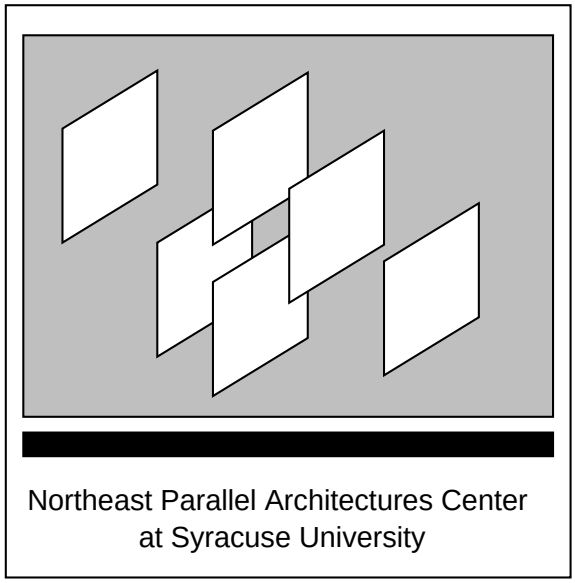
SCCS-479



An Interpretive Framework for Application Performance Prediction

Manish Parashar, Salim Hariri, Tomasz Haupt and Geoffrey C. Fox
parashar@npac.syr.edu

April 21, 1993

SYRACUSE UNIVERSITY
Northeast Parallel Architectures Center
111 College Place, Room # 3-201 · Syracuse, New York 13244-4100
Tel: (315) 443-1722 · Fax: (315) 443-1973

An Interpretive Framework for Application Performance Prediction

Manish Parashar, Salim Hariri, Tomasz Haupt and Geoffrey C. Fox
Northeast Parallel Architectures Center
Syracuse University
parashar@npac.syr.edu, hariri@cat.syr.edu

Contents

An Interpretive Framework for Application Performance Prediction

Manish Parashar, Salim Hariri, Tomasz Haupt and Geoffrey C. Fox
Northeast Parallel Architectures Center
Syracuse University
parashar@npac.syr.edu, hariri@cat.syr.edu

Abstract

The last few decades have seen an impressive developments in every aspect of parallel computing technology; viz. processing and storage technology, interconnect technology and software technology. Although these systems incorporate large amount of computing power, they are not general enough to efficiently support today's computation-intensive problems (e.g. the Grand Challenges), that warrant multiple computational models and levels of parallelism. We believe that the future of parallel computing lies in the integration of the plethora of "specialized" architectures into a single Heterogeneous High Performance Computing (HHPC) environment that allows them to cooperate in solving complex problems. Software development in such an environment is a non-trivial process and requires a thorough understanding of the application and the architecture. Evaluation tools form a critical part of any software development environment. These tools enable the developer to visualize the effects of various design choices on the performance of the application, to study the scalability of the application with system and problem size and to investigate the effects of changes in system run-time status and its configuration on the application execution. The objective of this paper is to propose an interpretive model for a source driven performance prediction framework which can meet challenges presented by an HHPC environment. The model provides a comprehensive characterization methodology to abstract and parameterize the behavior of the application and the computing environment. Interpretive techniques are then used to predict the performance of the abstracted application on the abstracted computing environment. A prototype performance prediction framework has been developed for the iPSC/860 using the proposed interpretive model. Numerical results obtained on this system are presented. These results confirm the potential of interpretive performance prediction techniques and their applicability to an HHPC environment.

1 Introduction

The last few decades have seen an impressive developments in every aspect of parallel computing technology; viz. processing and storage technology, interconnect technology and software technology. High performance computer systems today, include SIMD architectures like CM2 and DECmpp, shared memory MIMD, vector and pipelined architectures like the CRAY C90, NEC SX3, IBM POWER/4, distributed memory MIMD machines like the Paragon XP/S and iPSC/860 from Intel, the CM-5 from TMC, the KSR1, transputer based machines like the Parsytec GC, special purpose architectures like the BBN MP2000, etc.

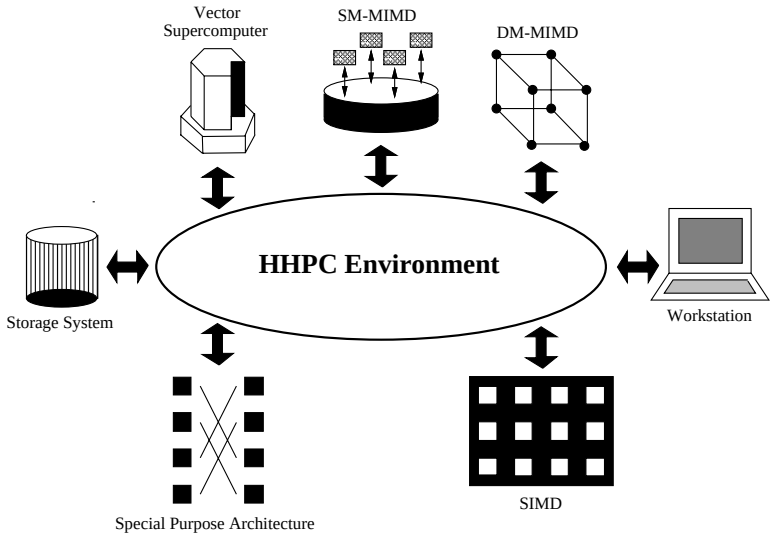


Figure 1: The Heterogeneous High Performance Computing Environment (HHPC)

Each of the above architectures have resulted from a unique set of trade-off's in system parameters and design decisions causing specific architectures to favor certain computational models and thereby deliver maximum performance only to a specific set of applications which lend themselves to one of those computation models. Further, this narrow applicability of current architectures has prevented them from being cost-effective. As a result, even though these architectures incorporate a large amount of computing power, they are not general enough to efficiently support today's computation-intensive problems, that warrant multiple computational models and levels of parallelism (e.g. The Grand Challenges [1]). Tackling applications of this magnitude and diversity would require a general, cost-effective, scalable, yet powerful computing model which will be able to efficiently support its varied computational and communication requirement. It is this realization that has spurred intense research in heterogeneous computing environments [2, 3, 4, 5, 6, 1, 7, 8].

We believe that the future of parallel computing lies in the integration of the plethora of "specialized" architectures into a single Heterogeneous High Performance Computing (HHPC) environment that allows them to cooperate in solving complex problems. The HHPC environment will capitalize on existing architectures and on current advances in computing, networking and communication technology to provide efficient, cost-effective, scalable, high-performance distributed computing.

Software development in any Parallel/Distributed environment is a non-trivial process and requires a thorough understanding of the application and the system architecture. This apparent from the fact that currently, applications are able to achieve only a fraction of peak available performance [7, 1]. The problem further intensifies as systems evolve into HHPC environments. Although, an HHPC environment provides the user with enormous computing power and a great deal of flexibility in using this power, this flexibility implies increased degrees of freedom which have to be optimized in-order to exploit the available potential.

For example, during software development in an HHPC environment, the developer is required to select the optimal hardware configuration for the particular application, the best decomposition of the problem on the selected hardware configuration, the best communication and synchronization strategy to be used, etc. The set of reasonable alternatives that have to be evaluated in such an environment, is very large and selecting the best alternative among these is a non-trivial task.

It is imperative therefore, that evaluations tools be provided as part of any HHPC software development environment, which enable the developer to visualize the effects of various design choices on the performance of the application, to study the scalability of the application with system and problem size and to investigate the effects of changes in system run-time status and its configuration on the application execution. Further, these tools must provide information about the contribution of various factors effecting the performance of the application. Finally, there is a need for a symbiotic relationship between the evaluation tools and other development tools (such as mapping, analysis, and optimizing tools) so as to complete the feedback loop of the "develop-evaluate-tune" cycle.

Predictive evaluation techniques have the potential of being effectively applied to software development in an HHPC environment. These techniques enable the estimation of performance of a given implementation on a given hardware configuration without having to actually execute it. Further it allows the developer to evaluate individual modules of the application during development instead of having to implement the entire application before evaluation. Heuristics and abstractions can be incorporated into the prediction models to significantly reduce the time required for evaluation; thereby allowing a greater number of options to be evaluated and increasing the probability of finding the best implementation. A key part of the evaluation process is to evaluate the application under different run-time conditions like loads, contentions etc. and in the presence of faults. The scalability of the application with problem and machine size also needs to be evaluated. Prediction techniques enable the user to experiment with the different scenarios in a cost-effective manner (both, in terms of resources required and time taken).

Conventional evaluation/prediction tools and techniques are either tuned to specific systems and cannot incorporate the heterogeneity and dynamic nature of an HHPC environment or are too general and lack feasibility and accuracy needed in HHPC software development. Analytic models for parallel/distributed systems [9, 10, 11] lead to large state spaces which result in large evaluation times. These techniques can be made tractable by introducing simplistic assumptions, but this makes them unrealistic and inaccurate. Monitoring techniques [12, 13, 14, 15], on the other hand, require extensive experimentation and data collection on the actual hardware. The process is not feasible or cost-effective since parallel/distributed systems are expensive resources and usually not freely available for such experimentation. Further, programming, running and data collection on most parallel/distributed systems is a tedious process and exhaustively evaluating the possible alternatives is usually not practical. Finally, these techniques are intrusive and can alter the execution of the application.

In this paper, we present the design of a performance prediction framework targeted to a general HHPC

environment. The framework uses a novel interpretive approach to provide accurate and cost-effective performance prediction. A comprehensive characterization methodology is proposed to abstract the system and application components of the HHPC environment into a set of well defined parameters. An interpreter engine then interprets the performance of the abstracted application in terms of the parameters exported by the abstracted system. The parameters required to abstract a system component can be generated off-line using existing techniques or even system specification. Further, the most effective techniques can be selected for each component. This flexibility allows the approach to be efficient, cost-effective, while maintaining desired accuracy and enabling the developer to capitalize on existing research. Further, the approach allows the developer to introduce heuristics into the evaluation and to experiment with various run-time and "What-if ?" scenarios. The evaluation approach can be applied to the application (or part of the application) and at any stage of the development process. The performance measures generated by the framework provide information about all aspects of the application and at all levels of the application, i.e. application level, node level, process level, procedure level, etc. A prototype system has been developed for the iPSC/860 hypercube. Our experience with this system and the numerical results obtained confirm the potential of interpretive performance prediction techniques and their applicability to an HHPC environment.

The rest of this paper is organized as follows: Section 2 discusses existing evaluation techniques and briefly describes existing evaluation tools. Section 3 introduces the interpretive model and describes the structure of the performance prediction framework and its modules. The characterization and abstraction methodology proposed is described in detail. An interpretive algorithm is defined and its application to interpret the performance of applications is described. Section 4 presents some numerical results obtained on a prototype system implemented on the iPSC/860 hypercube. Section 5 presents some concluding remarks.

2 Existing Evaluation Tools & Techniques for HHPC

Existing evaluation tools and techniques can be classified into the following categories: (1) Analytic, (2) Simulation, (3) Monitoring and (4) Estimation. In addition, some of the proposed systems make use of combination of the above techniques and are categorized as (5) Hybrid. In what follows we briefly discuss each of these categories.

2.1 Analytical Techniques

The development of accurate analytic models for concurrent hardware and software is an extremely difficult process because of the complex interactions that exist in such an environment. Further, modeling parallel operations leads to very large state spaces which makes them intractable and thereby infeasible. Another problem is the determination of appropriate probabilities and probability distributions for the various

parameters that are required in such models. These issues are usually resolved by making simplistic assumptions and approximations. Although these simplifications allow the models to be evaluated in a reasonable time, they are associated with a loss in accuracy. Another drawback of analytic models is that the performance information that can be obtained and the levels at which it can be obtained is limited. Finally, heterogeneity cannot be easily incorporated into analytic models.

A number of general analytic models have been developed to model both hardware [9, 16] and software [10, 17, 18, 19]. Analytic models for different interconnections and networks as well as transport protocols have also been proposed [20, 21]. Models for heterogeneous systems are proposed in [22, 23] but are limited to fork-join type of parallelism and apply to fixed system configurations. An analytic performance prediction technique has been proposed by B. Qin and R.A. Ammar in [11]. The approach followed is to approximate the parallel flow graph into a sequential flow graph by replacing well defined parallel structures by a single node with the same time-cost as the parallel structure. A more general analytic model for performance prediction on shared memory multiprocessors has been proposed by D. P. Siewiorek et al. in [24].

2.2 Simulation Techniques

Techniques in this category, simulate the hardware and the actual execution of a program on that hardware. However, most current computing systems are extremely complex and simulating these architectures in an intricate process requiring significant amounts of time and computing resources. This problem further intensifies in an HHPC environment. More feasible approaches sacrifice some accuracy for acceptable evaluation times and computing requirements by using approximations. These approaches include Distribution-Driven Simulations wherein the simulation is driven by statistical models and Trace-Driven Simulations which use execution traces gathered using instrumentation and monitoring techniques to drive the simulation. The former approach suffers from drawbacks of analytic techniques while the latter technique requires execution on the actual hardware to generate the traces. This technique also suffers from the problem inherent in monitoring approaches which are discussed in the following section.

