this regularity is exploited through symbolic simplification, reducing solution cost by many orders of magnitude. However, this advantage comes at a price. While manual approaches, such as BSP [33]

• *The author is with the Department of Information Technology and Systems, Delft University of Technology, PO Box 5031, NL-2600 GA Delft, The Netherlands. E-mail: a.j.c.vangemund@its.tudelft.nl.*

Manuscript received 2 Sept. 1999; revised 30 Nov. 2001; accepted 17 May 2002.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 110568.

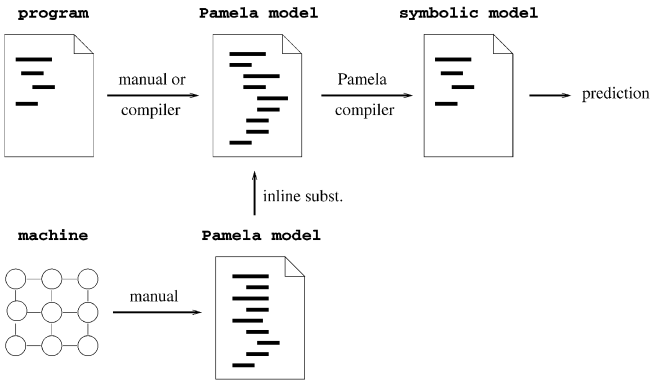

Fig. 1. PAMELA performance modeling process.

and LogP [9], pair accuracy with low solution cost, modeling cost is still significant due to the labor-intensive and error-prone derivation effort. Alternatively, symbolic prediction techniques based on stochastic SP (series-parallel) DAGs [29] and, most notably, deterministic SP DAGs offer a mechanized, compile-time derivation scheme (e.g., [5], [7], [11], [24], [35]). However, unlike the DAG/queuing network approaches mentioned earlier, pure DAG approaches do not account for mutual exclusion synchronization. Consequently, unacceptable prediction errors arise when performance is largely dominated by contention for resources such as locks, servers, processors, network links, or memories.

In this paper, we show that a symbolic performance modeling approach *is* possible that minimizes *both* modeling cost and solution cost, while providing a prediction accuracy that is *acceptable* during the first stages of parallel program design. Our approach is based on the use of an intermediate modeling formalism called PAMELA (PerformAnce ModEling LAnguage [12]). PAMELA is a compositional, process oriented performance simulation language. Apart from simulation, PAMELA features a performance analysis technique that enables PAMELA models to be *compiled* into symbolic performance models that trade prediction accuracy for the lowest possible solution complexity. In our approach, we combine the low cost critical path analysis of deterministic SP DAGs with an approximate analysis of the effects of mutual exclusion. The restriction to deterministic workloads, essential to obtain minimum solution cost, does not degrade prediction accuracy. Although at the fine grain task level variance can be considerable, at the highly aggregate level at which the effects of parallel composition is computed, variance is much more limited [2], [22]. While the mutual exclusion analysis does not compromise the low solution cost associated with SP DAG analysis, it fundamentally improves prediction accuracy to a level that is acceptable during the first stages of parallel system design. Experimental results on six parallel programs and two distributed memory architectures show an average prediction error of less than 10 percent, while the average worst-case error is limited to 50 percent. It is shown that this accuracy is more than sufficient for a clear assessment of application scalability and identification of the best coding or partitioning strategy.

Our modeling approach is illustrated in Fig. 1. For both program and machine, a separate PAMELA model is constructed (manually or generated). After algebraic substitution of the machine model in the program model, the resulting model is translated by a PAMELA compiler into the symbolic performance model. Due to the PAMELA compiler's symbolic simplification capabilities, a dramatic solution cost reduction of many orders of magnitude is achieved when compared to simulation.

As the PAMELA simulation formalism is intuitively close to the computational system under study, modeling cost is significantly reduced compared to manual symbolic approaches that do not provide tool support. Moreover, for programs that are expressed in terms of a high-level programming model such as the data-parallel model, the program modeling process can be entirely mechanized, requiring relatively few program annotations. Consequently, the entire performance modeling effort effectively reduces to *compiling* parallel programs to symbolic performance models [13], yielding an extremely high prediction ratio. We report on experiments with a PAMELA model generator that has been developed as part of a compiler that translates data-parallel Java programs for distributed-memory (MPI) machines [15]. Results for four data-parallel programs (MATMUL, ADI, GAUSS, and PSRS) demonstrate that with minimum program annotation and machine modeling effort, the symbolic model is generated in a matter of seconds, while the solution cost is in the order of milliseconds.

The paper is organized as follows: In the next section, we present the PAMELA language, the symbolic compilation process, and its inherent prediction accuracy. In Section 3, we demonstrate the cost/performance ratio of the PAMELA approach through a number of theoretical and practical modeling case studies. In Section 4, we describe the results obtained with the automatic PAMELA generator. In Section 5, we compare our approach with related work in symbolic performance prediction. Finally, in Section 6, we summarize our contribution.

2 Modeling Formalism

In this section, we describe the PAMELA language, the symbolic analysis, and we discuss the prediction accuracy. The language and analysis is implemented in terms of a mathematical tool [26] that compiles process expressions into simplified execution time expressions.

2.1 Language

PAMELA is a process-oriented performance simulation language designed to capture concurrency and timing behavior of parallel systems. Original data computations are only modeled in terms of their workload, similar to, e.g., Petri nets and queuing networks. Based on a process algebra, PAMELA uses the equation syntax and substitution semantics found in ordinary algebra. A PAMELA model of a program is written as a set of process equations. The left-hand side of the root equation is usually denoted L.

Work (computation, communication) is described by the elementary **use** process. The construct **use**(s, τ) exclusively acquires service from a server s for τ time units (excluding possible queuing delay). In the sequel we will generally refer to servers as *resources*. A resource s has a multiplicity, denoted $|s|$, that may be larger than one. As in traditional queuing

networks, resource queues have infinite capacity. The mutual exclusion synchronization offered by the **use** construct is defined according to a work conserving scheduling discipline with nondeterministic conflict arbitration. The precise discipline (such as FCFS or PS) is not relevant to the symbolic timing analysis described in the next section. As in queuing networks, it is convenient to define an infinite server ρ, where $|\rho| = \infty$. Instead of **use**(ρ, τ), we will simply write **delay**(τ).

Program and machine models are composed from **use** (or **delay**) processes using the following three process composition mechanisms:

- Sequential composition, expressed through the binary operator ";" and the n-ary replication operator **seq**, defined as **seq**$(i = a, b)\ L_i \equiv L_a\ ;\ \ldots ;\ L_b$.
- Parallel composition, expressed through the binary operator "$\|$ " and the n-ary replication operator **par**, defined as **par**$(i = a, b)\ L_i \equiv L_a \parallel \ldots \parallel L_b$. Parallel composition includes barrier synchronization to terminate the parallel process.
- Conditional composition, expressed through **if** $(c)\ L$, where c is a Boolean condition. An optional **else** construct is provided.

The implicit condition synchronization of the parallel and sequential composition constructs enables the expression of models that are similar to SP DAGs. However, the modeling power of PAMELA extends static DAGs by virtue of the mutual exclusion from the **use** construct, which also provides the nondeterminism required to model dynamically scheduled systems. This property is discussed in Section 3.

We conclude this section by modeling the Machine Repair Model (MRM) [21] which offers a typical demonstration of the modeling approach in PAMELA. In an MRM P clients either spend τ_l on local processing, or request service time τ_s from a single FCFS server s, for a total cycle count of N iterations (N may be infinite, i.e., symbolic). The PAMELA model of the MRM is given by the following process equation:

```
L = par (p = 1, P)
        seq (i = 1, N) {
            delay(τ_l) ;
            use(s, τ_s)
        }
```

For reading convenience, the process expression is displayed in program format including the usual indentation. Note that instead of modeling the server as a process, a passive *resource* is used. This is typical for PAMELA's top-down modeling paradigm where all inherent problem parallelism is expressed, possibly constrained by a limited number of resources such as locks, servers, processors, communication links, memories, etc. The modeling paradigm is further discussed in Section 3.

