Capturing and Automating Performance Diagnosis: The Poirot Approach

B. Robert Helm
Alan D. Malony
Stephen F. Fickas

CIS-TR-93-25
November 1993

Abstract

This paper concentrates on the specific process of diagnosing performance problems -- a process we call *performance diagnosis*. The more general process of *performance debugging* applies diagnosis and *performance tuning* in an iterative manner. We view performance diagnosis as a methodology (or set of methodologies) for producing an explanation of performance phenomena (in particular, performance bottlenecks) that meets a set of *diagnostic requirements*. It is our opinion that performance diagnosis is the key engineering issue in effective performance debugging, but that there is a general lack of support for the performance diagnosis process.

This paper discusses the Poirot approach to providing process support for performance diagnosis.We discuss the architecture of Poirot, and describe how Poirot can re-enact or *rationally reconstruct* a set of published performance diagnosis strategies.

Department of Computer and Information Science
University of Oregon

Capturing and Automating Performance Diagnosis Methodologies: the Poirot Approach

1.0 Introduction

The *scientific method* -- the systematic testing of hypotheses through controlled measurement of observable phenomena, analysis of collected data, and modeling of empirical results -- has been advocated as a working definition of "experimental computer science" [8] and as the basis of a "quantitative philosophy of performance evaluation" [12]. In any field, computer science included, progress in experimental science is inextricably coupled with advances in observational technology; the ability to test hypotheses is critically dependent on the requisite measurement and data analysis tools. In the performance evaluation of parallel computer systems, progress is made through better understanding the performance optimization space and in developing practical techniques that can be used to maximize the performance of real application programs. Better tools to observe, model, and understand performance will lead to better solutions to performance problems.

Although the scientific method's systematic measurement and hypothesis testing is both desirable and necessary for parallel performance evaluation, the limited understanding of parallel system dynamics and the complexities of performance data make the construction of experimental apparatus especially difficult. In general, a performance analysis environment should support the ability to 1) specify new parallel performance problems in terms of the characteristics of the parallel system, the structure and parameters of the application program, previously stored performance knowledge, and the current, empirical performance data (*performance hypothesis formulation*); and 2) conduct performance experiments to assess performance behavior (*performance observation and analysis*). The first requirement reflects the notion that effective performance evaluation should involve the iterative application of a systematic methodology, namely designing new performance experiments based on cumulative system and performance data. In practice, this feature is either implicit or lacking -- the data are most often maintained in the informal knowledge base of the tools and the computer science community; rarely is the information codified in a knowledge database. The second requirement, performance observation and analysis, focuses on building and using tools to test hypotheses concerning performance problems. In particular, the need for performance data accuracy and detail to validate a hypothesis must be balanced against the observational capabilities of the performance tools and their possible perturbations of the measurement results.

Our general goal is to make advances in parallel performance evaluation more accessible in parallel software development. As Ferrari [12] noted, computer performance evaluation research has not yet significantly influenced the instruction and practice of software development. More recently, Fox [14] has argued that performance issues are insufficiently considered during software development and he advances the notion of *performance engineering*, defined by Smith [36], as a critical component in large-scale software systems. Clearly, in parallel computing where performance is paramount, a performance engineering methodology is required. Given the intellectual ferment in high-performance, parallel processing, it is important to address what are the fundamental performance engineering issues, the key problems, and the appropriate techniques for their solution.

Our interests lie in empirical performance evaluation, and in the specific process of diagnosing performance problems -- a process we call *performance diagnosis*. The more general process of *performance debugging* applies diagnosis and *performance tuning* in an iterative manner. We view performance diagnosis as a methodology (or set of methodologies) for producing an explanation of performance phenomena (in particular, performance bottlenecks) that meets specific *diagnostic requirements*. Diagnostic requirements specify the information needed -- including level of detail and accuracy -- and the resources committed to acquiring it. A good diagnosis methodology is one that meets its diagnostic requirements, gathering the needed information within the resource constraints.