The performance prediction package for the CEDAR shared memory multiprocessor (CPPP) [25] primarily uses normal Instruction-Driven simulation and handles loop parallelism (Doall, Doaccross). Because of the large execution times the CPPP system provides two additional tools for performance prediction which are less accurate but faster.

Execution-Driven Simulation is used in the Rice Parallel Processing Testbed (RPPT) [26]. This approach advances simulation time only when the parallel processes interact, thereby interleaving the execution of the simulation model and the program. This approach reduces simulation overhead at the cost of accuracy.

The SiGLe system (Simulator at Global Level) [27] provides special description languages to describe the architecture, application and the mapping of the application onto the architecure. Simlation techniques are then used on these descriptions to provide an environment for constructing, testing and evaluating the

implementations of applications on distributed systems.

2.3 Monitoring Techniques

Monitoring techniques have been extensively used for evaluating parallel and distributed systems; the reason being that they are more deterministic than the other approaches. Commonly used monitoring techniques include software monitoring by source code instrumentation, software monitoring by object code instrumentation, hardware monitoring and hybrid monitoring. Monitoring techniques, in general, give rise to a number of issues which need to be addressed. The source code instrumentation required in software and hybrid monitoring introduces the overhead of modifying the code and re-compiling it and then executing it to obtain the required information. Object code instrumentation may not be feasible in an HHPC environment because of the different object code formats. Hardware monitoring, is expensive and requires special resources. Monitoring, in general involves a trade-off between the accuracy of the evaluation and the intrusiveness of the instrumentation. Not only does monitoring effect the execution of the application, it effects the accuracy of the perforamnce measures since the overheads introduced by the monitoring procedures itself are difficult to account for. Monitoring requires that the application be executed on the actual hardware so that evaluation data can be gathered. This procedure can be expensive in term of time required and resources needed (both computing and storage). Further, the technique cannot be applied to evaluate new system configurations or new designs nor can it be used to evaluate algorithmic templates or parts of the application. Finally, these techniques are dependent on the hardware or software instrumentation/monitoring probes and are sensitive to their operation and malfunctions. The heterogeneity of the HHPC environment introduces the additional problem of trace data representation and interpretation. Since evaluation in such an environment involves evaluating the interaction between processes running on different heterogeneous systems, it is necessary that instrumentation and monitoring techniques be portable across the platforms and that trace data formats be compatible so that inter-process communications and synchronization can be evaluated.

With respect to the software development process in an HHPC environment, evaluating the different alternatives available or the various run-time scenarios using monitoring techniques would require, implementing, running and then analyzing the trace data collected, for each of these alternatives. This is not practical since implementing the different alternatives involves considerable time and effort. In addition, it prevents the automation of the software development process. Further, most real-world applications require considerable time to execute (even on parallel computers). This execution time added to the time required to implement and evaluate alternatives can lead to impractical development times. Finally parallel computers are expensive resources and may not always be available for such "experimentation". Some existing evaluation tools using monitoring techniques are described below.

The IPS-2 system [12] uses software monitoring techniques to gather execution statistics and presents performance data to the user in a hierarchical form consisting of 4 levels, viz. Program level, Machine

level, Process level and Procedure level. The instrumentation is achieved using a compile time option.

The JEWEL system [28] is a distributed measurement environment and is made up of four functional blocks; viz. the system under test (SUT), the data collection and reduction system (DCRS), the graphical presentation system (GPS) and the experiment control system (ECS). The system allows the user to select the aspect (topic of interest), the level of detail and the performance index to be viewed.

The ZM4/SIMPLE system [14, 29] developed at University of Erlanghan, Germany, is a performance evaluation environment based on hybrid event-driven monitoring. It proposes a model-driven instrumentation methodology which uses a monitoring model as the desired level of abstraction. ZM4 is a special purpose hardware monitoring system which interfaces with the SIMPLE environment. SIMPLE also provides the capability of using the trace data collected to generate runtime distributions and branching probabilities, which can be used in an analytic performance model for performance prediction.

The performance evaluation tool within the TOPSYS parallel application development environment [13], PATOP, uses object-code instrumentation and hardware monitoring techniques to gather trace data. PATOP measures idle times, delays (in queues) and resource utilization and displays them at system, node or programming model object levels. It interfaces with the VISTOP visualization/animation tool which has the capability of displaying performance statistics online. Statistics of the host program, however cannot be monitored. TOPSYS has been implemented on the iPSC/2 and iPSC/860 hypercube systems.

The INCAS project [30] at the University of Kaiserslautern uses hybrid monitoring techniques in its performance measurement tools. It uses special hardware support based on the Test and Measurement Processor (TMP), which is a part of each node in the multicomputer system. Individual TMP's are connected via a TMP interconnection network to a central test station. The monitoring approach used in this system has very low interference but lack a global time base to synchronize the TMP's.

2.4 Estimation Techniques

Estimation techniques use heuristics, assumptions and simplifications to make the evaluation more tractable and cost-effective. Estimations are typically introduced while choosing distributions or probabilities or in reducing the solution space in the case of analytic techniques. They are used to simplify the operation model of the hardware in case of simulation techniques. In monitoring techniques, they are used to reduce the amount and level of instrumentation required. These techniques generally result in the reduction of accuracy in the obtained evaluations. Further, most heuristics or assumptions used are valid only for a particular set of systems or applications. Existing approaches using estimation techniques are discussed below.

Alan Sussman [31] has proposed a performance estimation technique to be used for automatic mapping of programs onto distributed memory architectures. The approach used is to create an execution model of the application description at compile time. Execution models have been defined for a set of structured

mapping techniques like block and interleaved data partitioning and forall loop body partitioning. The output of the system is the estimated execution time for the particular mapping. The system is targeted to the Warp systolic array machine.

An alternate approach to performance estimation of data decompositions on a distributed memory machine has been proposed by Balasundaram et al. in [32]. This approach uses a set of "training routines" to benchmark the performance of the the architecture. These benchmarks are then used by the estimator to estimate the cost of a particular data decomposition. The estimator is targeted to loosely synchronous problems implemented as SPMD programs using the Cubix paradigm and with no overlap between communication and computation. The synchronization delay is not modeled. The user is prompted for unknown variable and branch probabilities. This system is a part of the ParaScope parallel programming environment [33].

The Sigma editor of the FAUST system [34, 35] assists the developer in re-targeting and optimizing application code by estimating its performance on the hardware. Sigma analyzes the assembly code for each node, generated by the compiler, to estimate the execution time of parallel loops present. Analytic models are incorporated to estimate the performance of the memory system. The performance estimation is provided in terms of inputs and other unknown variables. Sigma estimates the performance of loops only in structured scientific applications and is targeted to the Alliant FX/8 architecture.

The MARC environment (MApping Routing Configuring) [36] developed at the IAM of the University of Berne uses performance information to determine the appropriate mapping of application components onto the distributed system. The approach uses monitoring to gather application execution statistics on a single processor. This information is then used to estimate the performance on multiple processors. The tool has been developed for Occam programs running on transputer based systems.

An approach for predicting the end-to-end performance for the Internet is presented in [37]. It uses experimental data to develop a predictive model to the predict the latency and bandwidth for the Internet.

An alternate approach to predicting the performance of distributed memory multiprocessors is presented D. Poplawski et al. in [38]. The approach, called "Synthetic Models" is based on the premise that performance on distributed memory machines is more dependent on communication performance than computation. A synthetic model of an application is generated by replacing the computation by empty delay loops with delays equal to the estimated computation time, and retaining only the communication structure of the application.

A general model for performance prediction using deterministic techniques is presented in [39]. The deterministic model views performance as the interaction of resources demanded by programs and provided by the multiprocessor system, in both, a system dependent and independent manner. This model parameterizes the architecture in terms of the structure and requirements of the application and then uses these parameters deterministically to express the execution time of the application. The model has been described for the Loughborough University NEPTUNE MIMD parallel computer system.

2.5 Hybrid Techniques

Hybrid techniques combine the approaches discussed above so as to overcome the drawbacks of the individual techniques and to obtain acceptable tradeoffs. Some existing evaluation tools using hybrid techniques are discussed below.

A hybrid performance prediction tool has been developed for the RP3 machine. The approach uses simplified analytic queueing models which are then tuned with the help of simulation results. The system requires hardware and software specifications (such as instruction mix, hit ratios, load, branch probabilities etc.) to be specified by the user.

An approach for performance prediction on the BBN GP1000 parallel processing system is presented in [40]. This approach builds load/store templates that can automatically characterize the performance of the architecture. Analytic models which match the experimental results of these templates, are then used to predict performance.

The Chitra system [41] uses monitoring techniques to generate a program execution sequence (PES). It then develops a homogeneous semi-Markov chain model fitting the PES. This model is used to predict the performance of the program for different parameters.

The performance predictor used to evaluate machine-level mappings is presented in [42]. This approach models the parallel program as a stochastic graph and uses approximation techniques to solve the model. The runtime of each node in the stochastic graph is modeled as a random variable and their distributions are obtained using hardware monitoring. Using these distributions, the overall runtime of the application graph is approximated.

3 An Interpretive Model for Performance Prediction

It is clear from the discussion above, that there exist a large number of tools and approaches for evaluating the performance of applications on parallel/distributed computers. Although these tools have been used effectively to model particular systems, they have a narrow applicability and can not incorporate the heterogeneity and dynamic nature of an HHPC environment. General methodologies that have been developed, lack the feasibility and accuracy needed in HHPC software development. The challenges presented by the HHPC environment to the software developer are summarized as follows: (1) Inherent lack of synchrony and global ordering; (2) Heterogeneity in architectures, programming models, data representation, communication and synchronization structure, etc.; (3) Increased design options and available degrees of freedom that have to be optimized; (4) Dynamically changing computing environment in terms of size, components, configuration, topology, etc. In this section we present an interpretive model for a source driven performance prediction framework which can meet these challenges. The model provides a comprehensive characterization methodology to abstract and parameterize the behavior of the application and the computing environment. Interpretive techniques are then used to predict the performance of the

Figure 2: A Interpretive Performance Prediction Framework for HHPC - Functional Block Diagram

agram of the performance prediction framework. The proposed framework consists of four modules: (1) Application Module, (2) Systems Module, (3) Interpreter Engine and (4) Output Module. Each module abstracts a specific set of components of the HHPC environment and presents a well defined interface to the rest of the system. A key feature of this framework is that each module is independent i.e. it is viewed by the rest of the system as a black box with the desired interface. This allows each module to be optimized independently. In what follows, we present a detailed description of each module.

3.1 Model Inputs

The proposed performance prediction framework requires the following inputs:

- An application description which could be the source of the entire or part of the application, an algorithmic description or a set of templates.
- A mapping description which associates tasks in the application description to components in the computing environment on which they are to be evaluated.

In addition to these essential inputs, the framework also accepts information about current system run-time status and parameters (e.g. load, existing faults, etc.), or about "What-if?" scenarios to be evaluated. If

these values are not provided, preset default values are used. If the inputs required by the application are not specified, the user is prompted for them during the course of the interpretation.

3.2 Systems Module

The systems module abstracts the target computing system into a set of parameters which are then exported to the Interpreter Engine. The abstraction is performed in a hierarchical manner, wherein, at each level of the hierarchy every unit is independent and is viewed by the other units as a black box with a well defined interface. A unit's interface can be generated using evaluation techniques best suited to that particular unit (e.g. analytic, simulation, specifications, etc.) The rest of this section presents a detailed description of the system abstraction model and the characterization methodology.

3.2.1 System Abstraction Model

The function of the system abstraction model is to provide an abstract representation of the underlying computing environment and to define the interface presented to the rest of the framework. The model hierarchically characterizes any heterogeneous network-based computing environment into a *System Abstraction Graph (SAG)*. SAG is a rooted tree such that each level of the tree represents a corresponding level in the characterization hierarchy. Figure 3 illustrates the SAG corresponding to the HHPC environments shown in Figure 4 (a description of each level of this graph and how it is generated is described in the following section). Each vertex of the SAG is a *System Abstraction Unit (SAU)* that represents the fundamental unit of abstraction at a particular level of the abstraction hierarchy. Each SAU is a tuple with 4 components: viz (1) Processing Component (P), (2) Memory Component (M), (3) Communication/Synchronization Component (C/S), and, (4) Input/Output Component (I/O); i.e. $SAU \equiv \langle P, M, C/S, I/O \rangle$. Each component of the tuple can be Compound, Simple, or Void. A compound component can be further decomposed into one or more levels in the hierarchy. A simple component represents the lowest level in the classification hierarchy and exports actual timing information required to abstract that component. A void component implies that the particular component is not applicable at that level. An SAU is considered compound if at least one of its components is compound. Further, every SAU has at least one component that is not void. Every leaf SAU in an SAG is simple. An SAG can now be formally defined as follows:

Definition 1 *The* **System Abstraction Graph (SAG)** *is a rooted tree*

$$\begin{aligned} & SAG = [\{\mathbf{SAU}\}, \varphi] \\ where: \quad & \forall SAU \in \{\mathbf{SAU}\}; \quad SAU \equiv \langle P, M, C/S, I/O \rangle \\ and: \quad & P, M, C/S, I/O \in \{Compound, Simple, Void\} \end{aligned}$$

and φ is a predecessor-successor relation defined such that $SAU_i \ \varphi \ SAU_j$; $SAU_i, SAU_j \in \{\mathbf{SAU}\}$ implies SAU_j is a subcomponent of SAU_i.