2.2 Analysis

As PAMELA models represent a time simulation of concurrent system execution, each PAMELA expression L has an associated execution time $T(L)$. $T(L)$ is typically computed by simulation,[1] which is a high-cost solution technique. The low-cost, symbolic transformation procedure we describe in this paper, yields a lower bound $T^l(L)$, trading accuracy for a drastic solution cost reduction. The approach is based on an approximation of the time delay due to mutual exclusion (contention), integrated within a condition synchronization delay analysis (critical path analysis of the SP DAG).

Conditional composition is transferred into the time domain according to the transformation

$$T(\textbf{if}\ (c)\ L) = \textbf{if}\ (c)\ T(L). \tag{1}$$

The time domain expression is subsequently reduced using information on the truth value (probability) of c. An example is given in Section 3.2. In the following, we assume all conditional process compositions have been processed.

Let L denote a PAMELA model comprising some parallel and/or sequential composition of **use** or **delay** tasks. One estimate of $T^l(L)$ is given by analyzing the effects of condition synchronization. Let $\varphi(L)$ denote the critical path estimate. In terms of the binary composition operators ";" and "$\|$," the following recursion holds

$$\varphi(L) = \begin{cases} \varphi(L_1) + \varphi(L_2), & L = L_1\ ;\ L_2; \\ \max(\varphi(L_1), \varphi(L_2)), & L = L_1 \parallel L_2; \\ \tau, & L = \textbf{delay}(\tau); \\ \tau, & L = \textbf{use}(r, \tau). \end{cases} \tag{2}$$

As mutual exclusion is ignored it follows $\varphi(L) \leq T(L)$.

Another estimate of $T(L)$ is given by analyzing the effects of mutual exclusion. Let $\underline{\delta}(L) = (\delta_1, \ldots, \delta_M)$ denote the total service demand vector of L, where M is the total number of resources involved and δ_m denotes the service demand on resource r_m. We will write $\delta_m(L)$ to denote the mth element of $\underline{\delta}(L)$. It (recursively) holds

$$\underline{\delta}(L) = \begin{cases} \underline{\delta}(L_1) + \underline{\delta}(L_2), & L = L_1\ ;\ L_2; \\ \underline{\delta}(L_1) + \underline{\delta}(L_2), & L = L_1 \parallel L_2; \\ \tau \underline{e}^m, & L = \textbf{use}(r_m, \tau), \end{cases} \tag{3}$$

where $\underline{e}^m = (0, \ldots, 0, 1, 0, \ldots, 0)$ is the M-dimensional unit vector in the m direction, and addition and multiplication are defined element-wise. Let $\omega(L)$ denote the lower bound on $T(L)$ due to mutual exclusion. Then,

$$\omega(L) = \max_{m=1\ldots M} \frac{\delta_m(L)}{|r_m|}. \tag{4}$$

As condition synchronization is ignored it follows $\omega(L) \leq T(L)$.

Combining the lower bounds due to contention and critical path analysis it follows that a lower bound on $T(L)$ is given by the following recursion

$$T^l(L) = \begin{cases} T^l(L_1) + T^l(L_2), & L = L_1\ ;\ L_2; \\ \max(T^l(L_1), T^l(L_2), \omega(L)), & L = L_1 \parallel L_2; \\ \tau, & L = \textbf{delay}(\tau); \\ \tau, & L = \textbf{use}(r, \tau). \end{cases} \tag{5}$$

The integration of contention and critical path analysis is implemented in terms of the second transformation rule that applies to parallel composition. Traditional compile-time analysis typically disregards ω while queuing analysis typically disregards φ.

1. PAMELA models are also compiled to simulators [14]. Although simulation data are included in some of our results, this feature is outside the focus of this paper.

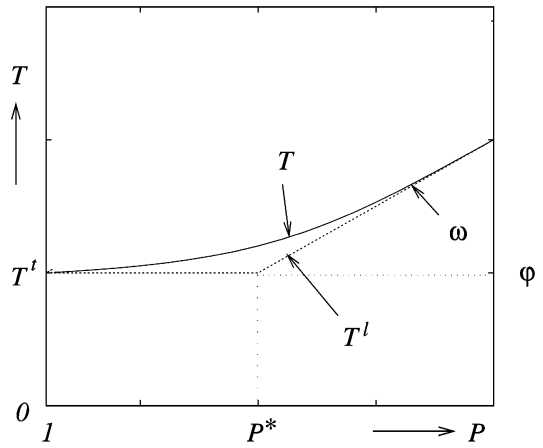

Fig. 2. T^l vs. T for the MRM.

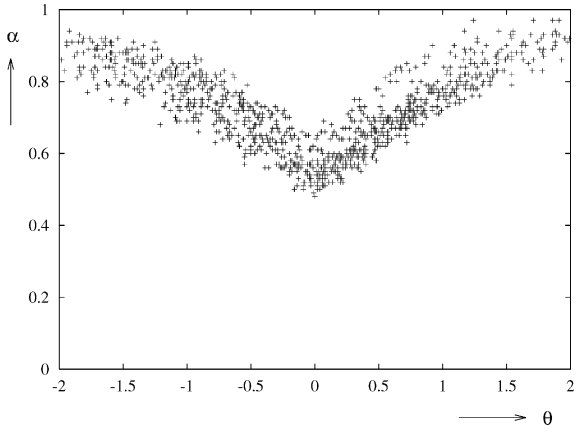

Fig. 3. T^l/T for 1,000 random models.

Note that the time complexity of the T^l expression generation equals that of traditional critical path analysis. While the above process generates symbolic expressions whose numeric evaluation complexity still compares to that of simulation, the typical regularity within the expressions enables the PAMELA compiler to simplify T^l, reducing evaluation cost by many orders of magnitude. This important property of symbolic techniques is demonstrated throughout the paper where $O(1)$ time complexity is achieved in most examples and experiments.

We conclude this section with a symbolic analysis of the MRM example presented earlier. Mechanically applying the above transformations yields

$$T^l(L) = \max\left[\max_{p=1\ldots P}\sum_{i=1}^{N}(\tau_l+\tau_s), \max\left(\sum_{p=1}^{P}\sum_{i=1}^{N}\tau_s\ \underline{e}^0\right)\right].$$

This expression is simplified by the compiler to $T^l = N\max(P\tau_s, \tau_l+\tau_s)$ which has $O(1)$ solution complexity. Unlike traditional compile-time analysis, T^l accounts for the additional queuing delay when s is saturated.

2.3 Accuracy

As mentioned earlier, our analysis approach implies a sacrifice in prediction accuracy. In this section, we study the tightness of the lower bound T^l compared to the simulation result T. Although the prediction error of T^l is theoretically unlimited, we will show that the *average worst-case* error is limited to 50 percent, i.e., $E[T^l/T] \geq 0.5$, while the average error is typically much less.

To introduce our study, Fig. 2 compares T^l with T for the MRM example. The figure shows that T^l essentially forms the asymptotes of T, with a deviation of less than 50 percent occurring at the saturation point $P^* = (\tau_s+\tau_l)/\tau_s$. Notice the $O(P)$ error of traditional compile-time analysis (φ) for $P > P^*$. From (5) and the above example it follows that T^l approaches T either when $\varphi \gg \omega$ (critical path dominates) or when $\varphi \ll \omega$ (queuing dominates). To study the prediction error, we introduce an independent parameter θ, coined *contention index*, that expresses the relative degree of contention within a model according to $\theta = \log(\omega/\varphi)$.