It is our opinion that performance diagnosis is the key engineering issue in effective performance debugging. This opinion is supported by the fact that many of the problems that performance tool researchers have been attacking in recent years fall under the realm of performance diagnosis: performance loss (bottleneck) models for different parallel systems and programming styles [23][1][4][9][27]; measurement techniques for parallel systems that control observational complexity [33][4][25][26][19][37]; analysis methods that focus performance attention [1][23][16][28][29][38]; and experiment management and control [10][34][35].

However, the majority of the tools developed do not provide direct support for diagnosis methodology. Although significant advancements have been made in developing enabling technologies for performance diagnosis application, the lack of a formal, engineered infrastructure for capturing and automating diagnosis methodologies, we believe, has resulted in problems of *flexibility*, *extensibility*, and *retargetability* in current performance tools. With respect to influencing parallel software development practice, these problems make it difficult to address the following important performance engineering concerns:

1. Programmers do not know how to *select* (or are unaware of) the appropriate performance diagnosis techniques for their problems;

2. Programmers cannot *use* performance diagnosis tools effectively, due to the tools' complex functionality and system dependence;

3. Programmers cannot *manage* the large numbers of program files, configuration scripts, and data files generated by the performance diagnosis process.

In sum, we see a general lack of support for the performance diagnosis process. While individual tools may automate a particular diagnosis process, there has been no attempt to integrate the diagnostic techniques found in such tools into an automated diagnosis environment that is portable across machines.We believe that this requires a diagnosis system infrastructure that is engineered to provide flexibility, extensibility, and retargetability. Our research goal to create such an environment for performance diagnosis, one that addresses the three automation issues of selection, use, and management.

This paper discusses the Poirot[1] approach to providing process support for performance diagnosis.By reviewing the literature on performance diagnosis tools, we have identified a common set of methodological principles and process building blocks, applicable across a wide range of languages, machines, and measurement environments. Additionally, we have identified research in expert systems, software process modeling, and semantic databases which has partially addressed capture and automation of diagnosis methodologies. After presenting this work, we then discuss the architecture of Poirot, our initial definition of a performance diagnosis support environment. We show how Poirot can create tools that re-enact or *rationally reconstruct* a set of published performance diagnosis strategies. Lastly, we review the status of Poirot and outline future work.

2.0 Performance diagnosis assistance: research issues

Most programmers follow an iterative performance diagnose-debug cycle. They write an initial parallel version of a program, identify its principal performance problems, and then transform the program in response to those problems, repeating diagnosis if necessary. We believe choice of *methodology* -- a policy for setting up experimental program runs, collecting data, and analyzing and interpreting results -- is key for successful performance diagnosis, and, as a consequence, performance tuning. As evidence, programmers experienced with an application and its runtime environment can quickly isolate many performance bugs by careful choice of measurement tools, experimental conditions, and analysis techniques. Conversely, less experienced programmers can waste time and machine resources gathering data of low utility [19]. We thus believe that parallel performance research must identify and prescribe appropriate diagnosis methodologies for classes of applications, architectures, and measurement environments.

Many researchers prescribe methodologies by building tools to automatically support them. For instance, IPS-2 [29] and ParaGraph [17] prescribe event tracing for data collection, whereas Quartz [1] and MTOOL [16] prescribes profiling with timers and sampled counters. Prescription is not limited to lower-level decisions such as measurement technology, but also higher level decisions such as what to analyze and where. MTOOL, for instance, automatically selects instrumentation points in programs based on results from an initial time profile. IPS-2 [29] provides an automated method of critical path analysis, to focus the programmer on sequences of events in a program that determine its runtime. Crovella and Leblanc [23] show how to diagnose program performance using their *predicate profiling* tools and speedup analysis. Kirkpatrick and Schwan [23] show how to diagnose the behavior of a program by comparing its actual behavior to predictions via an abstract model, and monitoring behaviors that fall outside the predicted range.