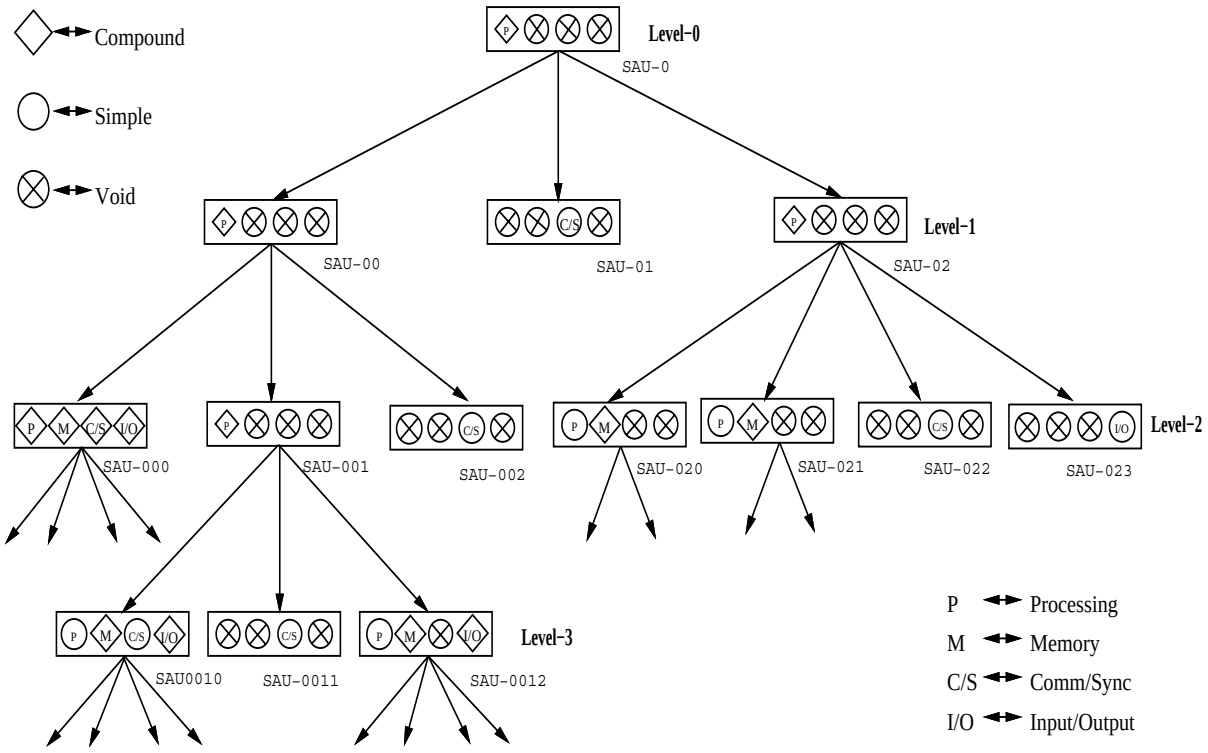


Figure 3: System Abstraction Graph (SAG)

In what follows we define the system characterization algorithm and illustrate the steps required to obtain the SAG for a particular computing environment using a running example. The four components of the SAU tuple and the parameters required to abstract them are also described.

3.2.2 System Characterization Methodology

The system characterization methodology abstracts each component of the underlying computing system according to the system abstraction model presented in Section 3.2.1. The methodology itself views the HHPC environment as a hierarchical structure. The highest level of the hierarchy consists of the entire HHPC system viewed as a single virtual machine while the lowest level of this hierarchy is made up of individual physical elements within each node in the environment. Units at each level of the hierarchy, are abstracted into four components (P,M,C/S,&I/O), corresponding to the SAU tuple. The processing component abstracts the processing capability of the unit while the memory component abstracts accesses to the memory system. The communication/synchronization component abstracts the communication/synchronization structure and protocol of the unit. The input/output component abstracts file and device I/O. As mentioned above, these components can be simple, compound or void. The characterization methodology proceeds recursively down the system hierarchy, generating units of finer granularity at each level. The process terminates at the level at which the parameters required to abstract the performance of all units (at that level) can be obtained with the desired level of accuracy and cost-effectiveness. The output of the system characterization methodology is a SAG as defined in Section 3.2.1. The system characterization algorithm is defined below:

Algorithm 1 *SystemCharcterize*

(1) Level0 ← $\langle \diamond, \otimes, \otimes, \otimes \rangle$ (Level 0 consists of a single compound processing component representing the entire computing system)
(2) foreach level in the characterization
- foreach compound component at that level
  characterize component
  end foreach

end foreach

End

In what follows, we use the HHPC environment shown in Figure 4 as a running example to demonstrate the use of the system characterization algorithm. The sample environment consists of two high speed backbone networks (HSBN) interconnected via a local area network (LAN). Each backbone network interconnects high performance computing elements (HPCE), storage systems (SS), etc. HSBN-1 interconnects a CM2 and an iPSC/860 while HSBN-2 interconnects 2 high performance workstations (HPW) and an SS. The methodology is illustrated in Figures 4-8.

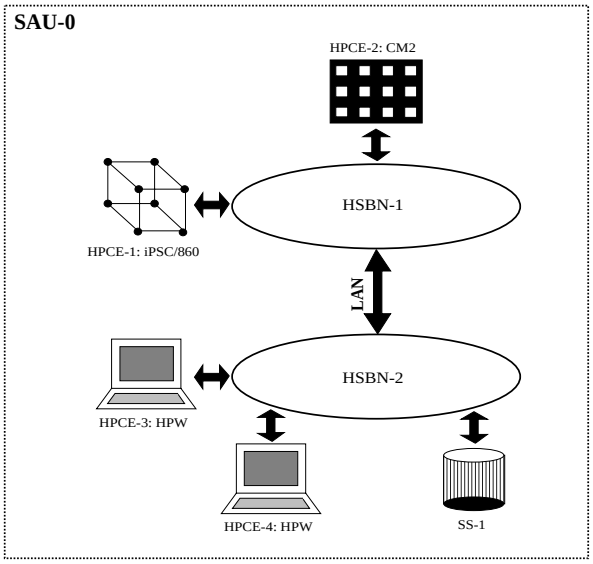


Figure 4: System Characterization Methodology - Level 0

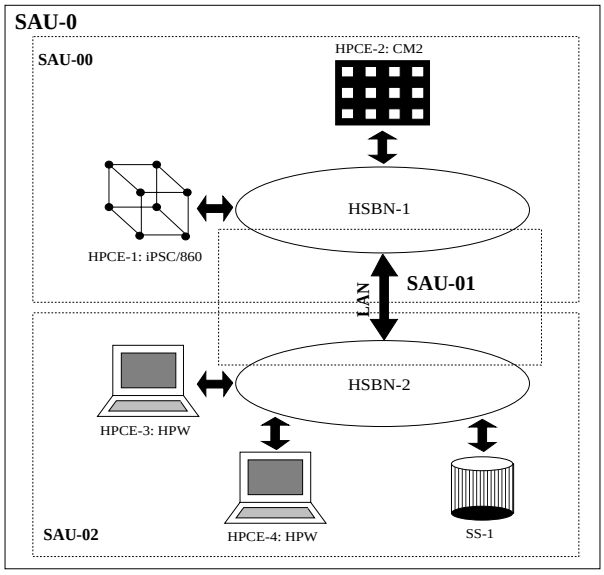


Figure 5: System Characterization Methodology - Level 1

Level 0 At the highest level of the abstraction hierarchy, there is a single system abstraction unit (SAU-0) which abstracts the entire system as a single virtual machine. At this level, we have a compound processing component. The memory, communication/synchronization and input/output components are void. Level 0 characterization of the system is shown in Figure 4. This level corresponds to step 1 of the system characterization algorithm.

Level 1 Level 1 of the characterization hierarchy is shown in Figure 5. This level contains three units; two of which (SAU-00 & SAU-02) contain compound processing components while the third unit (SAU-01) contains a simple communication/synchronization component. The processing components abstract the HPCE clusters on the two HSBN. The communication/synchronization component exports parameters which abstract the performance of the LAN interconnecting the two HSBN's.

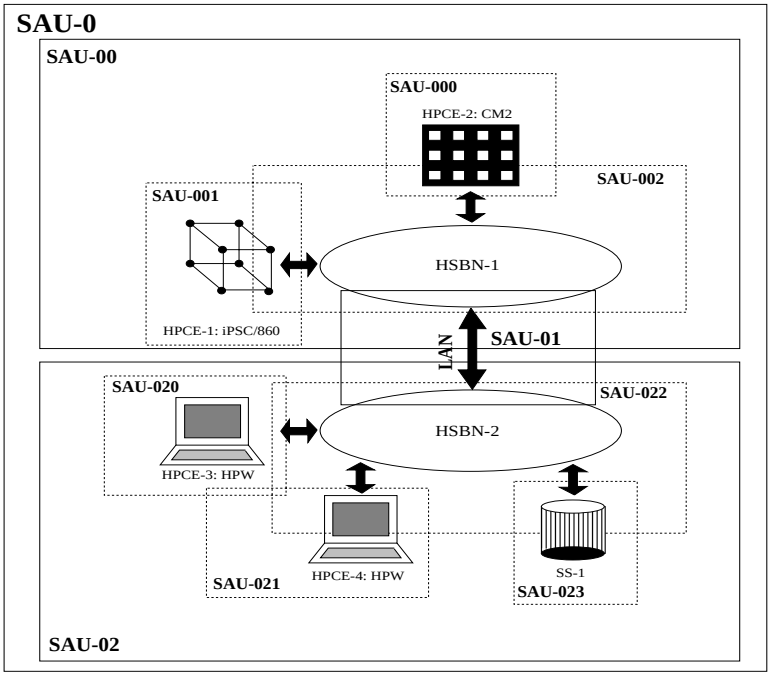


Figure 6: System Characterization Methodology - Level 2

Level 2 Level 2 of the characterization hierarchy is shown in Figure 6. At this level, the compound processing components of SAU-00 and SAU-02 are further broken down. SAU-00 is decomposed into 3 units (SAU-000, SAU-001 & SAU-002). SAU-000 abstracts the CM2 node of the environment. It consists of a compound processing component, a compound memory components, a compound communication/synchronization component, and a compound input/output component. SAU-001 abstracts the iPSC/860 hypercube system and consists of a compound processing component. The other three components are void at this level. SAU002 abstracts the HSBN and contains a simple communication/synchronization component. Its processing, memory and input/output components are void. SAU-02 is broken down into 4 units (SAU-020 through SAU-023). SAU-020 and SAU-021 abstract the two HPW's. They consist of a simple processing component and a compound memory component. SAU-023 abstracts the file server and consists of a simple input/output component only. SAU-022 abstracts the HSBN and contains a simple communication/synchronization component.

The subsequent levels of the characterization hierarchy are described in detail for the iPSC/860 (SAU-001). The particular unit is chosen since it provides insight into the applicability of this methodology to

an existing architecture and because the iPSC/860 is the target of our prototype system.

Level 3 At level 3, the iPSC hypercube system (SAU-010) is decomposed into 3 units; viz. SAU-0010, SAU-0011 and SAU-0012. SAU-0010 abstracts the actual cube. This unit consists of simple processing component, a compound memory components (memory hierarchy and main memory), a simple communication/synchronization component and a compound input/output component (i/o node and SS). SAU-0011 abstracts the interconnection between the System Resource Manager (SRM) and the iPSC/860 cube. It has a simple communication/synchronization component. SAU-0012 abstracts the SRM (or host) of the iPSC/860 hypercube system. It has a simple processing component, a compound memory component, and a compound input/output component.

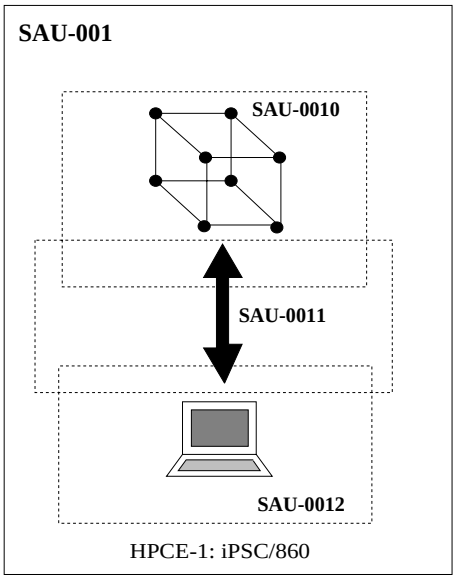


Figure 7: System Characterization Methodology - Levels 3

iPSC/860 (SAU 0010)	
Component	**Type**
Processing	Simple
Memory	Compound
Comm/Synch	Simple
Input/Output	Compound

Figure 8: System Characterization Methodology - Levels 4

Level 4 We use level 4 of the characterization hierarchy to illustrate the nature of the parameters exported by the various types of (simple) components since, the granularity of characterization at this level was sufficient to parameterize the iPSC/860 hypercube with the accuracy required by our prototype.