Fig. 3 shows the ratio $\alpha = T^l/T$ for 1,000 random PAMELA models with θ values ranging from $-2 < \theta < 2$. Each model comprises $N = 100$ coarse-grain tasks that execute a large sequence of random accesses to M resources, where M varies from $2, \ldots, 100$ across the 1,000 models. The access distribution is uniform, as load balance produces the worst accuracy scenarios. The results clearly show a high degree of correlation between α and θ, where the deviation from unity is maximum for models that exhibit $\theta = 0$, in which case it holds $T^l = 0.5T$ on average. The predictive value of θ is quite strong, especially considering the fact that two models with comparable θ values are usually entirely different. Our measurements show that N is large enough to be representative for much larger models. Additional measurements indicate that the *minimum* value of α at $\theta = 0$, denoted α^*, highly correlates with M, while exhibiting an asymptote for large M given by $\alpha^* \approx 0.5$ [14]. This result agrees with the result from asymptotic bounding analysis of queuing systems [38]. For an M server, load balanced system with total service demand D, Mean Value Analysis [21] yields a response time given by the recursion

$$R_P = D + \frac{D}{M}\frac{R_{P-1}}{R_{P-1}+Z}(P-1),$$

where P denotes the number of jobs. Let $C(P) = R(P) + Z$ denote the mean cycle time. The largest deviation of $C(P)$ from the lower bound $C^l = \max(D+Z, (D/M)P)$ occurs for $P^* = (D+Z)M/D$. Since for finite P the slope of $R(P)$ is less than D/M, it follows that $C(P) < D + (D/M)P + Z$. Consequently $C(P^*) < 2C^l$ which agrees with our average 50 percent worst-case prediction error mentioned above.

3 Modeling Examples

In this section, we demonstrate the particular PAMELA approach to modeling parallel programs and machines through a number of modeling examples. In order to enable our static, symbolic analysis PAMELA is restricted to an *orthogonal* synchronization model, separating condition synchronization constructs (**seq**, **par**), from mutual exclusion (**use**). Despite the restricted modeling power, a wide range of parallel computations can be adequately modeled by focusing on the *algorithm* (problem) level, instead of the implementation (machine) level. At the algorithm level inherent problem parallelism is expressed in terms of condition synchronization (DAG), while the mapping of the parallel algorithm on a limited number of (virtual) machine resources can be separately expressed in terms of mutual exclusion

(including the nondeterminism required to adequately model dynamic scheduling). At the implementation level, however, such an orthogonal synchronization structure is often no longer apparent. A typical example is a dynamic message-passing program where static and dynamic synchronizations have become intertwined, prohibiting our static analysis scheme. A compelling demonstration of the potential semantic gap between algorithm and implementation is the MPI implementation of the APSP algorithm described in Section 3.8. At algorithm level, the application is simply modeled in terms of an SP DAG, combined with mutually exclusive server access (compare MRM). In contrast, directly modeling the MPI code, featuring a dynamic work load scheduling process, would require a modeling power equivalent to that of, e.g., CSP. Consequently, automatically "reverse-engineering" the implementation back into the algorithm level model is typically beyond machine capability. We coin our top-down modeling approach *contention modeling* to express the fact that the various scheduling constraints on problem parallelism, induced by software and hardware resources, are modeled in terms of contention (mutual exclusion). In the following, we present some typical modeling case studies to demonstrate our modeling approach, the ease with which symbolic performance models are derived, the low solution complexity that is achievable, and the limited prediction error that is incurred.

3.1 Pipelining

Consider the pipelined processing of N data sets involving an M unit pipeline (e.g., vector unit, packet-switched communication pipeline, software pipeline). Instead of modeling the implementation in terms of a classic, static non-SP synchronization pattern (diamond DAG), parallelism is expressed at the algorithm level in terms of parallel data set processes obtaining service from each stage. Consequently, the model is expressed as an *SP* DAG of *contending* tasks

$$\begin{aligned} L = \ & \mathbf{par}\ (i = 1, N) \\ & \quad \mathbf{seq}\ (m = 1, M) \\ & \quad\quad \mathbf{use}(u_m, \tau_m), \end{aligned}$$

where u_m models stage m, and τ_m denotes the associated processing time. Symbolic analysis yields

$$T^l = \max\left(\sum_{m=1}^{M} \tau_m, N \max_{m=1}^{M} \tau_m\right).$$

The prediction error compared to the exact (manual) solution $T = M\tau_m + (N-1)\max_{m=1}^{M} \tau_m$ is negligible for cases where either φ dominates (latency term, $N \ll M$) or where ω dominates (bandwidth term, $N \gg M$). The maximum error (50 percent) occurs for load-balanced systems (i.e., $\tau_m = \tau$) for the case $N = M$, where T^l reduces to $T^l = N\tau$ while $T = (2N-1)\tau$. Note that pipeline synchronization cannot be modeled in great detail. As mutual exclusion in PAMELA assumes infinite queues, all data is implicitly modeled to queue at the slowest stage whereas in reality data would occupy each stage. This lack of modeling power, however, does not affect the accuracy of the overall performance result T^l.

3.2 Branching

Consider the sequential program

```
for (i = 1:N)
    if (x[i] != 0)
        x[i] = x[i] * a;
```

that scales the vector x by a constant a. Let the machine model be given by the single equation $mult = \mathbf{use}(cpu, \tau)$ (we ignore all other instructions for simplicity). Then, the algorithm is modeled by

$$L = \mathbf{seq}\ (i = 1, N)\ \mathbf{if}\ (nz(i))\ mult,$$

where nz models the nonzero test. Substituting the machine model in the program model and subsequently applying the symbolic analysis yields

$$T^l = \sum_{i=1}^{N} \mathbf{if}\ (nz(i))\ \tau.$$

Let $nz(i)$ be modeled by a stochastic binary process $B_p \in \{0, 1\}$, where the truth probability parameter p is determined by, e.g., vector density profiling. Then, T^l reduces to $T^l = p\tau$ which has $O(1)$ complexity. Note that memory hierarchy is also modeled using conditional composition.

3.3 Vectorization

Consider the vector scaling

```
forall (i = 1:N)
    x[i] = x[i] * a;
```

Let the algorithm be modeled as

$$L = \mathbf{par}\ (i = 1, N)\ mult(i)$$

where $mult(i)$ models the multiplication process (we ignore all other instructions for simplicity). We consider three different machine mappings. A *sequential* implementation is expressed by the machine model $mult(i) = \mathbf{use}(s, \tau_s)$, where s models a scalar multiplier unit. It follows $T^l = N\tau_s$. Alternatively, *vectorization* implies the use of a pipelined unit modeled by $mult(i) = \mathbf{seq}\ (j = 1, S)\ \mathbf{use}(v_j, \tau_v)$ where v_j models each of the S vector stages. It follows $T^l = \max(S\tau_v, N\tau_v)$ which implies speedup provided $\tau_v < \tau_s$ and N is sufficiently large. Finally, a *fully parallel* implementation is modeled by $mult(i) = \mathbf{use}(s_i, \tau_s)$, i.e., N scalar units, which yields $T^l = \tau_s$. Thus, evaluating machine mappings is a matter of substituting the appropriate machine model while the algorithm model remains unchanged.

3.4 Vector Chaining

Consider mapping

```
forall (i = 1:N) {
    a[i] = d[i] * e[i];
    c[i] = a[i] * b[i];
}
```

to a machine with vector units $mult_1$ and $mult_2$ as defined earlier. Sequentializing the statements is modeled by

$$\begin{aligned} L = \ & \{\ \mathbf{par}\ (i = 1, N)\ mult_1(i)\ \}\ ; \\ & \{\ \mathbf{par}\ (i = 1, N)\ mult_2(i)\ \}, \end{aligned}$$

which compiles to $T^l = 2\max(S\tau_v, N\tau_v)$. Alternatively, chaining both vector operations is modeled by

$$L = \textbf{par}\ (i = 1, N)\ \{\ mult_1(i)\ ;\ mult_2(i)\ \},$$

where $mult_1(i)\ ;\ mult_2(i)$ *literally* models the hardware concatenation. This model compiles to $T^l = \max(2S\tau_v, N\tau_v)$ which predicts double bandwidth (for large N). The example illustrates how *structure* (PAMELA) is compiled into *behavior* (symbolic model).