Tools that provide complete support for a methodology often provide programmers with substantial automation of diagnosis; the programmer using MTOOL need not decide which parts of a program to instrument, for instance. They can also present a simpler interface to the programmer, since many subsystems are integrated inside the tool. Thus, the programmer using MTOOL also does not need to learn the syntax and file formats required by its profile displays, since they are integrated with data collection. Finally the tool can relieve the programmer of many kinds of management tasks; a tool which automatically generated speedup analyses from a source program, for

1. Hercule Poirot, Agatha Christie's fictional Belgian detective, famous for explaining apparently inexplicable crimes.

instance, could automatically keep track of which data set and executable file correspond to which number of processors in the analysis.

However, by "hard-wiring" a methodology into code, integrated tools limit their applicability across a broad range of architectures and programming environments. Clearly, no single methodology is appropriate for all systems and applications; certainly the choice of metrics to measure [10], and of techniques for data collection [16][23] depends strongly on the target machine. Higher level choices, such as the type of behavior to investigate in an application [23], are equally dependent on the application and the programmer's goals. Current tools frequently make machine-dependent commitments in all of these areas. The code must be extensively rewritten to change these commitments and to extend functionality. There is no formal means of describing the services a diagnosis tool requires from its environment, or the services it provides to the user. As a result, it is difficult to port tools among systems, or even to determine whether porting will be useful.

Our goal is to allow expert programmers to integrate tools (often fine-grained and system-specific) into flexible, portable diagnosis methodologies. We have defined a development environment, called *Poirot*, for creating and automating such methodologies. We base our approach on research in expert systems, software process modelling, and semantic databases. In particular, we synthesize work from these fields to address three issues:

Issue 1: How can performance diagnosis tools formally encode methodologies to automatically choose diagnosis actions, while remaining flexible and composable?

Research in knowledge-based systems has created implementation frameworks [24] [13] [11] that fill this need, and very high-level languages for describing methodologies in these frameworks. Knowledge-based diagnosis research has defined a library of building blocks for retargetable diagnosis methodologies [21] [7] [32] [5]. We intend to adapt this work to program performance in Poirot.

Issue 2: How can performance assistants interact with and integrate programming and measurement tools to simplify their use for the programmer, while still being retargetable across programming environments?

Environments for process-directed software development [22][30][20] have developed techniques for integrating externally developed programming tools into formally defined programming methodologies. We intend to extend this work, which has focused to date on general software development and group coordination issues, to apply to performance diagnosis and individual problem-solving.

Issue 3: How can performance assistants manage data and programs generated by the program environment, while remaining independent of that environment?

Research in semantic database support for programming environments [31] [2] [37] has developed mechanisms such as expressive type systems and access-directed invocation, that give applications flexible access to persistent data, independent of how and where those data are stored or computed. We intend to adapt this work to represent and manage the state of the diagnosis process in Poirot.