In the discussion below, only the iPSC/860 cube (SAU-0010) is considered, without the SRM. The prototype system however, does model the SRM and interactions between the node processors and the SRM while predicting performance.

It should be noted that the parameters exported by each component may be either in the form of absolute times or as a count of the system clock. Since the clock frequency is a part of the parameters exported by the processing component, the parameters can be converted to a standard format (real time) before being exported to the unit interface. Further, a parameter can have its values in the form of a range specifying

the minimum and the maximum. During interpretation, the user can decide which of these values to use depending on whether best-case or worst-case performance is required. The user may also define some function of these two values to be used (e.g. required time = (max - min)/2 + min). Finally, if a parameter is a known function of a run-time parameter (like system load), then this function can be exported to the interface. During interpretation, the current value of that parameter is used to evaluate the function. For the prototype system, we use a combination of timing values and clock counts. Also, wherever possible, a range of values is specified to provide additional flexibility. The values of these parameters have been obtained either via benchmarking runs performed on the hypercubes at the Center for Research in Parallel Computing (CRPC) at Rice University and at the Northeast Parallel Architectures Center (NPAC) at Syracuse University, or from iPSC/860 hardware documentation. In addition, performance information reported in [43, 44, 45] was referenced.

The parameters exported by each component are discussed below.

Processing Component The processing component models the cost of different operators occurring in a typical application description. The parameters exported by this component can be grouped into the following classes on the basis of the operators they model. Some key parameters required to abstract the different operators of the processing component are summarized in Table 1.

Algebraic Operators This class characterizes algebraic operators like $+$, $-$, $\times$ & $\div$. The number of clocks required by these operators when used with integer, real and floating point operations is exported. In addition, the overhead associated with mixed operations are also characterized. In case of the iPSC/860, mixed operation represents an integer operand used in an expression to generate a floating point result or alternately a floating point operand used in an expression to generate an integer result.

Iterative Operators The iterative operators class characterizes the overhead associated with iterative loops (e.g. DO-ENDDO, WHILE, etc.). This class abstracts the time required to check and update loop indexes and to make appropriate branches. The parameters exported by this component are of two types: those required to abstract a fixed overhead incurred every time a loop operator is encountered and those required to abstract the per-iteration overhead associated with each iteration of the loop. The overheads are determined for both types of loop limits; viz. when a stride (step) is present and when it is absent (i.e. the default stride of 1 is used).

Conditional Operators This class characterizes the overhead associated with conditional structures (e.g. IF-THEN-ELSE). Specifically, the time required for each condition, i.e. each "if", "else if" & "else" statement, is abstracted. In addition, the overheads associated with a fall-through and a branch-taken are also parameterized.

Processing Component			
Parameter Class	Parameters	iPSC/860 Characterization	
General			
	Unit Name	iPSC/860	
	Unit Clk Freq.	40 Mz. (2.5e-8 sec)	
	Unit Max Procs.	16	
Operators		Maximum	Minimum
Algebraic Opers			
	Int/Real Add/Sub	1	1
	Int/Real Mul	3	2
	Int Div	(1.256e-6/Clk)+2	(1.106e-6/Clk)+1
	Real Div	(2.391e-6/Clk)+5	(2.241e-6/Clk)+2
	Mix Oper: Int → Real	14	8
	Mix Oper: Real → Int	2	1
	.....		
Iterative Opers			
	No Step Limit Ovhd	10	5
	Per-Iter Ovhd (No Step)	14	8
	Step Lim Ovhd	(1.256e-6/Clk)+17	(1.106-6/Clk)+10
	Per-Iter Ovhd (Step)	17	10
	.....		
Conditional Opers			
	Condt Ovhd	1	1
	Branch Taken Ovhd	1	1
	.....		
Func/Sub Call Opers			
	Call Ovhd	2	1
	.....		
Library Characs			
	float()	15	7
	abs()	7	4
	.....		
Machine Spec Libs/Opers			
	numnodes() (NX/2)	(5.683e-7/Clk)	(5.552e-7/Clk)
	mynode() (NX/2)	(5.421e-7/Clk)	5.302e-7/Clk)
	.....		

Table 1: Processing Component Characterization

Function/Subroutine Call Operators This class characterizes the overheads associated with each call to a function or a subroutine or for a system call. The overhead consists of two components: the overhead of making the call itself and the overhead associated with each argument passed.

Library Characterization This class characterizes the performance of standard libraries (e.g. math libraries, etc.) and system routines (e.g. time, etc.).

Memory Component	
Parameters	iPSC/860 Characterization
General	
Cache Size	8 KBytes
Cache Blk Size	32 Bytes
Cache Assoc	2
Cache Repl Policy	Random
Cache Write Policy	Write-Back
MM Size	8 MBytes
MM Page Size	4 KBytes
Inst Cache Size	4 KBytes
Inst Cache Blk Size	32 Bytes
Inst Cache Assoc	2
Memory Hierarchy	Clks
Fetch	1
Fetch Miss	6
Store	2
Store Miss	4
Main Memory	Clks
Main Memory Fetch	3-1 (pipelined)
Main Memory Store	3-1 (pipelined)

Table 2: Memory Component Characterization

Machine Specific Operators/Libraries This class models operators, system calls and library calls which are specific to that particular machine. In case of the iPSC/860, this includes the "load" operator which loads a program onto a processing node and specific functions to obtain configuration information like the number of nodes used, node id's, etc.

Memory Component The memory component is actually a compound component which models the 2 level memory hierarchy. The first component models accesses to the entire memory hierarchy in a conventional manner. The second component abstracts accesses made directly to the second level of the memory hierarchy (i.e. the main memory). Typically it would lead to another level in the characterization hierarchy wherein each level would be modeled individually. However, for the compactness of description, the two levels have been combined (with no loss in detail) into a single simple level in the hierarchy. The parameters exported by the memory component are the fetch and store access times and the overheads associated with a read/write misses (if applicable). The physical size and organization of the memory hierarchy and replacement/writing policies used are also parameterized. Some key parameter used to abstract the memory component are listed in Table 2

Communication/Synchronization Component		
Component	Parameters	iPSC/860 Characterization
Specifications		
	Topology	hypercube
	Routing	e-cube, circuit switched
	Static Buffer Size	100 Bytes
Communication Structure		
Static Buffers		Time (sec)
	Startup Ovhd	70e-6
	XMission/Byte	0.42e-6
	Link Ovhd/Hop	11 e-6
	Receive Ovhd	-
	.....	
Dynamic Buffers		Time (sec)
	Startup Ovhd	175e-6
	XMission/Byte	0.36e-6
	Link Ovhd/Hop	33e-6
	Receive Ovhd	-
	.....	
Synchronization Structure		Time (sec)
	Sync Ovhd	67e-6
	.....	

Table 3: Communication/Synchronization Component Characterization

Communication/Synchronization Component This component models the communication and synchronization structure of the computing node. The parameters applicable to each structure are discussed below.

Communication Structure Typical parameters required to abstract the communication structure are the startup overheads (required for packing and marshaling messages), the actual transmission times per unit and network link, overhead associated per hop and the receive overhead (for unpacking and copying messages). Further, these parameters are exported for communication using static and dynamic buffers. Other specifications like the size of the static buffers, the routing scheme followed, etc. are also specified. The value of some key parameters for the iPSC/860 hypercube are listed in Table 3.

Synchronization Structure Parameters required to abstract the synchronization structure include the overheads required to create and check barriers and to lock/unlock and test global variables (if a shared memory space exists). The value of some relevant parameters for the iPSC/860 hypercube are listed in Table 3.

Input/Output Component The input/output component models file and device I/O for the computing unit. It exports parameters which provide two levels of abstraction. The first level models I/O at a higher level and parameterizes the opening and closing of units, positioning within units and the read/write operation to the unit. The second level exports device specific parameters like the seek time, rotational latency, data transfer times etc.

The Input/Output component for the iPSC/860 (SAU-0010) is compound and consists of 3 sub-units; viz. the i/o processor, the data transfer channel and the storage system. We are currently in the process of modeling these units. The relevant parameters however, are not available at the time of the writing of this paper.

In the description presented in this paper, we stop the characterization at level 4 since it was possible to parameterize the iPSC/860 unit at this level with the granularity and accuracy required. However, the same procedure can be applied recursively to achieve a finer granularity, i.e. considering each physical component of the iPSC/860 as a separate unit and modeling it separately.

The system characterization methodology proposed in this section is general enough to accommodate any size, configuration and composition of the HHPC environment, while retaining sufficient detail to model its performance. Further, it models the interactions between components while allowing the developer to select appropriate models for generating the parameters for each component. The granularity of the information exported at each interface defines the accuracy of the prediction at that interface. The developer thus has the flexibility to model key components in detail and use approximation for component not critical to the performance. Finally, the hierarchical structure of the model reflects the structure of the environment and can be used to obtain and abstract view of the HHPC environment while generating system level mappings.

3.3 Application Module

The application module is responsible for abstracting the application description into a set of parameters which define its structure and performance. These parameter are then exported to the interpreter engine so that their performance can be interpreted in terms of the parameters exported by the systems module. The application module is composed of two components: (1) Machine Independent Abstraction Module and (2) Machine Specific Filter. The machine independent abstraction module is responsible for characterizing the application into an abstraction graph according to application abstraction model defined below. This graph is then passed through the machine specific filter where it is augmented to incorporate machine specific information based on the mapping inputs provided. The application module is designed to be general enough to handle any structured application description. In what follows, we first define the application abstraction model and then illustrate how it can be used to characterize an HHPC application.

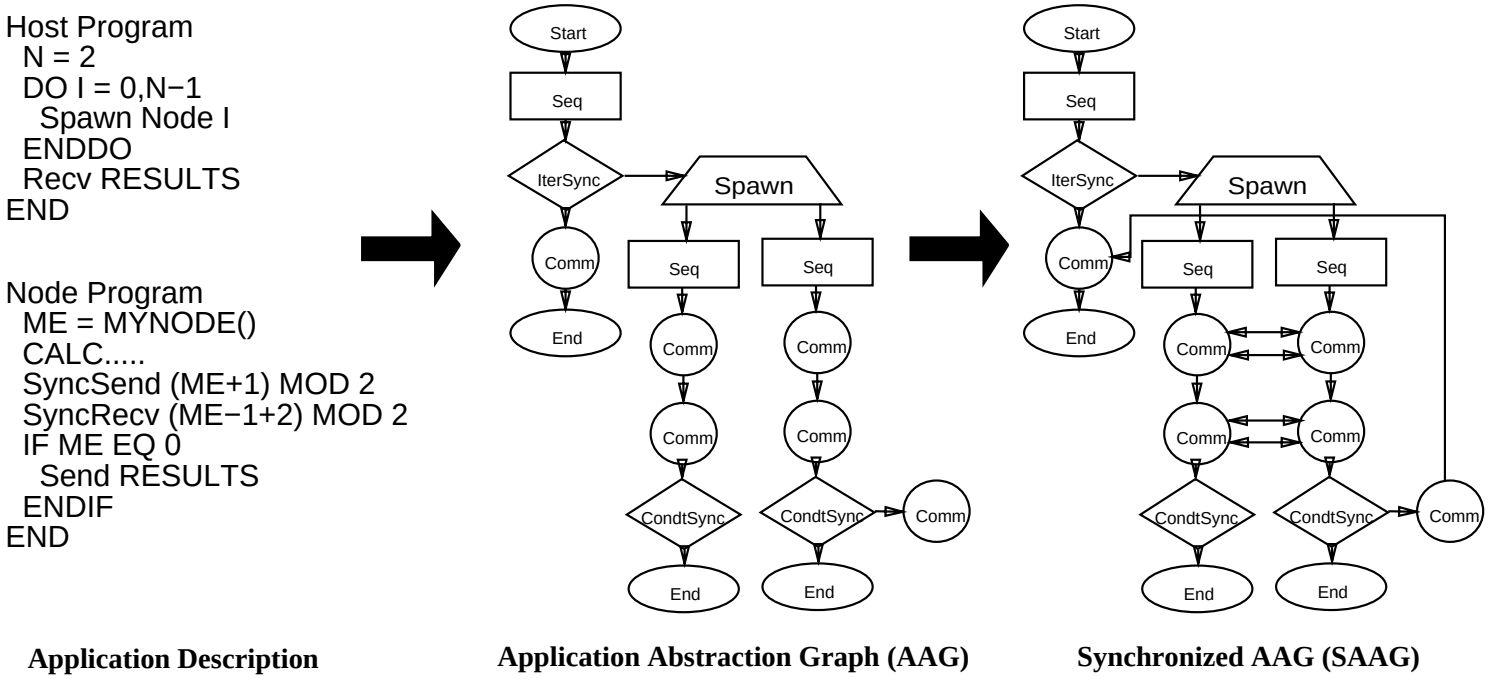


Figure 9: Application Abstraction Process

3.3.1 Application Abstraction Model

The application abstraction model recursively characterizes an application description into well defined *Application Abstraction Units (AAU)*. AAU's represent the fundamental unit of abstraction of the application description. An AAU can be of two types: A *Simple* AAU that cannot be further decomposed. It exports a set of well defined parameters which abstract the portion of the application description associated with it. A *Compound* AAU can be further decomposed, using similar techniques, into a set of simple or compound AAU's. The various classes of simple and compound AAU's are described in detail in the following sub-section. The AAU's are then combined to abstract the control structure of the application to produce the *Application Abstraction Graph (AAG)*. The AAG is simple directed graph defined as follows:

Definition 2 *An* **Application Abstraction Graph (AAG)** *is a simple directed graph such that*

$$AAG = [\{\mathbf{AAU}\}, \varphi]$$

$$where: \quad \forall AAU \in \{\mathbf{AAU}\};\ AAU \in \{\mathbf{Simple}, \mathbf{Compound}\}$$

$$and: \quad \mathbf{Simple} \equiv \{Start, End, Seq, Spawn, Comm, Sync, SyncOper\},$$

$$\mathbf{Compound} \equiv \{IterD, IterSync, IterND, CondtD, CondtSync, Call\}$$

where φ is an ordering relation defined such that $AAU_i\ \varphi\ AAU_j$; $AAU_i, AAU_j \in \{\mathbf{AAU}\}$ implies AAU_i precedes AAU_j in execution.