3.5 Memory Bank Contention

Consider a memory bank system comprising M interleaved memory banks with access time τ_m. Let a memory vector port generate the address sequence $f(i), i = 1, \ldots, N$. The vector access is modeled by

$$\begin{aligned} L = \textbf{par}\ \ & (i = 1, N)\ \{ \\ & \textbf{use}(port, \tau_c)\ ; \\ & \textbf{use}(m_{f(i) \bmod M}, \tau_m) \\ & \} \end{aligned}$$

The port is modeled as a passive resource, serializing the N parallel requests at the port request rate $1/\tau_c$. It follows

$$\begin{aligned} T^l = & \max(\tau_c + \tau_m, \max(N\tau_c, \\ & \max_{m=1}^{M} \sum_{i=1}^{N} \textbf{if}\ (f(i) \bmod M = m)\ \tau_m)). \end{aligned}$$

When M is sufficiently large, memory contention does not occur and T^l reduces to the familiar *memory pipeline* characterized by startup time $\tau_c + \tau_m$ and bandwidth $1/\tau_c$ [18]. For small M memory bank contention depends on f. Let $f(i) = Si$, where S denotes the access stride. Then, T^l reduces to

$$T^l = \max(\tau_c + \tau_m, \max(N\tau_c, N\gcd(M, S)\tau_m/M)).$$

As $\tau_c < \tau_m$ it follows $T^l = N\gcd(M, S)\tau_m/M$ which implies an effective memory bandwidth of $M/(\gcd(M, S)\tau_m)$. This mechanically compiled result equals the effective memory bandwidth derived in, e.g., [25].

3.6 Data Partitioning

Consider the vertical relaxation phase

```
for (i = 1:N-2)
    forall (j = 0:N-1)
        update(i,j);
```

in ADI where `update` refers to the scalar update of matrix element $a_{i,j}$ as a function of $a_{i-1,j}$ and $a_{i+1,j}$. Let $a_{i,j}$ be mapped on processor $\mu(i, j)$. Let `update (i,j)` also be performed by processor $\mu(i, j)$ (owner-computes rule). The algorithm is modeled by

$$\begin{aligned} L = \textbf{seq}\ \ & (i = 1, N - 2) \\ & \textbf{par}\ (j = 0, N - 1) \\ & update(i, j) \end{aligned}$$

where $update(i, j) = \textbf{use}(cpu_{\mu(i,j)}, \tau_u)$ models `update (i,j)`. We consider the choice between a block partitioning on either the j or the i axis. Without loss of generality, we assume $P|N$ and $B = N/P$. The j axis partitioning implies $\mu(i, j) = \lfloor j/B \rfloor$. Hence, $update(i, j) = \textbf{use}(cpu_{\lfloor j/B \rfloor}, \tau_u)$ which yields $T^l = (N - 2)N/P\tau_u$ (linear speedup). In contrast, the i axis partitioning implies $\mu(i, j) = \lfloor i/B \rfloor$ which yields $T^l = (N - 2)N\tau_u$ (no speedup). The "processor contention" modeling approach is also used in the PAMELA generator described in Section 4. The corresponding measurement results of the vertical ADI phase are presented in Section 4.3.

3.7 Network Contention

In this section, we model a synthetic benchmark program running on a 4×4 T800 transputer mesh. The benchmark is characteristic for irregular finite element (FEM) computations and executes a coarse grain FEM computation graph (SP DAG) in a macro data flow style. The $N = 100$ tasks are mapped to the $P = 16$ processors using multithreading. Although out of service, the T800 mesh makes an interesting target machine. The communication infrastructure is still implemented in software, which necessitates accurate network contention modeling.

A task (thread) running on processor p is modeled by a $\textbf{use}(cpu_p, \ldots)$ process. The communication model is more detailed. The message-passing implementation is based on a "virtual link" service that provides a dedicated logical channel between a sender and receiver thread [27]. Each data transfer is modeled as a pipeline, each process modeling the propagation of an individual 120 bytes packet (compare Section 3.1). Let $move(s, r, l)$ denote the PAMELA model of an l bytes nonblocking, asynchronous data transfer between sending processor index s and receiving processor r. Let $n_k = s \ldots r$ denote the index of the K nodes involved in the pipeline route (dimension-order routing). Then, the communication model is given by [μs]

$$\begin{aligned} move(s, r, l) = \textbf{par}\ & (i = 1, \lceil l/120 \rceil)\ \{ \\ & \textbf{seq} = (k = 2, K - 1)\ \{ \\ & \quad \textbf{use}(f_{n_k}, 181)\ \| \\ & \quad \textbf{use}(x_{n_k}, 108) \\ & \}\ ; \\ & \textbf{use}(x_r, 108) \\ & \} \end{aligned}$$

where x_n denotes node link transfer (DMA) services, and f_n denotes node forwarding services. The model accurately captures the effective bandwidth degradation when multiple virtual links are simultaneously active. Table 1 shows some results for 10^6 byte concurrent data transfers for the subtopology $0 - 1 - 2$. Each tuple (s, r) corresponds to a transfer from node s to r. An exponent k denotes k concurrent (s, r) transfers. Apart from T^l, the measured value T^m, and the PAMELA simulation result T, the traditional static prediction T^t is listed to illustrate the prediction errors of latency/bandwidth models, which do not model contention.

The results show that the error of T^l is within 10 percent. The difference between T^m and T in the last row is caused by the fact that the PAMELA's nondeterministic conflict arbitration model sometimes yields better results that the actual arbitration performed in the router.

Table 2 summarizes the measurement results for the benchmark program for 14 random SP graphs $G_1 \ldots G_{14}$, selected to cover the worst-case θ region. The computation workload per task (thread) is uniformly distributed over $[0.06, 6.1]$ s, while the message size is uniformly distributed

TABLE 1
Results for Some 10^6 Byte Concurrent Transfers [s]

transfers	T^m	T	T^l	T^t
$(0,1)$	0.9	0.9	0.9	0.9
$(0,2)$	1.5	1.5	1.5	1.5
$(0,1)^2$	1.8	1.8	1.8	0.9
$(0,2)^2$	3.0	3.0	3.0	1.5

transfers	T^m	T	T^l	T^t
$(0,1) \parallel (0,2)$	1.8	1.8	1.8	1.5
$(0,1)^2 \parallel (0,2)$	2.7	2.7	2.7	1.5
$(0,1) \parallel (0,2)^2$	3.3	3.3	3.0	1.5
$(0,1)^3 \parallel (0,2)^3$	6.0	5.4	5.4	1.5

TABLE 2
Measurements vs. Predictions [s]

G	T^m	T	θ	α	T^t
G_1	118.7	114.7	1.08	0.73	25.9
G_2	93.9	92.5	0.85	0.62	21.2
G_3	95.6	92.8	0.76	0.68	25.8
G_4	94.1	87.4	0.37	0.70	31.5
G_5	73.4	70.9	0.27	0.66	30.3
G_6	105.8	103.9	-0.19	0.55	58.3
G_7	98.4	87.0	-0.28	0.60	47.4

G	T^m	T	θ	α	T^t
G_8	89.2	87.1	-0.35	0.68	52.5
G_9	87.6	84.4	-0.73	0.78	65.7
G_{10}	109.5	106.4	-0.91	0.75	79.5
G_{11}	141.2	138.4	-1.23	0.80	107.6
G_{12}	149.5	144.8	-1.51	0.87	125.0
G_{13}	165.9	163.2	-1.62	0.86	140.2
G_{14}	172.3	171.0	-1.70	0.96	165.4

over $[10^4, 10^6]$ bytes. The symbolic prediction T^l is presented in terms of α (T^l/T) which shows that the worst-case prediction error is well within 50 percent (27 percent on average with a maximum of 45 percent). Program and machine performance is captured by the PAMELA model with reasonable accuracy. On average, T under estimates T^m by only 4 percent which is almost entirely due to ignoring the bandwidth consumption of message acknowledgements. The T^t value (i.e., φ) demonstrates the prediction error of traditional static analysis.

3.8 APSP

In this section, we model a parallel All Pairs Shortest Path (APSP) algorithm implemented in terms of an MPI message-passing interface. The algorithm computes the shortest paths between all pairs of nodes in an N nodes weighted directed graph, by running Dijkstra's sequential algorithm for each node in parallel. The implementation is based on an SPMD farmer-worker scheme where P worker processors execute Dijkstra's algorithm, and a central farmer processor serves the workers by issuing target nodes and collecting the shortest path length information. As the work load of the sequential algorithm is highly node-dependent, worker scheduling is dynamic on an FCFS basis in order to avoid unnecessary performance loss (i.e., a selective communications scheme is used). Rather than considering the complex and analytically intractable MPI implementation, we focus on the parallel algorithm level, i.e., parallel clients that dynamically contend for service.