3.0 Poirot: An architecture for automating performance diagnosis

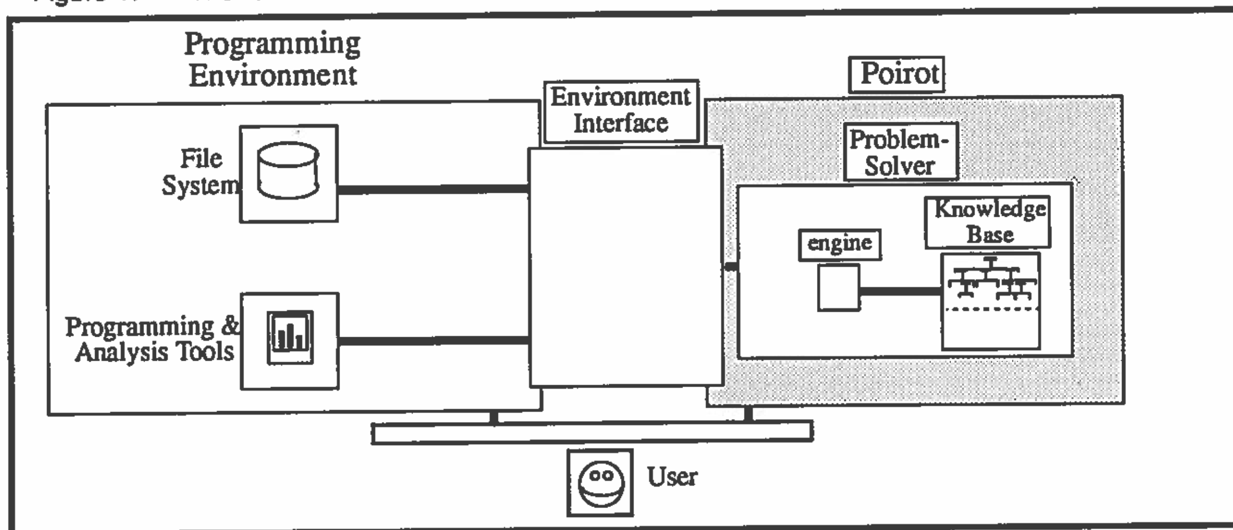
We have defined a prototype performance diagnosis environment called Poirot. Our goal is to formalize and automate performance diagnosis methodologies in a way that is extensible, flexible, and retargetable.

In this section, we sketch the architecture of Poirot and its rationale. Figure 1 gives an overview of Poirot and its role in the programming process. Poirot is a knowledge-based system (specifically, a *problem solver*). It is structured around an interpreter called the *engine* which interprets a "program" called the *knowledge base*. Poirot interacts with a target programming environment via the *environment interface*. While Poirot is designed to do performance diagnosis autonomously, we expect that in many cases there will be an active partnership with the user. The user can take any actions the problem-solver can. In addition, the user can ignore the environment interface if necessary and interact with tools directly. As in Glitter [13], the system on which Poirot is modelled, we provide means for the user to document actions taken outside of Poirot's problem-solving context as if they were taken inside.

3.1 Environment Interface

Figure 2 shows the structure of Poirot in more detail. We focus first on the environment interface. The task of the environment interface is to formally specify the diagnosis facilities of the programming environment, while hiding system-specific details of their implementation. In this way, Poirot can be installed in any system that supports the objects and operations of the interface.

Figure 1. The role of Poirot.



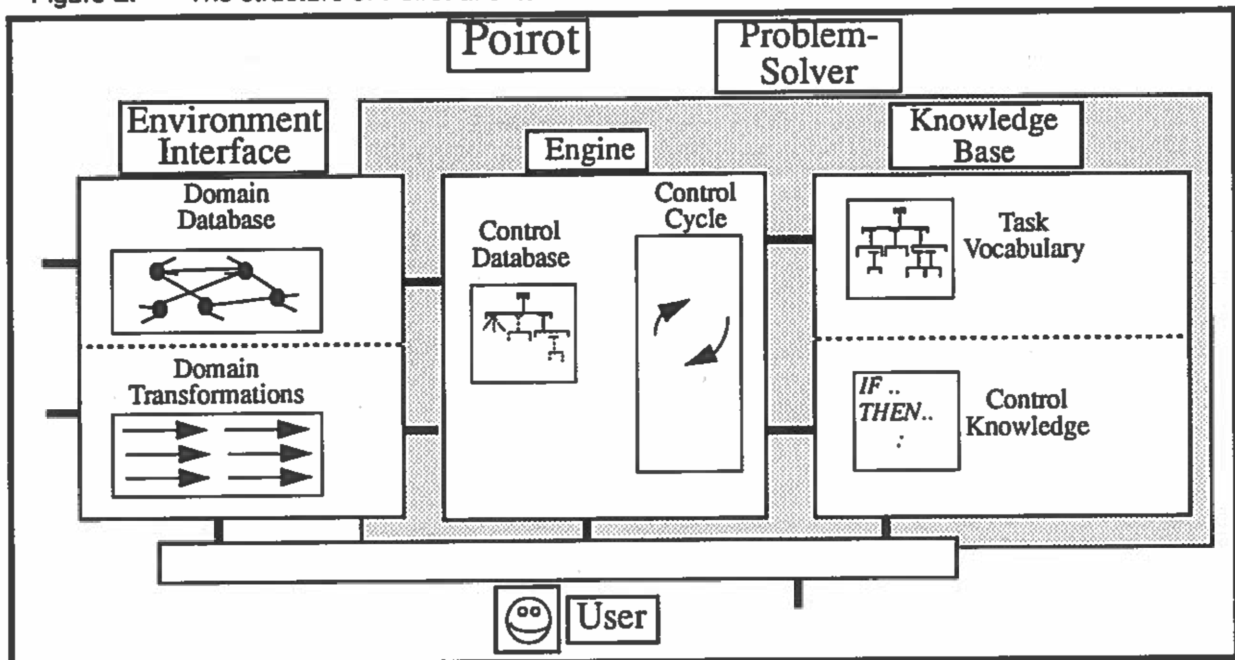
This has two benefits: (1) it ensures retargetability of methodologies, by allowing them to examine the state and to invoke the utilities of the environment without being tied to the peculiarities of a particular data representation or tool (2) it presents a simpler view of the environment to the user, by hiding many details such as command syntax and file formats.