The communication/synchronization structure of the application is then superimposed onto the AAG by augmenting the graph with a set of edges defined by the following communication/synchronization relations:

$$\forall \quad AAU_i, AAU_j \in \{\mathbf{AAU}\};$$

1. $AAU_i \Leftrightarrow AAU_j$
   iff $AAU_i, AAU_j \in \{Comm, SyncSeq\}$
   and AAU_i & AAU_j *are of the same type,*
   and AAU_i & AAU_j *communicate with synchronization.*
2. $AAU_i \Rightarrow AAU_j$
   iff $AAU_i, AAU_j \in \{Comm\}$
   and AAU_i & AAU_j *communicate without synchronization.*
3. $AAU_i \rightleftharpoons AAU_j$
   iff $AAU_i, AAU_j \in \{Sync, SyncSeq\}$
   and AAU_i & AAU_j *are of the same type,*
   and AAU_i & AAU_j *synchronize.*

The structure generated after augmentation is called the *Synchronized Application Abstraction Graph (SAAG)*. Note that a compound AAU can be recursively decomposed into an AAG and an associated SAAG and that the compound AAU abstracts all sub-AAG's associated with it.

The SAAG is then passed through the machine dependent filter which uses the input mapping information to define a *Mapping Abstraction Function (μ)* from the SAAG to the SAG so as to assign:

1. Every $AAU_i \in SAAG$ to an $SAU_j \in SAG$ on which it is to be interpreted.
2. Every communication/synchronization edge (defined by relations $\Leftrightarrow, \Rightarrow, \&\ \rightleftharpoons$) in SAAG to an ordered set {**sau**} ({**sau**} $\subseteq$ {**SAU**}) which represents the actual route followed by the particular communication/synchronization for the specified mapping (e.g a communication from an external unit to a hypercube node has to be routed through the SRM).

The AAG and SAAG associated with a sample application description are shown in Figure 9. In what follows we describe the steps involved in the application abstraction procedure. The different classes of AAU's and the nature of the parameters exported by them are also described.

3.3.2 Application Characterization Methodology

The application characterization methodology is responsible for generating the application abstraction units and the abstraction graphs from the input application description using the application abstraction model defined above. This methodology uses parsing techniques to extract the required information and consists of 3 levels of parsing. The first level parsing generates the AAG. In addition, this level also extracts information about critical variables (defined in the following paragraphs), required external inputs, and creates an augmented symbol table. During second level parsing, the communication/synchronization structure of the application is superimposed onto the AAG to generate the SAAG. The global synchronization structure required during interpretation is created during this level. The final level of parsing is performed by the machine specific filter and is responsible for all machine specific augmentation to the

AAU Type	Sample Parameters
Start	(Not Applicable)
End	(Not Applicable)
Seq	NumFetch, NumStore, NumIntAdd, NumIntMul, NumFptMul, NumCalls, ListCalls[ ], ...
Spawn	NumChildren, Location[ ], *Children[ ] ...
IterD	NumIters, LimitType, *SeqBody, ...
IterSync	NumIters, LimitType, *SeqBody, *CommBody, *SeqBody, ...
IterND	LimitType, LimitExpr[ ], *IterBody, ...
CondtD	NumCondts, ListCondts[ ], *Bodies, ...
CondtSync	NumCondts, ListCondts[ ], *BodiesD, *BodiesSync, ...
Comm	Type, Size, Src, Dest, ...
Sync	Type, NumSyncNodes, SyncNode[ ], ...
SyncSeq	Type, NumSyncNodes, SyncNodes[ ], *SeqBody, ...
Call	Type, NumArgs, ArgList[ ], *CallBody, ...

Table 4: Application Characterization

SAAG. This level defines the mapping abstraction function (μ). In what follows, the three parsing levels are described in detail.

Level 1 Parse

The first level parses are responsible for characterizing each statement in the application description, clustering contiguous statements belonging to the same class and abstracting the clustered unit into an AAU. Note that the number of statements abstracted into a single AAU is controlled by the user. This allows the developer to dictate the granularity of abstraction and hence the granularity of the prediction. At the finest level, there is no clustering and performance measures are generated for each statement. The default is to generate the largest cluster possible. An AAU can be either simple or compound as defined earlier. The level 1 parse is applied recursively to compound AAU's to characterize, clusters and abstract them and to generate associated sub-AAG. The different classes of AAU's and the key parameters required to characterize them are summarized in Table 4. These classes are described below.

Start AAU (Start) The Start AAU marks the beginning of the application and the starting point for the interpreter engine. The global clock used by the engine is reset at this node. A structured application description (to which this framework is targeted) can have only one Start AAU and this AAU is the root of the associated abstraction graphs.

End AAU (END) The End AAU represents the termination of an independent flow of control (i.e. a subgraph of the AAG). For example, the termination of a node program would be represented by an End AAU. All leaf vertices of the AAG are End AAU's.

Sequential AAU (Seq) A Sequential AAU abstracts a set of contiguous statements which contain calls to library functions, system routines, assignments and/or arithmetic/logical operations.

Spawn AAU (Spawn) A Spawn AAU abstracts a "fork" type statement which initiates execution of a sub-graph of the AAG (possibly on another node). All measurements associated with the new execution are offset by the activation time (virtual time at which interpretation of the AAU begins as will be defined later) of the spawn AAU. The parameters required to abstract the Spawn AAU are listed in Table 4. In case of the iPSC/860, the "load" system call executed by the host processor belongs to this class.

Communication AAU (Comm) A communication AAU abstracts statements in the application description which involves explicit communication. This AAU covers synchronous/asynchronous sends and receives and point-to-point as well broadcast/multicast communication. In case of the iPSC/860, this includes calls to "csend/crecv", "isend/irecv", etc.

Synchronization AAU (Sync) The Synchronization AAU abstracts all statements involving explicit synchronization. This includes locks as well as barrier type synchronization.

Synchronized Sequential AAU (SyncSeq) This AAU abstracts any Seq AAU which requires synchronization or communication. This includes global arithmetic operations (add/mul), global logical operations (and/or) or global comparisons (max/min), as well writes to shared variable (in the case of AM-MIMD architectures), etc.

Iterative-Deterministic AAU (IterD) The Iterative-Deterministic AAU abstracts an iterative flow control structure (of the type DO-ENDDO) in the application description which has no descendants AAU's in it's body which are of type Spawn, Comm, Sync, or SyncSeq and whose execution order and number of execution are known during parsing (possibly in terms of external inputs). The IterD AAU has an AAG and SAAG associated with its body, and abstracts these sub-graphs.

Iterative-Synchronized AAU (IterSync) The Iterative-Synchronized AAU abstracts an iterative flow control structure (of the type DO-ENDDO) in the application description whose execution order and number of execution are known during parsing (possibly in terms of external inputs), but has at least one descendants AAU's in it's body which is of type Spawn, Comm, Sync, or SyncSeq.

Iterative-NonDeterministic (IterND) An Iterative Non-Deterministic AAU is a compound AAU and has one or both of the following characteristics: (1) The body of the iterative structure is dependent on the iteration. (2) The number of iteration depends on a parameter defined within the body of the iterative structure. The performance of this type of an iterative node is estimated either by un-rolling the iterative structure or by using user defined heuristics to resolve the non-determency.

Conditional-Deterministic (CondtD) The Conditional-Deterministic AAU abstracts a conditional flow control structure (of the type IF-THEN-ELSE) in the application description which has no AAU's in the sub-AAG' associated with its bodies, which belong to the set {Spawn,Comm,Sync,SyncSeq}. The Conditional-Deterministic AAU is compound.

Conditional-Synchronized (CondtSync) A Conditional-Synchronized AAU abstracts a conditional flow control structure (of the type IF-THEN-ELSE) in the application description which has at least one AAU in the sub-AAG' associated with its bodies which belong to the set {Spawn,Comm,Sync,SyncSeq}. The Conditional-Synchronized AAU is compound.

Call AAU The Call AAU abstracts all invocations of user-defined functions or subroutines. This AAU is compound and abstracts information about the nature of the call and the number of arguments passed. It also abstracts the sub-AAG's associated with the subroutine/function body.

In addition to generating the AAG, a list of critical variables is also created during this pass. A critical variable is defined as a variable whose value influences the flow of execution. This includes index variables of an iterative AAU, or the variables used to define conditions in a conditional AAU. For each critical variable, a definition path is created and the AAU where it is defined is tagged. If a critical variable is an external input, it is accordingly tagged. The user is then prompted for its value.

The third task of the level 1 parse is to augment the regular symbol table to store information about the access pattern in order to model accesses to the memory hierarchy. The information stored is the number of accesses between consecutive accesses to a particular variable.

Level 2 Parse

The parses at level 2 are responsible for abstracting the communication and synchronization structure of the application and superimposing it on top of the AAG to generate the SAAG. This is done by adding three classes of edges between AAU sets of type Comm, Sync and SyncSeq. These edges correspond to the relations $\Leftrightarrow, \Rightarrow, \& \rightleftharpoons$ defined in the abstraction model. In addition, each communication/synchronization occurrence in the application is assigned a unique *id* and the associated set of Comm, Sync or SyncSeq nodes are tagged with that *id*. A global synchronization structure is created which is indexed by this *id* and contains contains information about the type of communication/synchronization (sync/async, pt-to-pt/bcast/mcast, barrier, lock, etc.). The list of communicating AAU's, and other parameters associated with the particular communication/synchronization type are also extracted and stored in this structure.

Using the SAAG and the global synchronization structure, the level 2 parses can detect error in the application description like mismatched communication/synchronization calls, missing calls, potential deadlocks, etc.

Level 3 Parse

The level 3 parse is performed by the machine specific filter and is responsible for generating the mapping abstraction function μ. It also augments the global synchronization structure to introduce machine specific details about the communications/synchronization. For example, in the iPSC/860, a broadcast (using the NX/2 communication library) is implemented using a logarithmic algorithm; a broadcast call is thus replaced by the corresponding set of point-to-point messages defined by this algorithm. Similarly, the number of hops required for a message from AAU_i to AAU_j using the particular routing algorithm (e-cube in the iPSC/860) is calculated. Another characteristic specific to the iPSC/860 which is handled by the level 3 parses is tagging all synchronous messages greater that 100 bytes in length as requiring dynamically allocated buffers. This level also introduces compiler transformations/optimizations specific to the machine. Overlaps between communications and computations are tagged so that they can be accounted for during interpretation.

3.4 Interpretive Engine

The interpreter engine is responsible for the actual performance interpretation. It uses the system, application and mapping abstractions, to predict the performance of the abstracted application. In what follows we first define the interpretation model and algorithm. The methodology used to interpret specific types of AAU's is then described. Finally we briefly describe how it can handle experimentation and "What-if?" scenarios.

3.4.1 Interpretation Model

The interpretation model consists of two components: (1) An *Interpretation Function (Ω)* that interprets the performance of an individual AAU and (2) An interpretation algorithm which recursively applies the interpretation function to the SAAG to predict the performance of the corresponding application.