The performance of the APSP program is captured by the PAMELA model

$$
\begin{aligned}
apsp = \ & \textbf{par}\ (p = 1, P) \\
& \quad \textbf{seq}\ (i = 1, N/P)\ \{ \\
& \quad\quad \textbf{delay}(\tau_c)\ ; \\
& \quad\quad \textbf{use}(s, \tau_s) \\
& \quad \}
\end{aligned}
$$

The computation workload due to Dijkstra's algorithm is modeled by **delay**(τ_c). The **use**(s, τ_s) process models both the computation service (issuing nodes, collecting path lengths) as well as the communication costs as the server processor is involved in all incoming and outgoing traffic. The dynamic load balancing is implicitly modeled in terms of the **use** process, a constant iteration bound N/P, and a client-independent workload τ_c (a more detailed, i-dependent bound and workload would make no difference).

The MPI program is run on a 64 nodes partition of the DAS I distributed-memory machine (Pentium Pro nodes interconnected by Myrinet hardware [6]). The average computation workload τ_c is determined by dividing the sequential execution time of the N nodes APSP algorithm by N. The computation and communication workload τ_s is determined by separately measuring the computation workload for one service iteration, while adding communication workload for one node issue plus N vector collects (computed using a simple latency/bandwidth model that has been measured separately). Fig. 4 compares the execution time predictions (p) with the actual measurements (m) for two diamond graphs of $N = 256$ nodes and $N = 1,024$ nodes. Similar to the MRM example, the largest prediction error occurs near the server saturation point. For $N = 256$, the prediction error is 4 percent on average with a maximum of 10 percent.

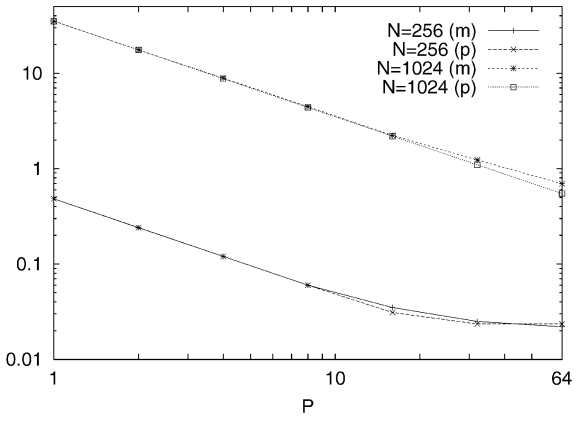

Fig. 4. APSP execution time [s] (N = 256 and 1,024).

For $N = 1,024$, the prediction error is 9 percent on average with a maximum of 27 percent.

4 Automatic Performance Modeling

As mentioned in Section 3, PAMELA expresses condition synchronization and mutual exclusion according to an orthogonal synchronization model. As the high-level, data-parallel programming model expresses condition synchronization (data dependencies), it falls within the orthogonal model. Consequently, the entire performance modeling process can be automated, providing an extremely high prediction ratio. The non-SP synchronizations that are present in some data-parallel computations are necessarily ignored in the mapping to the PAMELA SP condition synchronization model. As our results confirm, however, the corresponding increase in execution time prediction is small compared to the inherent accuracy of our symbolic analysis approach. Even for pathological non-SP synchronization patterns and task work load distributions, the error is shown to be quite limited [10]. In this section, we present the results of a PAMELA generator [15], implemented as part of a compiler that translates programs written in Spar/Java, a data-parallel Java dialect [34], for distributed-memory machines. The four applications involved in the experiment are MATMUL (Matrix Multiplication), ADI (Alternate Direction Implicit integration, vertical phase), GAUSS (Gaussian Elimination), and PSRS (Parallel Sorting by Regular Sampling). By supplying a few program annotations to provide information on data-dependent branches and loop bounds, the Spar/Java compiler automatically generates a PAMELA program model.

As explained in Section 3, model generation is based on the *source* code, rather than the generated SPMD message-passing code. Let the vector `V` be cyclically partitioned over P processors. A (pseudocode) statement

```
forall (i = 1:N)
   V[i] = .. * ..;
```

will generate

```
par (i = 1,N) {
    ... ;
    mult(i mod P) ;
    ...
}
```

(if the compiler uses a simple owner-computes rule). The machine model $mult(p)$ models multiplication workload charged to processor (index) p in terms of mutual exclusion (**use**), similar to the ADI modeling example in Section 3.6. The generated PAMELA program model is subsequently compiled, producing a simplified symbolic performance model in a matter of seconds. Our results demonstrate that this performance modeling approach delivers sufficient accuracy to enable correct selection between various programming alternatives.

4.1 Machine Model

The PAMELA program model is generated in terms of computation models such as $mult(p)$ (multiply), and the communication models $lmove(p)$ (local move), $gmove(p, q)$ (interprocessor move), and $bcast(p, P)$ (P processors broadcast). In the machine models, p and q denote processor indices. The distributed-memory machine used for the experiments is the DAS I machine mentioned in Section 3.8. Aiming to provide a mere proof of concept, the machine model is kept extremely simple. All local computations (integer, double precision) such as $mult(p)$ map to the same scalar workload model $\mathbf{use}(cpu_p, \tau_c)$. Local memory traffic is modeled by $lmove(p) = \mathbf{use}(cpu_p, \tau_l)$, where τ_l depends on the access stride (described later on). Point-to-point communication is modeled by $gmove(p, q) = \mathbf{use}(cpu_p, \tau_g)$, where τ_g depends on whether communication is generated in scalar mode (τ_g models latency) or vector mode (τ_g models bandwidth). Broadcasts are modeled by $bcast(p, P) =$ **seq** $(p = 0, P - 1)\ gmove(p, q)$. The absence of broadcast parallelism is due to the sequential code generated by the Spar/Java compiler. For the current virtual machine (Spar/Java compiler, Pentium Pro, Myrinet) communication cost is dominated by CPU overhead rather than network delays. Hence, no explicit network contention model such as in Section 3.7 is required.

The machine parameters τ_c, τ_l, and τ_g are measured at the Spar/Java virtual machine level. The local communication parameter τ_l is determined by a simple matrix computation benchmark

```
for (k = 1:K)
    for (i = 1:N)
        for (j = 1:S:N)
            statement;
```

where $N = 1,024$ represents a matrix dimension, S denotes the stride, and `statement` has the forms `V[i] = A[i,j]`, and `A[i,j] = V[i]`, to distinguish between a matrix load and store, respectively. Although simple, the benchmark is representative for applications where matrices are accessed that exceed the 256 Kbytes L2 cache capacity. As the 32 bytes cache line size accommodates four matrix elements, the fraction of cache misses per store is proportional to S for $S = 1, \ldots, 4$. Measurements show that τ_l is approximately proportional to $\min(S, 4)$ for matrix stores, while the stride sensitivity for loads is less. The local computation parameter τ_c is determined by extending `statement` with various simple arithmetic expressions (additions, subtractions, multiplication). The global communication parameter τ_g is determined by mapping `A` and `V` onto different processors for point-to-point transfers and broadcasts. Unlike point-to-point transfers, broadcasts are only measured for the `V[i] = A[i,j]` statement, where `V` is replicated on all processors.

In the following sections, all measurements ("m" in the plots) and predictions ("p" in the plots) are given terms of absolute execution time instead of speedup, in order to fully demonstrate the accuracy of the prediction technique. The sizes of the four test codes range from 40 (ADI) to 900 (PSRS) lines of Spar/Java code. The sizes of the generated PAMELA models range from 3,000 (ADI) to 5,000 (PSRS) lines of PAMELA code (including the models for runtime functions, and debugging comments). The sizes of the compiled symbolic performance models for the `main` function range from 15 lines (ADI) to 20 lines (GAUSS).

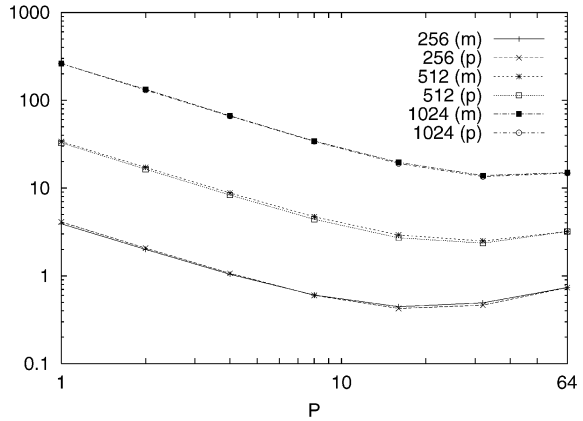

Fig. 5. MATMUL execution time [s] (N=256, 512, and 1,024).

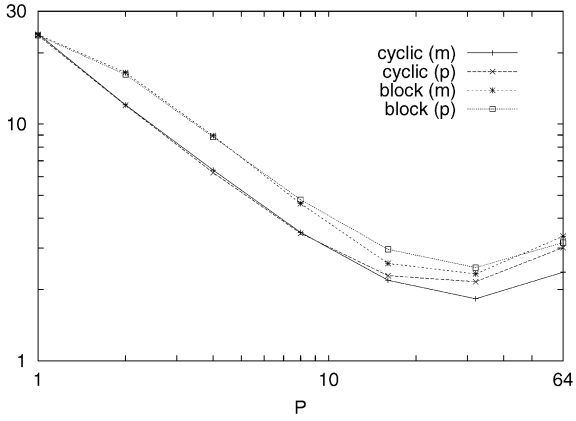

Fig. 7. GAUSS execution time [s] (j-i loop, cyclic and block mapping).

4.2 MATMUL

MATMUL computes the product of $N \times N$ matrices A and B, yielding C. A is block-partitioned on the i axis, while B and C are block-partitioned on the j-axis. In order to minimize communication, the row of A involved in the computation of the row of C is assigned to a replicated vector (i.e., broadcast). The experiment demonstrates the consistency of the prediction model for various N and P. The results for N = 256, 512, and 1,024 are shown in Fig. 5. The prediction error is 5 percent on average with a maximum of 7 percent.

4.3 ADI

The ADI experiment follows the discussion in Section 3.6. The prediction for a $1,024 \times 1,024$ matrix, shown in Fig. 6 clearly distinguishes between the block partitioning on the j-axis (vertical) and the i-axis (horizontal). The prediction error of the vertical version for large P is caused by the fact that the PAMELA model generated by the compiler does not account for the loop overhead caused by the SPMD level processor ownership tests. The maximum prediction error is therefore 77 percent but must be attributed to the current generator version, rather than the PAMELA method. The average prediction error is 15 percent.

4.4 GAUSS

The Gaussian elimination code illustrates the use of the PAMELA model in predicting the difference between cyclic and block partitioning, and also demonstrates the importance of modeling cache effects. The 512×512 matrix is partitioned on the j-axis. The submatrix update is coded in terms of an i loop, nested within a j loop. As the SPMD ownership tests apply to the j axis, the $j-i$ loop arrangement minimizes the overhead. The results shown in Fig. 7 illustrate the ability of the PAMELA model to predict the superior load balancing of a cyclic partitioning compared to a block partitioning. The prediction error for large P is caused by the fact that individual broadcasts may partially overlap due to the use of asynchronous communication, which is not modeled by $bcast$.

Fig. 8 shows the performance of a slightly modified code based on an $i-j$ loop arrangement. The results show that the cache performance improvement as a result of the arrangement outweighs the ownership test overhead. For a cyclic partitioning S scales with P which causes delayed speedup. For a block partitioning it always holds $S = 1$. Indeed, the model predicts that, for compute-bound settings, it is *block* partitioning that produces the best results. The prediction error is 13 percent on average with a maximum of 35 percent.

4.5 PSRS

PSRS sorts a vector X of N elements into a result vector Y. The vectors X and Y are block-partitioned. Each X partition is sorted in parallel. Using a global set of pivots X is repartitioned into Y, after which each Y partition is sorted

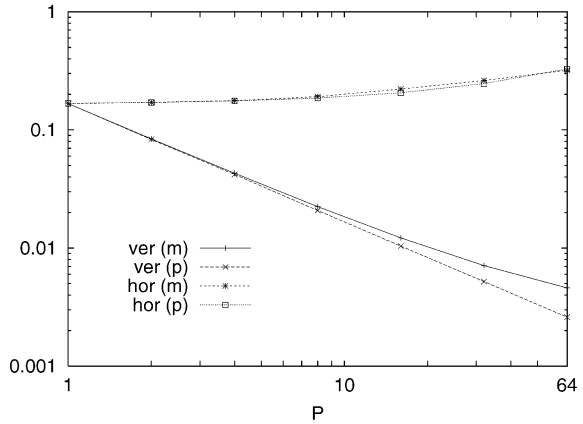

Fig. 6. ADI execution time [s] (vertical and horizontal data mapping).

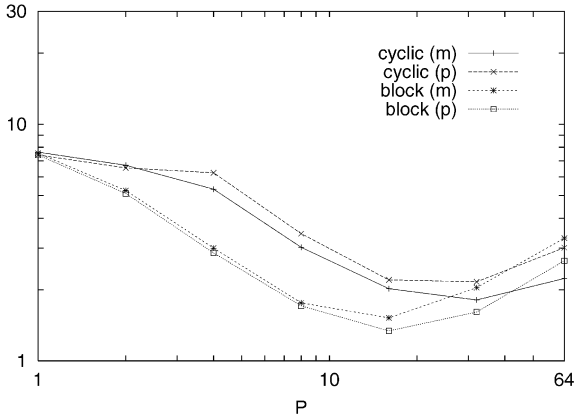

Fig. 8. GAUSS execution time [s] (i-j loop, cyclic and block mapping).

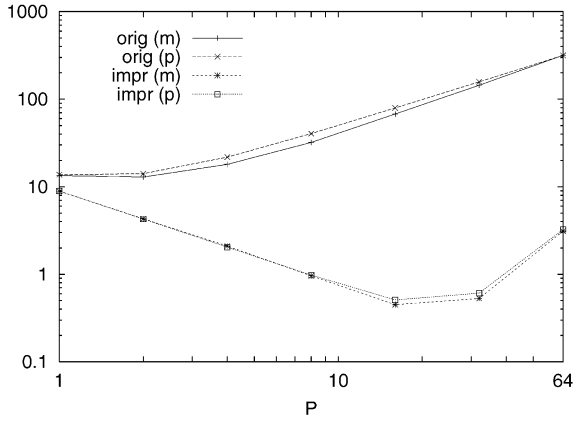

Fig. 9. PSRS execution time [s] (original and improved data mapping).

in parallel. Fig. 9 shows the prediction results for $N = 819,200$ for two different data mapping strategies. In the original program, all arrays except X and Y are replicated (i.e., pivot vector and various index vectors). This mapping, however, introduces a severe $O(NP)$ communication bottleneck. In the improved program version this problem is solved by introducing a new index vector that is also partitioned. The prediction error is 12 percent on average with a maximum of 26 percent.

4.6 Summary

For each of the four test programs, Table 3 lists the number of application-specific annotations (A), the compilation time (CT) of the PAMELA model generated by the Spar/Java compiler (generation time itself is negligible), the solution time (ST) of the model, and the average prediction error (err). The timing results are expressed in CPU s (450 Mhz Pentium II). In PSRS, a data-dependent, dynamic program, six annotations were necessary, only one of which required a few sequential profiling runs (the Quicksort function). Since machine benchmarking, sequential profiling, and PAMELA compilation time is amortized over many parameter settings, modeling cost cost is negligible. The average time to compile a symbolic performance model is 11 s. To illustrate the cost reduction impact of symbolic cost estimation, the solution costs are shown before (ST_1) and after automatic simplification (ST_2). The solution time ST_2 listed in the table reflects the simplification capability of the current PAMELA compiler version. For instance, large strings of maximizations, additions and multiplications of only two or three symbols are not yet properly reduced, as well as certain sum reductions in GAUSS although $O(1)$ solutions exist [13]. The table shows that the average time to obtain the performance plots currently ranges from 330 μs to 16.1 ms per point. As the numerical evaluation of the symbolic model is performed using the PAMELA compiler's internal numeric evaluator, more specialized evaluators may produce further speedup. The overall average prediction error is less than 10 percent with a maximum of 77 percent, which is due to trivial modeling inaccuracies, rather than the inherent inaccuracy of the PAMELA approach as explained earlier. Apart from providing a good scalability assessment, the model correctly predicts the best design choice in all cases.