Definition 3 *The* **Interpretation Function** (Ω) *operates from the set* { **AAG** } *to the set* $\Re$ *of real numbers and assigns to each* $AAU \in SAAG$ *a subset of* $\Re$ *which represents the performance statistics of that AAU. i.e.*

$$\Omega(AAU_i, SAG, \mu) : AAU_i \mapsto \{\mathbf{R}\}; \quad where\ AAU_i \in \{\mathbf{AAU}\}\ and\ \{\mathbf{R}\} \subset \Re$$

where μ is the mapping abstraction function.

Before we state the algorithm, the following terminology needs to be defined: A *Chain* of an SAAG is defined as a set of contiguous AAU's in that SAAG. The first AAU is called the *Head* of the chain. *Evaluating* an AAU consists of applying the interpretation function Ω to the AAU to obtain its performance statistics. A *Red* AAU denotes an AAU that has been evaluated. An *Active* AAU is an AAU whose immediate predecessors are red AAU's. An *Active Chain* is a chain whose head is active.

Let τ_{AAU_i} denote the time (measured from the start of the application) at which AAU_i became active. τ_{Start} represents the beginning of the application execution and the reference point for all measurements

made by the interpretation model. Let δ_{AAU_i} denote the execution time of AAU_i returned by the interpretation function Ω. The interpretation algorithm can now be defined as follows:

Algorithm 2 *Interpret*

(1) color **Start** AAU red;
$\tau_{Start} \leftarrow 0$;
$\delta_{Start} \leftarrow \Omega(AAU_{Start}, SAG, \mu)$ [evaluate AAU_{Start}]
(2) for each active AAU
repeat until there is no active AAU
  • for each AAU (AAU_i) in the associated active chain
  repeat until AAU_i cannot be evaluated due to synchronization requirements
    $\tau_{AAU_i} \leftarrow \tau_{AAU_{i-1}} + \delta_{i-1}$
    $\delta_i \leftarrow \Omega(AAU_i, SAG, \mu)$ [evaluate AAU_i]
    color AAU_i red
  end repeat
end repeat
(3) if all leaf AAU's of the SAAG are not red
  ERROR
end if

End

The above algorithm proceeds down each active chain (i.e. depth first) in the SAAG, and evaluates each AAU of the active chain. It also updates a global time base (τ) as it proceeds. If the current AAU cannot be evaluated because it has to wait for synchronization (i.e. it is a Comm, Sync or SyncSeq AAU), that AAU remains active and the algorithm moves to the next active chain. If at the end of the algorithm, the leaf AAU's of the (topmost) SAAG are not red, an error has occurred in the implementation of the application and has caused it to hang-up. The application of the interpretation algorithm to a sample SAAG is shown in Figure 10. The interpretation methodology used to handle various classes of AAU's is described below.

3.4.2 Interpretation Methodology

In this section we briefly describe the models used to handle accesses to the memory hierarchy, communication and synchronization between computing units, external input variables, and how these models are used to interpret the performance of each AAU type. The experimentation with different run-time situations and "What-if?" scenarios.

Handling External Inputs External inputs or unknown variables encountered in the application description are tagged during the level 1 parses. If the variables are critical, the interpreter engine prompts the user for their values. Otherwise, an attempt is made to interpret the performance of the application as a function of these variables and if this is not possible, the user is prompted for the values.

Figure 10: Application of the Interpretation Algorithm

Modeling Access to the Memory Hierarchy Access to the memory hierarchy of a computing element is modeled using heuristics based on the access patterns in the application description and the physical structure of the hierarchy. In the prototype implementation, the symbol table is augmented to store additional information about access patterns. During the level 1 parses of the application description, the number of unique accesses between successive accesses to a particular variable is stored. A working set window is then defined based on the physical structure of the cache. The access time for a variable is then defined as a weighted sum of the times required to access the two levels of the memory hierarchy. The weights are determined on the basis of the size of the working window and the information stored in the augmented symbol table.

A more detailed memory access model can be developed by creating a memory map of the application from the application description. This map, along with the specifications of the cache can then be used to to emulate the state of the cache at the time of each access. If the access distribution for the application is known, a stochastic model can be used.

Modeling Communication-Computation Overlaps The amount of overlap between communication and computation depends on the capability of the particular computing unit and is tagged by the machine specific filter during level 3 parsing. This overlap is accounted for during interpretation as a fraction of the communication cost; i.e. if a communication takes time t_{comm} and $f_{overlap}$ is the fraction of this time overlapped with computation, then the execution time of the Comm AAU is weighted by the factor $(1 - f_{overlap})$; i.e.

$$t_{AAU_{Comm}} = (1 - f_{overlap}) \times t_{comm}$$

Modeling Communication/Synchronization Communication or synchronization operations in the application are decomposed during interpretation into three components: viz. (1) call overhead, (2) transmission time, and (3) waiting time. The call overhead represents the fixed overheads associated with the operation. At the source of the operation, this overhead includes the startup cost associated with copying and packing the message and initiating transmission. At the destination, this overhead includes the time required to receive and unpack the message and to copy it to the user space. The transmission time is the time required to actually transmit the message from the source to the destination (via the defined path). The waiting time models the synchronization overhead and is defined as the part of the total execution time of the operation which is not due to startup or actual transmission. This could be the overhead caused by unavailable links in the communication path, by a non-ready receiver in case of rendezvous communication or by unavailable buffers in case of buffered communication. The waiting time is computed using the global synchronization structure. This structure is continually updated during interpretation to keep track of the state of each communication/synchronization operation. The exact order and times of execution of AAU's associated with each operation is maintained. If synchronization is required, this information specifies the number of units ready to synchronize or whether or not the barrier count has been met. In case of asynchronous messages, it keeps track of global time at which the message is transmitted.

The transmission time represents the time required to actually transmit the message from the source to the destination. The model of communication/synchronization operations is illustrated in Figure 11. Note

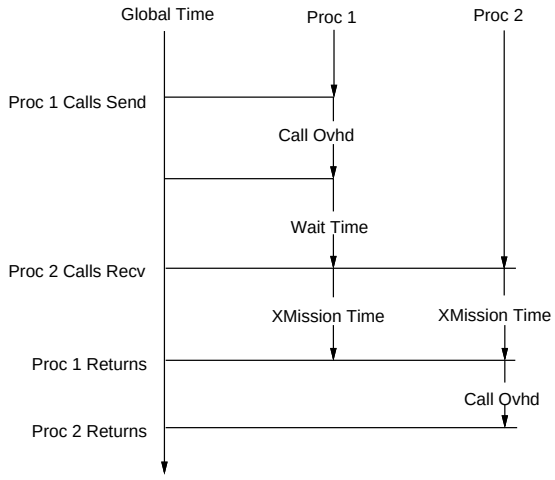


Figure 11: Interpretation Model for Communication/Synchronization AAU's

that any of the above components could be null depending on the type of communication and on whether the unit is the source or destination. A null component implies that the component does not contribute to the execution time of the AAU. For example, in case of asynchronous communication, the sender's waiting time and transmission time components are null since they do contribute to the execution time of the corresponding AAU.

Interpretation of Comm, Sync & SyncSeq AAU's The interpretation of Comm, Sync or SyncSeq AAU's depends on the type of communication/synchronization required. The different cases are described below.

(1) In case of asynchronous communication, the sender's call overhead represents the time between the instant the call is made to the instant it returns. Transmission time at the sender's end is null. This time however is stored in the global synchronization structure to compute the time the message is ready to be received at the destination. The waiting time is also null for the sender.

(2) In case of buffered communication using static buffers, the sender's call overhead and transmission time are computed as described in the model above. However, the waiting time for the sender is zero if a buffer is available; else it is the time until a buffer is freed.

(3) In case of buffered communication using dynamically allocated buffers, the sender's waiting time is the time from when a request is sent until the time the required buffers are allocated. The other two components are computed as described by the model above.

For all the above cases, the three components at the receiver end are not null and are computed as defined by the model described above. Similarly, for unbuffered communication, both sender and receiver components are computed as defined by the model.

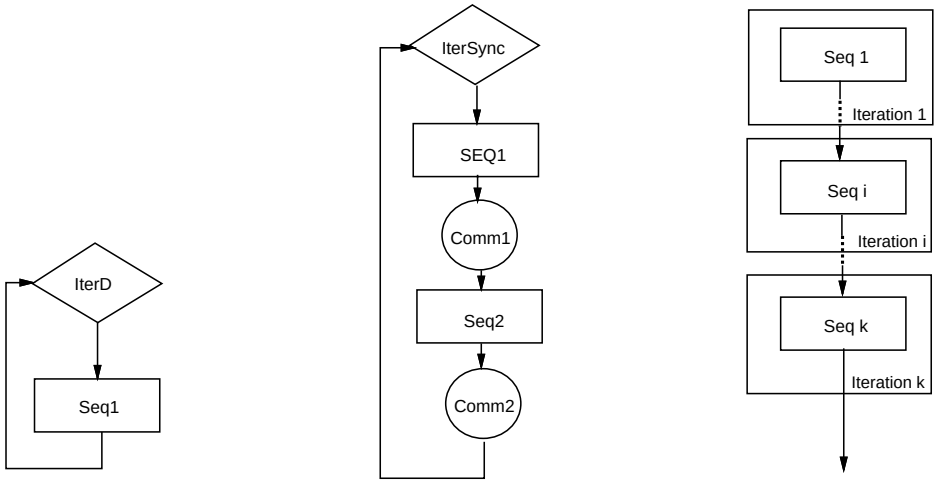
(4) In case of Sync AAU's, the call overhead and wait time components are computed according to the model described above. The transmission time depends on the nature of synchronization. If a synchronization signal is used, this time is null; else if a message is used, the transmission time is the time to transmit this message.

(5) For a SyncSeq AAU, there is an additional component which represents the time to perform the Seq operation. Further, in this type of AAU, the number of communications and their order depends on the algorithm used.

(6) Broadcasts and multicasts are treated as multiple messages whose order and paths again, depend on the algorithm used.

Our prototype system models asynchronous and buffered/unbuffered synchronous communication on the iPSC/860. Synchronous messages less than 100 bytes in length are automatically buffered. Each node has about 1000 buffers of 100 bytes each. Messages larger than 100 bytes require dynamic buffers to be allocated. On this system broadcasts, multicasts and SyncSeq operation use a logarithmic algorithm.

Modeling of Iterative Flow-Control Structures The execution of an iterative flow control structure is broken up into three components: (1) the startup overhead associated with setting up the structure and the additional overhead associated with the last iteration, (2) the overhead associated with each loop iteration, (3) the execution cost of the iteration body. The two overheads components (1 & 2) are a function of the type of iterative structures and the nature of its iteration limits and stride. The per-iteration overhead has two parts: the overhead at the start of the iteration body and that at the end. The abstraction of the three types of iterative AAU's are shown in Figure 12. Their interpretation is described below.



(a) IterD AAU **(b) IterSync AAU** **(c) IterND AAU**

Figure 12: Abstracted Iterative AAU's

Interpretation of IterD AAU's The abstracted IterD AAU is shown in Figure 12(a). The body of this AAU can be abstracted into a single Seq AAU since its execution is deterministic and the activation times of the individual AAU's within this body is not required (no Spawn, Comm, Sync, or SyncSeq AAU's in the body). The individual AAU's in the body are abstracted (using techniques described in this section) into their respective interpretation functions. The interpretation function for the entire body can the be defined as a combination of the individual interpretive functions. This interpretation function may be expressed in terms of external inputs. The function is then assigned to a virtual operation which is abstracted as a Seq AAU. Since the number of iteration for an IterD AAU is known during parsing, the interpretation function for the entire structure can now be defined as the linear combination of the fixed overhead, the per-iteration overhead and the and the execution time of the virtual operation; i.e.

$$\delta_{IterD} = t_{FixedOvhd} + NumIters \times [t_{PerIterOvhd} + \delta_{VirtualOper}]$$

Interpretation of IterSync AAU's The abstracted IterSync AAU is shown in Figure 12(b). All contiguous AAU's in the iteration body which are not of type Spawn, Comm, Sync, or SyncSeq are abstracted into a Seq AAU as described above and are assigned a virtual interpretation function. The interpretation function for the entire AAU is then defined as a recursive equation such that the the execution time of the current iteration is a function of the execution time of the previous iteration. Similarly, the activation and execution times of the Spawn, Comm, Sync, or SyncSeq AAU's in the iteration body can be defined in terms of the execution time of the previous iteration. For example (see Figure 12(b)), let the activation time of the IterD AAU (AAU_{IterD}) be T; then:

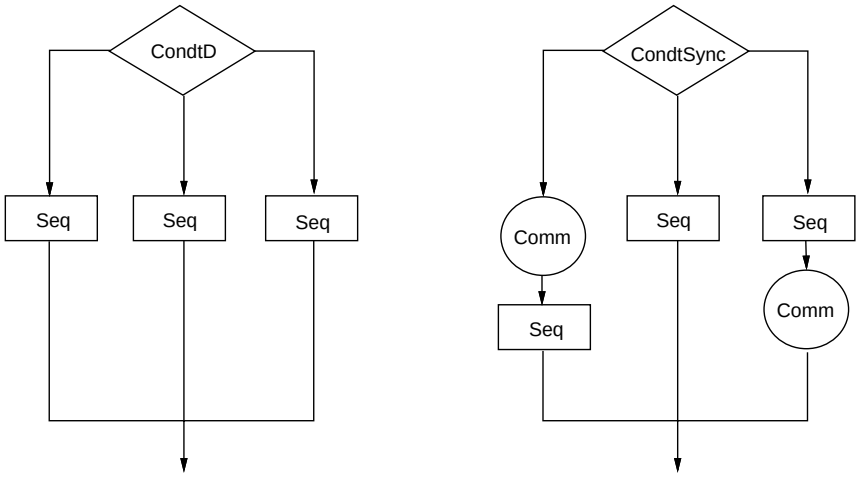
$$\begin{aligned}
\tau_{IterD}(1) &= T \\
\tau_{Comm_1}(1) &= \tau_{IterD}(1) + overhead + \delta_{Seq_1} \\
\tau_{Comm_2}(1) &= \tau_{IterD}(1) + overhead + \delta_{Seq_1} + \delta_{Comm_1}(1) + \delta_{Seq_2}
\end{aligned}$$

And for the i^{th} iteration

$$\begin{aligned}
\tau_{IterD}(i) &= \tau_{IterD}(i-1) + overhead + \delta_{Seq_1} + \delta_{Comm_1}(i-1) + \delta_{Seq_2} + \delta_{Comm_1}(i-1) \\
\tau_{Comm_1}(i) &= \tau_{IterD}(i) + overhead + \delta_{Seq_1} \\
\tau_{Comm_2}(i) &= \tau_{IterD}(i) + overhead + \delta_{Seq_1} + \delta_{Comm_1}(i) + \delta_{Seq_2}
\end{aligned}$$

Interpretation of IterND AAU's The abstracted IterSync AAU is shown in Figure 12(c). This type of iterative structure is interpreted by unrolling, i.e each iteration is interpreted sequentially. The user can however define heuristics to resolve the non-determency and to model this type of AAU.

Modeling of Conditional Flow-Control Structures The execution time for a conditional flow control structure is broken down into three components: (1) the overhead associated with each condition tested (i.e. every "if", "elseif", "else", etc.), (2) an additional overhead for a true condition, and (3) the time required to execute the body associated with the true condition. The interpretation function for the conditional AAU itself is weighted sum of the interpretation functions associated with the bodies of the



(a) CondtD AAU **(b) CondtSync AAU**

Figure 13: Abstracted Conditional AAU's

structure. The weights assigned to a particular interpretation function is the logical expression that must be evaluated before the its body is executed. The expression evaluates to 1 if the conditions are true and to zero otherwise. For example consider the iterative structure below:

Algorithm 3 *CondtD*

if C_1 and C_2 then
    Seq_1
elseif C_3
    Seq_2
else
    Seq_2
end if

End

where C_1, C_2 and C_3 are the logical condition that evaluate to 1 or 0. Let t_{Ovhd1} be the overhead associated with each condition and t_{Ovhd2} be the the overhead associated with a true condition. The associated interpretation function is defined as follows:

$$\begin{aligned}\Omega_{IterD} &= [C_1C_2] \times [\Omega_{Seq_1} + 2t_{Ovhd1} + t_{Ovhd2}] \\ &+ [(1-C_1)(1-C_2)C_3] \times [\Omega_{Seq_2} + 3t_{Ovhd1} + t_{Ovhd2}] \\ &+ [(1-C_1)(1-C_2)(1-C_3)] \times [\Omega_{Seq_3} + 4t_{Ovhd1} + t_{Ovhd2}]\end{aligned}$$

The abstraction of the two types of conditional AAU's are shown in Figure 13. Their interpretation is described below.

Interpretation of CondtD AAU's The abstraction of the CondtD AAU is shown in Figure 13(a). The abstraction of the bodies follows the same procedure as described for the IterD AAU.

Interpretation of CondtSync AAU's The abstraction of the CondtSync AAU is shown in Figure 13(b). The bodies of this structure which have one or more AAU's of the type Spawn, Comm, Sync or SyncSeq, are interpreted using the recursive technique described above for the IterSync AAU. The other bodies can be evaluated in the same way as the IterD AAU.

Interpretation of Call AAU's A call to a user-defined function or subroutine is handled by an inline expansion of that function/subroutine following the Call AAU. Normal characterization, abstraction and interpretation techniques described in this papers are applied to the function/subroutine body. Another alternative available to the user is to specify the interpretation function corresponding to the function/subroutine call. The interpretation algorithm then, uses this specified function during interpretation.

Interpretation of Start, End AAU's Interpretation of Start/End AAU's mainly consists of administrative operations. The Start AAU requires the virtual timer to be reset. The End AAU indicates that the particular chain has been completely interpreted and the engine can now move to the next active chain.

Interpretation of Seq AAU's The interpretation function for the Seq AAU's is linear combination of the number of operations of different types weighted by the time required to execute these operations on the particular SAU.

Interpretation of Spawn AAU's Interpretation of the Spawn AAU consists of activating the initiated sub-AAG's and initializing their timer to the execution time of the Spawn AAU. Its interpretation function returns the time required to complete the initiation.

Modeling Runtime Parameters The effect of run-time parameters are incorporated into the interpretation model by abstracting their effects on the various parameters exported by the systems and application modules. Heuristics are used to perform this abstraction. For example, the effect of increased network load on a particular communication channel is modeled by decreasing the effective available bandwidth on that channel. An appropriate scaling factor is then defined which is used to scale the parameters exported by the C/S component associated with the communication channel.

Modeling "What-if's ?" The interpreter engine provides the user with an interactive interface through which the user can experiment with different scenarios. These scenario include increasing communication bandwidths of a particular channel, increasing the number of nodes in a computing unit or the processing capacity of each node, etc. The scenarios are modeled by abstracting them into scaling factors which are then used to scale appropriate parameters exported by the systems and applications modules.

3.5 Output Module

The output module provides an interactive interface through which the user can access the interpreted performance statistics. The user has the option of selecting the type of information, the level at which the information is to be displayed. Performance statistics can be obtained at the following levels:

- **AAG Level** Performance information at the AAG level deals with the entire application. Statistics available at this level include cumulative execution times, the communication time/computation time breakup and the existing idle times.
- **Sub-AAG Level** Performance information at this level deals with the specified part of the AAG. Cumulative statistics for the specified subgraph are displayed.
- **AAU Level** Performance information at this level is specific to a particular AAU. All statistics relevant to that AAU are displayed.

Visualization software can be interfaced to this module to provide graphical displays of the available information. Animation capabilities can also be incorporated. We are currently working on interfacing the outputs produced by the prototype system, with the ParaGraph [46] visualization package which provides the above mentioned capabilities.

4 Numerical Results

This section presents some preliminary numerical results obtained through experimentation on a prototype performance prediction framework. The objective of this experiment was threefold:

1. To validate the system and application abstraction model and to demonstrate their feasibility and applicability.
2. To validate the performance interpretation model proposed.
3. To demonstrate the cost-effectiveness of the approach in terms of both, resources required and time taken.

To meet the first objective we chose an architecture which is widely used to solve scientific and engineering applications. The computing system used consisted of an iPSC/860 hypercube connected to a 80386 based host processor. The particular configuration of the iPSC/860 consists of 16 i860 nodes. Each node has a 4 KByte instruction cache, 8 KByte data cache and 8 MBytes of main memory. The node operates at a clock speed of 40 MHz and has a theoretical peak performance of 80 MFLOPS for single precision and 40 MFLOPS for double precision.

The application chosen was part of a standard benchmark set (The Purdue Benchmark Set [47]) and were written using FORTRAN 77 and the NX/2 communication libraries. The implementation was tweaked to

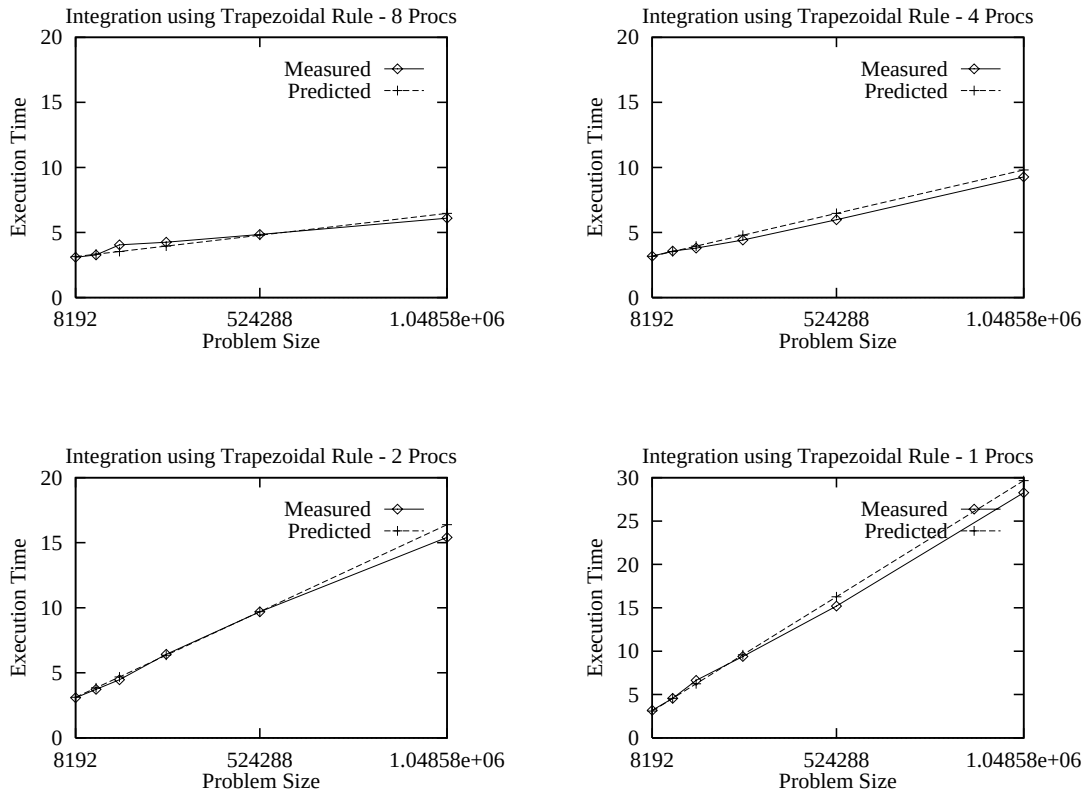


Figure 14: Comparison of Predicted and Measured Times (sec)) - Application: Integration using Trapezoidal Rule

incorporate a wide range of programming constructs. The chosen application evaluates the integral, T_N, of $f(x)$ using the trapezoidal rule.

$$T_N = h * (f(a)/2 + \sum_{i=1}^{N-1} f(a + ih) + f(b)/2)$$

The implementation uses the host-node programming model wherein the host program allocates the node processors and loads the node programs. It then uses cyclic distribution to distribute the integration domain among the nodes and broadcasts integration parameters. The host program receives the integral from the nodes after completion. The node processors calculate the integral over their domains and then perform a global sum. Node zero then sends the results to the host. The above procedure is repeated multiple times in a loop. The number of intervals into which the integration domain was divided was an external input. The number iteration were provided as external inputs.

The experimentation consisted of varying two variables: the external input (Problem Size) and the number of processing nodes used. The time on the host from the instant the node program was loaded till the final result was received, was measured. The experimentation was performed in two phases:

Phase 1 This phase consisted of implementing the application and then running it for each combination of problem size and number of processors. The implementation was instrumented to measure execution times. Multiple runs were made for each case to account for noise in the measured timings.

Phase2 This phase consisted of abstracting the application and feeding it to the performance prediction prototype. Then, using the interactive interpreter engine, prediction were obtained for all the desired combinations. A comparison between the measured and predicted times (in seconds) are plotted in Figure 14. The results obtain show that the predicted values lie within 15% of the measured results. This meets our second objective. The cost-effectiveness of the interpretive approach is obvious from the fact the entire experiment was completed in a single run and on a Sun workstation.

5 Summary & Concluding Remarks

Evaluation tools form a critical part of any software development environment and enable the developer to visualize the effects of various design choices on the performance of the application, to study the scalability of the application with system and problem size and to investigate the effects of changes in system run-time status and its configuration on the application execution. Further, these tools must provide information about the contribution of various factors effecting the performance of the application. Finally, there is a need for a symbiotic relationship between the evaluation tools and other development tools (such as mapping, analysis, and optimizing tools) so as to complete the feedback loop of the "develop-evaluate-tune" cycle. Conventional evaluation tools and techniques are either tuned to specific systems and cannot incorporate the heterogeneity and dynamic nature of an HHPC environment or are too general and lack feasibility and accuracy needed in HHPC software development.