TABLE 3
Prediction Statistics (MATMUL: $N = 1,024$)

program	A [-]	CT [s]	ST_1 [Ks]	ST_2 [ms]	err [%]
MATMUL	0	5.9	684	0.82	5
ADI	0	5.7	72	0.33	15
GAUSS	0	14.2	34	16.1	13
PSRS	6	18.1	32	1.84	12

5 Related Work

As in all performance modeling approaches, symbolic techniques aim at finding an acceptable tradeoff between accuracy and modeling cost. An important accuracy issue in static approaches is the way in which dynamic program and machine information is incorporated in the model. As acquiring this information involves some form of actual measurement, this term in the modeling cost equation may potentially degrade the prediction ratio. In abstract symbolic modeling approaches, the model is complemented by a carefully chosen measurement methodology. A typical example is the measurement of the *latency metric*, introduced by Zhang et al. [39], which accounts for all dynamic overheads for a particular program-machine combination. When measured for a different number of processors, the analytical model is capable of predicting application scalability with high accuracy. In the more detailed performance prediction approach by Xu et al. [37], measurement also plays a central role in determining, e.g., loop execution times at program level and, e.g., communication times at the machine level (all measured for a particular program-machine combination). The importance of the measurement cost has led them to explicitly address the issue in terms of the prediction ratio we cited in the introduction. In the above approaches, dynamic information is measured in a $program \times machine$ space, in order to provide the highest accuracy. Trading accuracy for prediction ratio, we separate program and machine modeling by using an explicit workload *interface* at the relatively fine grain (virtual) machine instruction level, similar to simulation (yet without the simulation costs). Consequently, dynamic information is measured in a $program + machine$ space, significantly improving model *portability* by amortizing machine benchmarking cost (e.g., τ_c, τ_l, τ_g) and program profiling cost (e.g., branching probabilities) over the combined space. In contrast to portable approaches such as BSP [33] and LogP[9], we model algorithm and architecture workload and synchronization structure in terms of a *program*, the symbolic model being compiled *automatically*.

Our symbolic performance modeling approach has been primarily inspired by the existing work in the static, compile-time prediction field, where the separation of program and machine spaces is commonly adopted. Although the idea of automatically generating symbolic performance models from programs originates from the sequential domain [36], most influential work is found in the parallel domain. Approaches

based on deterministic SP DAG analysis in the flavor of (2) include the work of Atapattu and Gannon [5], Balasundaram et al. [7], Clement and Quinn [8], Fahringer [11], Mendes and Reed [24], and Wang [35]. Approaches based on stochastic SP DAGs include the work of Lester [22], and Sahner and Trivedi [29]. None of the work addresses mutual exclusion, except in terms of architecture-specific forms of memory contention analysis [5], network contention analysis [11], and the benchmarking of collective (contending) communication procedures [7]. Although the mutual exclusion oriented DAG approaches mentioned in the introduction [1], [20], [23] are inherently superior in modeling contention, they are not considered in the above comparison because of their numeric approach.

Static approaches that take into account the effects of a limited number of processor resources include the work of Allen et al. [4], Polychronopoulos and Banerjee [28], Sarkar [30], and So et al. [32]. Similar to our approach the critical path prediction is augmented with a lower bound of comparable accuracy, which is derived using Graham's list scheduling theory [17]. An improvement has been described by Jain and Rajaraman [19], by iteratively applying the analysis to separate DAG layers, much in the flavor of (5). Unlike our generic approach, only processor resources are considered. Furthermore, the list scheduling theory cited only applies to *dynamic*, work conserving scheduling disciplines (unforced idleness), while *forced* idleness is quite common, especially in systems featuring *static* mappings that are poorly designed. Targeted at real-time systems, Shaw [31] presents a prediction scheme that computes both a lower and upper bound on the execution time. The scheme approximately accounts for critical sections and processor sharing but does not explicitly model memory and network contention.

6 Conclusion

Performance prediction cost is a critical success factor in the design of parallel applications and architectures. This particularly applies to the initial design stages where prediction cost is of more priority than optimum prediction accuracy, given the huge design space involved. In this paper, we have presented a symbolic performance modeling approach aimed at minimizing modeling time and solution time, while providing sufficient prediction accuracy to select the best design alternatives. The approach is based on modeling a parallel program and machine in terms of a performance simulation formalism called PAMELA. Instead of simulation the PAMELA model is compiled to a symbolic performance model. The compilation method integrates critical path analysis typical for compile-time cost estimation, with asymptotic bounding analysis from queuing theory, yielding a lower bound T^l on the simulation result T. This analytic tractability is the result of a careful restriction of the formalism's modeling power to the equivalent of deterministic SP DAGs (condition synchronization) combined with mutual exclusion.

The introduction of PAMELA minimizes modeling cost in two ways. Instead of going through a labor-intensive and error-prone process of directly deriving a symbolic performance model, only a PAMELA model needs to be constructed. As the PAMELA simulation formalism is intuitively close to the system under study, modeling cost is greatly reduced, making PAMELA an attractive tool to model and (analytically) reason about the performance impact of algorithmic and architectural designs. For high-level structured programming models such as the data-parallel model, PAMELA models can be directly compiled from the program. As annotation effort is relatively small, modeling cost nearly reduces to compilation cost, which is in the order of seconds. Moreover, as the performance model is parameterized, modeling cost (including program annotation, profiling, machine benchmarking) is amortized over many model evaluations.

Due to the inherent regularity (replication) of many parallel or sequential programs and data partitionings, the symbolic performance models are amenable to an aggressive symbolic simplification. The simplification that is part of the PAMELA compilation process dramatically reduces solution cost by many orders of magnitude compared to simulation. Combined with the very low modeling cost this results in an extremely high prediction ratio. For the data-parallel program experiments in Section 4, the solution cost currently ranges in the milliseconds. This cost is expected to decrease as the PAMELA compiler's symbolic simplification engine further matures.

Although a static analysis technique is used to compile PAMELA models to symbolic predictions, prediction accuracy is improved over traditional critical path analysis techniques due to the approximate, symbolic analysis of contention. Theory and experiments show that on average the inherent, worst-case prediction error of T^l is limited to a 50 percent underestimation of T, regardless of system size, whereas traditional static techniques, that account for condition synchronization only, entail errors that are virtually unlimited. In addition, it is shown that the potential underestimation can be predicted by the contention index θ that is compiled as a side result of T^l. PAMELA's restricted modeling power, required for analytic tractability, implies that parallel program/machine combinations be best modeled at the parallel algorithm level to avoid potential modeling errors due to the semantic gap between algorithm and implementation. In practice, this restriction hardly induces modeling inaccuracies. This is demonstrated by a number of theoretic and empirical modeling case studies, including a parallel benchmark program on a T800 transputer grid, and five parallel programs on a cluster of Pentium Pros connected by Myrinet (APSP, MATMUL, ADI, GAUSS, PSRS). Our results show that the average prediction error is less than 10 percent. It is also shown that this prediction accuracy is more than sufficient to assess application scalability (all applications), and to correctly select between alternative data partitionings (ADI, GAUSS) and code modifications (GAUSS, PSRS).

Acknowledgments

This research was supported in part by the European Commission under ESPRIT II LTR grant 28198 (the JOSES project). The DAS I partition was kindly made available by the Dutch graduate school "Advanced School for Computing and Imaging" (ASCI). The T800 machine was kindly made available by the Interdisciplinary Center for Computer-based Complex systems research Amsterdam (IC^3A). The author

extends his sincere gratitude to the anonymous referees for their valuable and insightful comments.

REFERENCES

[1] V.S. Adve, "Analyzing the Behavior and Performance of Parallel Programs." PhD thesis, Technical Report #1201, Univ. of Wisconsin, Madison, WI, Dec. 1993.

[2] V.S. Adve, M.K. Vernon, "The Influence of Random Delays on Parallel Execution Times," *Proc. 1993 ACM SIGMETRICS '93,* pp. 61-73, May 1993.

[3] M. Ajmone Marsan, G. Balbo, and G. Conte, "A Class of Generalized Stochastic Petri Nets for the Performance Analysis of Multiprocessor Systems," *ACM Trans. Computer Systems,* vol. 2, pp. 93-122, May 1984.

[4] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh, and V. Sarkar, "A Framework for Determining Useful Parallelism," *IEEE Proc. 1988 Int. Conf. Parallel Processing,* pp. 207-215, Aug. 1988.

[5] D. Atapattu and D. Gannon, "Building Analytical Models into an Interactive Prediction Tool," *Proc. ACM Supercomputing '89,* pp. 521-530, 1989.

[6] H. Bal, "The Distributed ASCI Supercomputer Project," *Operating Systems Review,* vol. 34, pp. 76-96, Oct. 2000.

[7] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A Static Performance Estimator to Guide Data Partioning Decisions," *Proc. 3rd ACM SIGPLAN Symp. PPoPP,* Apr. 1991.

[8] M.J. Clement and M.J. Quinn, "Multivariate Statistical Techniques for Parallel Performance Prediction," *IEEE Proc. 28th Hawaii Int'l Conf. System Sciences,* pp. 446-455, Jan. 1995.

[9] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," *Proc. 4th ACM SIGPLAN Symp. PPoPP,* pp. 1-12, May 1993.

[10] A. González Escribano, A.J.C. van Gemund, and V. Cardeñoso Payo, "Performance Trade-Offs in Series-Parallel Programming Models," *Proc. 8th Int'l Workshop Compilers for Parallel Computers (CPC '00),* pp. 183-189, Jan. 2000.

[11] T. Fahringer, "Estimating and Optimizing Performance for Parallel Programs," *Computer,* pp. 47-56, Nov. 1995.

[12] A.J.C. van Gemund, "Performance Prediction of Parallel Processing Systems: The PAMELA Methodology," *Proc. 7th ACM Int'l Conf. Supercomputing,* pp. 318-327, July 1993.

[13] A.J.C. van Gemund, "Compiling Performance Models from Parallel Programs," *Proc. 8th ACM Int'l Conf. Supercomputing,* pp. 303-312, July 1994.

[14] A.J.C. van Gemund, "Performance Modeling of Parallel Systems," PhD thesis, Delft Univ. of Tech., Apr. 1996.

[15] A.J.C. van Gemund, "Automatic Cost Estimation of Data Parallel Programs," Tech. Report 1-68340-44(2001)09, Faculty of Information Technology and Systems, Delft Univ. of Tech., Oct. 2001.

[16] N. Götz, U. Herzog, and M. Rettelbach, "Multiprocessor and Distributed System Design: The Integration of Functional Specification and Performance Analysis using Stochastic Process Algebras," *Proc. SIGMETRICS '93,* 1993.

[17] R.L. Graham, "Bounds on Multiprocessing Timing Anomalies," *SIAM J. Appl. Math.,* vol. 17, no. 2, pp. 416-429, 1969.

[18] R.W. Hockney and I.J. Curington, "$(f_{1/2})$: A Parameter to Characterize Memory and Communication Bottlenecks," *Parallel Computing,* vol. 10, pp. 277-286, 1989.

[19] K.K. Jain and V. Rajaraman, "Lower and Upper Bounds on Time for Multiprocessor Optimal Schedules," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, pp. 879-886, Aug. 1994.

[20] H. Jonkers, A.J.C. van Gemund, and G.L. Reijns, "A Probabilistic Approach to Parallel System Performance Modelling," *IEEE Proc. 28th Hawaii Int'l Conf. System Sciences,* pp. 412-421, Jan. 1995.

[21] S.S. Lavenberg, *Computer Performance Modeling Handbook.* Academic Press, 1983.

[22] B.P. Lester, "A System for Computing the Speedup of Parallel Programs," *Proc. 1986 Int'l Conf. Parallel Processing,* pp. 145-152, Aug. 1986.

[23] V.W. Mak and S.F. Lundstrom, "Predicting Performance of Parallel Computations," *IEEE Trans. Parallel and Distributed Systems,* vol. 1, pp. 257-270, July 1990.

[24] C.L. Mendes and D.A. Reed, "Integrated Compilation and Scalability Analysis for Parallel Systems," *Proc. Int'l Conf. Parallel Architectures and Compiler Technology '98,* pp. 385-392, Oct. 1998.

[25] W. Oed and O. Lange, "On the Effective Bandwidth of Interleaved Memories in Vector Processor Systems," *IEEE Trans. Computers,* vol. 34, pp. 949-957, Oct. 1985.

[26] PAMELA Project Web Site http://rama.pds.twi.tudelft.nl/~gemund/pamela.html.

[27] Parsytec Computer GmbH, *Parix Release 1.2 Software Documentation,* Mar. 1993.

[28] C.D. Polychronopoulos and U. Banerjee, "Speedup Bounds and Processor Allocation for Parallel Programs on Multiprocessors," *Proc. 1986 Int'l Conf. Parallel Processing,* pp. 961-968, Aug. 1986.

[29] R.A. Sahner and K.S. Trivedi, "Performance and Reliability Analysis Using Directed Acyclic Graphs," *IEEE Trans. Software Eng.,* vol. 13, pp. 1105-1114, Oct. 1987.

[30] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, 1989.

[31] A.C. Shaw, "Deterministic Timing Schema for Parallel Programs," *Proc. 5th Int'l Symp. Parallel Processing,* pp. 56-63, 1991.

[32] K. So, A.S. Bolmarcich, F. Darema, and V.A. Norton, "A Speedup Analyzer for Parallel Programs," *Proc. 1987 Int'l Conf. Parallel Processing,* pp. 653-661, Aug. 1987.

[33] L. Valiant, "A Bridging Model for Parallel Computation," *Comm. ACM,* vol. 33, pp. 103-111, Aug. 1990.

[34] C. van Reeuwijk, A.J.C. van Gemund, and H.J. Sips, "Spar: A Programming Language for Semi-Automatic Compilation of Parallel Programs," *Concurrency: Practice and Experience,* vol. 9, pp. 1193-1205, Nov. 1997.

[35] K-Y. Wang, "Precise Compile-Time Performance Prediction for Superscalar-Based Computers," *Proc. ACM SIGPLAN PLDI '94,* Orlando, pp. 73-84, June 1994.

[36] B. Wegbreit, "Mechanical Program Analysis," *Comm. ACM,* vol. 18, pp. 528-539, Sept. 1975.

[37] Z. Xu, X. Zhang, and L. Sun, "Semi-Emprical Multiprocessor Performance Predictions," *J. Parallel and Distributed Comp.,* vol. 39, pp. 14-28, 1996.

[38] J. Zahorjan, "Balanced Job Bound Analysis of Queueing Networks," *Comm. ACM,* vol. 25, pp. 134-141, Feb. 1982.

[39] X. Zhang, Y. Yan, and K. He, "Latency Metric: An Experimental Method for Measuring and Evaluating Parallel Program and Architecture Scalability," *J. Parallel and Distributed Comp.,* vol. 22, pp. 392-410, 1994.

Arjan J.C. van Gemund received the BSc degree in physics, the MSc degree (cum laude) in computer science, and the PhD degree (cum laude) in computer science, all from Delft University of Technology, the Netherlands. Between 1980 and 1992, he joined various Dutch companies and institutes, where he worked in embedded hardware and software development, acoustics research, fault diagnosis, and high-performance computing. In 1992, he joined the Information Technology and Systems Faculty at Delft University of Technology, where he is currently serving as an associate professor. He is also the cofounder of Science and Technology, a Dutch software and consultancy startup. His research interests are in the areas of parallel systems performance modeling, fault diagnosis, and parallel programming and scheduling. He is a member of the IEEE Computer Society.