In this paper we presented the design of a performance prediction framework which uses a novel interpretive approach to to provide accurate and cost-effective performance prediction. A comprehensive system abstraction model was defined which provides a methodology to characterize any heterogeneous computing environment. A corresponding application abstraction model is also defined which can be used to characterize the structure of any structured application. Finally an interpretation model is defined which uses the system and application abstraction to achieve performance interpretation. The application of the proposed interpretation model to abstract accesses to memory hierarchy, communication and synchronization, overlap between computation and communication, interpret the performance of programming structures is shown. In addition, the ability of the model to handle different run-time and "what-if?" scenarios is demonstrated. Finally, interpreted performance of a standard parallel benchmark on a widely used distributed memory architecture is compared to the measured performance. The results obtained not only validated the accuracy and feasibility of the abstraction, and interpretation model, but also demonstrated its cost-effectiveness, both in terms of resources required and time taken. The key features of the approach can be summarized as follows:

Global Ordering The proposed performance prediction framework introduces a global ordering during interpretation, allowing it to overcome the inherent lack of synchrony that exists in an HHPC environment. This enables communication and synchronization overheads to be accurately modeled.

Flexibility The proposed approach enables the user to use the optimal technique (analytic, monitoring, specifications, etc) to generate parameters required to characterize individual components in the environment. This not only allow it to capitalize on existing technologies but also enables test architectures and configuration to be evaluated before actual implementation. Further, this approach supports the current trends towards the standardization of hardware components. Standard models for these components can be developed and then reused as building blocks in models for various architectures.

Cost-effectiveness The proposed performance prediction framework does not require the actual hardware to be present during evaluation. This makes it cost-effective, both, in terms of resources needed and time and effort required for the evaluation

Utility The proposed framework provides the capability of evaluating the effects of different runtime and "what-if ?"scenarios. The effect of changes in system state like the occurrence of faults and the gradual degradation of the system can also be evaluated. Since the approach is source driven, it can be automated and incorporated into an intelligent compiler.

Usability The proposed framework does not require the actual hardware to be present during evaluation and can be executed from within a friendly workstation based graphical interface. Further graphical capabilities of the workstation can be used to display the performance measures.

Current work in on this project is progressing in two direction. The first direction consists of extending the prototype to a heterogeneous network-based environment, and tuning the models and heuristics used. A model for the I/O components is also being developed. Concurrently we are also incorporating an interpretive performance prediction tool into the High Performance Compiler being developed at the Northeast Parallel Architectures Center, Syracuse University. This tool will assist the compiler to optimize data-distribution.

Acknowledgment

The presented research has been jointly sponsored by DARPA under contract #DABT63-91-k-0005 and by Rome Labs under contract #F30602-92-C-0150. The content of the information does not necessary reflect the position or the policy of the sponsors and no official endorsement should be inferred.

References

[1] Glenn Zorpette, "Teraflops Galore", *IEEE Spectrum*, vol. 29, pp. 26–76, sep 1992.

[2] Salim Hariri, Manish Parashar, Jong Baek Park, Fang-Kuo Yu, and Geoffrey Fox, "A Case for Heterogeneous High Performance Computing", Technical Report SCCS-417, Northeast Parallel Architectures Center, 111 College Place, Room # 3-201, Syracuse NY 13244-4100, 1992.

[3] Salim Hariri, Manish Parashar, JongBaek Park, and Fang-Kuo Yu, "An Environment for High-Performance Distributed Computing", Technical Report SCCS-418, Northeast Parallel Architectures Center, Syracuse University, 111 College Place Room # 3-201, Syracuse NY 13244-4100, 1992.

[4] Salim Hariri, Manish Parashar, JongBaek Park, Fang-Kuo Yu, and Geoffrey Fox, "A Message Passing Interface for Parallel and Distributed Computing", To be presented at the 2nd International Symposium on High Performance Distributed Computing, Spokane, Washington, July 1993.

[5] Richard F. Freund and D. Sunny Conwell, "Superconcurrency: A Form of Distributed Heterogeneous Supercomputing", *Supercomputing Review*, vol. , pp. 47–50, oct 1990.

[6] Norris Parker Smith, "The Future of High-Performance Computing: The 1990 Federal Assessment", *Supercomputing Review*, vol. , pp. 52–53, oct 1990.

[7] Gordon Bell, "Ultracomputers: A Teraflop Before Its Time", *Communications of the ACM*, vol. 35, pp. 27–47, aug 1992.

[8] Ashfaq Khokar, Viktor K. Prasanna, Mohammad Shaaban, and Cho-Li Wang, "Heterogeneous Supercomputing: Problems and Issues", *Heterogeneous Processing Workshop, IPPS '92*, vol. , 1992.

[9] Horace P. Flatt and Ken Kennedy, "Performance of Parallel Processors", *Parallel Computing*, vol. 12, pp. 1–20, Oct. 1989.

[10] A. Kapelnikov, R. R. Muntz, and M. D. Ercegovac, "A Methodology for Performance Analysis of Parallel Computations with Looping Constructs", *Journal of Parallel and Distributed Computing*, vol. 14, pp. 105–120, 1992.

[11] Reda A. Ammar and Bin Qin, "A Technique to Derive the Detailed Time Costs of Parallel Computations", *Proceedings of the* 12^{th} *Annual International Computer Software and Application Conference*, vol. , pp. 113–119, 1988.

[12] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System", *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 206–217, Apr. 1990.

[13] Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Thomas Treml, and Roland Wismüller, *The Design and Implementation of TOPSYS*, Technische Universität München, Institut Für Informatik, July 1991, Version 1.0.

[14] Markus Siegle and Richard Hofmann, "Monitoring Program Behaviour on SUPRENUM", *ACM SIGARCH, Proceedings of the* 19^{th} *International Symposium on Computer Architecture*, vol. , pp. 332–341, May 1992.

[15] James R Larcus and Thomas Ball, "Rewriting Executable Files to Measure Program Behavior", Technical Report 1083, Computer Science Department, University of Wisconsin-Madison, v1210 West Dayton Street, Madison WI 53706, Mar. 1992.

[16] Giovanni Chiola, Marco Ajmone Marsan, and Gianfranco Balbo, "Product-Form Solution Techniques for the Performance Analysis of Multiple-Bus Multiprocessor Systems with Nonuniform Memory References", *IEEE Transactions on Computers*, vol. 37, pp. 532–540, May 1988.

[17] Daniel A. Menascé and Luiz Andreé Barroso, "A Methodology for Perforamce Evaluation of Parallel Applications on Multiprocessors", *Journal of Parallel and Distributed Computing*, vol. 14, pp. 1–14, 1992.

[18] Philip Heildelberger and Kishore S. Trivedi, "Analytic Queueing Models for Programs with Internal Concurrency", *IEEE Transactions on Computers*, vol. C-32, pp. 73–82, Jan. 1983.

[19] J. T. Robinson, "Some Analysis Techniques for Asynchronous Multiprocessor Algorithms", *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 24–31, Jan. 1979.

[20] Kijoon Chae and Arne A. Nilsson, "Performance Evaluation of FDDI Network and Interconnection Heterogeneous Networks", IEEE - 0742-1303/90/0000/0075 pp 75-83, 1990.

[21] Daniel A. Reed and Dirk C. Grunwald, "The Performance of Multicomputer Interconnection Networks", Technical report, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, .

[22] S. Balsamo and L. Donatiello, "Approximate Performance Analysis of Parallel Processing Systems", in M. Cosnard and C. Girault, editors, *Decentralized Systems*, pp. 325–336. Elsevier Science Publishers B. V. (North-Holland), 1990.

[23] Lishing Liu and Jih-Kwon Peir, "A Peformance Evaluation Methodlogy for Coupled Multiple Supercomputers", *Proceedings of the 1990 International Conference on Parallel Processing*, vol. , pp. 198–202, Aug. 1990.

[24] Dalibor F. Vrsalovic, Daniel P. Siewiorek, Zary Z. Segall, and Edward F. Gehringer, "Performance Prediction and Calibration for a Class of Multiprocessors", *IEEE Transactions on Computers*, vol. 37, pp. 1353–1365, Nov. 1988.

[25] Walid Abu-Sufah and Alex Y. Kwok, "Performance Prediction Tools for CEDAR: A Multiprocessor Supercomputer", IEEE 1985 (0149-7111/85/0000/0406$01.00), 1985, pp 406-413.

[26] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice Parallel Processing Testbed", 1988 ACM 0-89791-254-3/88/0005/0004 pp 4-11, 1988.

[27] F. Andre and A. Joubert, "SiGle: An Evaluation Tool for Distributed Systems", *Proceedings of the International Conference on Distributed Computing Systems*, vol. , pp. 466–472, 1987.

[28] F. Lange, R. Kroeger, and M. Gergeleit, "JEWEL: Design and Implementation of a Distributed Measurement System", *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 657–671, Nov. 1992.

[29] P. Dauphin, R. Hoffman, R. Klar, B. Mohr, A. Quick, M. Siegle, and F. Sötz, "ZM4/SIMPLE: A Genral Approach to Performance-Measurement and -Evaluation of Distributed Systems", in T. L. Casavant and M. Singhal, editors, *Advances in Distributed Computing: Concepts and Design*. IEEE Computer Society Press, 1992.

[30] Dieter Wybranietz and Dieter Haban, "Monitoring and Performance if Distributed Systems During Operation", *Proceeedings of the 1988 ACM SIGMETRICS Conference; Measurement and Modeling of Computer Systems, Sante Fe, NM*, vol. , pp. 197–206, 1988.

[31] Alan Sussman, "Execution Models for Mapping Programs onto Distributed Memory Parallel Computers", Technical Report 189613, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Virginia 23665-5225, Mar. 1992.

[32] Vasant Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer, "A Static Performance Estimator in the Fortran D Programming System", in Joel Saltz and Piyush Mehrotra, editors, *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, pp. 119–138. Elsevier Science Publishers B.V., 1992.

[33] Vasanth Balasundaram, Ken Kennedy, Ulrich Kremer, K. McKinley, and J Subhlok, "The ParaScope Editor: An Interactive Parallel Programming Tool", *Supercomputing '89, Reno, Nevada*, vol. , Nov. 1989.

[34] Jr. V. A. Guarna, D. Gannon, Y. Gaur, and D. Jablonowski, "FAUST: An Environment for Programming Parallel Scientific Applications", *Supercomputing 1988*, vol. , pp. 3 – 10, 1988.

[35] Daya Atapattu and Dennis Gannon, "Building Analytic Models into an Interactive Performance Prediction Tool", *Proceedings, Supercomputing '89*, vol. , pp. 521–530, Nov. 1989.

[36] J. E. Boillat, H. Burkhart, K. M. Decker, and P. G. KROPF, "Parallel Computing in the 1990's: Attacking the Software Problem", *Physics Report (Review Section of Physics Letters)*, vol. 207, pp. 141 – 165, 1991.

[37] Richard A. Golding, "End-to-end Performace Prediction for Internet", Technical Report UCSC-CRL-92-26, Concurrent Systems Laboratory, Computer and Information Sciences, Uni, University of California, SantaCruz, CA 95064, June 1992.

[38] Jodi L. Murray and David A. Poplawski, "Performance Prediction of Distributed Memory Parallel Architectures", Computer Science Technical Report CS-TR 91-05, Michigan Technological University, Houghton, Michigan 49931-1295, Jan. 1991.

[39] M. P. Bekakos and David J. Evans, "A Model for Deterministic Performance Analysis of MIMD Parallel Computer Systems", in David J. Evans, editor, *Advances in Parallel Computing*, vol. 1, pp. 173–212. JAI Press, Greenwich, Connecticut, 1990.

[40] François Bodin, Daniel Windheiser, and William Jalby, "Performance Evaluation and Prediction for Parallel Algorithms on the BBN GP1000", *Proceedings of the 1990 International Conference on Supercomputing*, vol. 18, pp. 401–413, June 1990.

[41] Marc Abrams, Naganand Doraswamy, and Anup Mathur, "Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event, and Frequency Domains", *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 672–685, Nov. 1992.

[42] Franz Sötz, "A Method for Performance Prediction of Parallel Programs", in H. Burkhart, editor, *Joint International Conference on Vector and Parallel Processing, Proceedings, Zurich, Switzerland*, pp. 98–107. Springer, Berlin, LNCS 457, Sep. 1990.

[43] T. H. Dunigan, "Performance of the Intel iPSC/860 Hypercube", Technical Report ORNL/TM-11491, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, June 1990.

[44] Rudolf Berrendorf and Jukka Helin, "Evaluating the Basic Performance of the Intel iPSC/860 Parallel Computer", *Concurrency: Practice and Experience*, vol. 4(3), pp. 223–240, May 1992.

[45] Steven A. Moyer, "Performance of the iPSC/860 Node Architecture", Technical Report IPC-TR-007, Institute for Parallel Computation, School of Eng and App Sc, Univ of Virginia, Charlottesville, Virginia 22903, May 1991.

[46] J. A. Etheridge M. Heath, "Paragraph", Technical report, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, Oct 1991.

[47] J. R. Rice and J. Jing, "Problems to Test Parallel and Vector Languages", Technical report, CSD-TR-1016, 1990.