The environment interface consists of a *domain database*, and the *transformation library.* The domain database summarizes the state of the diagnosis project, using a uniform information model which represents such diverse items as data sets, programs, and configurations files as

objects with *attributes*. Objects may also represent assertions about the target environment, such as "The target system does not have a cache monitor." Several programming environments have similarly adopted a persistent, active database as a facility for data storage and tool integration.[34][23]. Poirot's database supports associative retrieval of objects; the problem-solver relies heavily on this ability.

Poirot interacts with the tools in the environment by representing them as a uniform library of transformations. This is similar to the approach taken in Marvel [15]. Each transformation in the library represents a low-level performance diagnosis action, such as inserting an instrumentation point in a program. Each transformation has an *interface* that characterizes its requirements and effects in terms of the domain database. Transformations can be used simply to update the database. However, most transformations also have an *action*, a script that performs computation, exchanges information with the user, or invokes a tool in the programming environment and translates its results into objects in the domain database. In addition to hiding tool interfaces, transformations support project management, by generating and maintaining links among entities in the environment and objects in the database.

Figure 2. The structure of Poirot and its interface.



3.2 Problem solver

Poirot's problem solver selects diagnosis actions and carries them out via the environment interface. The problem solver is based on the Glitter system[13]. Its actions are determined by its knowledge base, which is divided into *task vocabulary* and *control knowledge*.

3.2.1 Task vocabulary

The task vocabulary describes the *goals* and *methods* of methodologies. Goals represent tasks that must be performed in a performance methodology. These tasks may be high- or low-level; thus, "**evaluate** whether there is an synchronization bottleneck" and "**instrument** statement 1" could both be goals.

Methods describe means for accomplishing the goals. For example, the goal "**evaluate** whether there is an synchronization bottleneck" might be solved by the method "Measure synchronization rate". Methods are indexed by the goals that they address. The *body* of a method is a set of diagnosis actions, of two types. One type of action is to post *subgoals*, representing subtasks that must be performed to carry out the method. The subgoals of "Measure synchronization rate", for instance, are "instrument with synchronization counters", "run program", and "collect data". The other type of action is to invoke transformations via the environment interface. For example, the "Measure synchronization rate" method invokes a transformation that generates a new version of the program to instrument.

3.2.2 Problem-solving engine and control knowledge.

Given goals and methods, the problem-solver automates selection of diagnosis actions by top-down goal refinement. As an example, assume the system is pursuing the goal "**evaluate** whether there is an synchronization bottleneck" introduced above. The problem-solving *engine* retrieves methods which are indexed to this goal. If more than one method is retrieved, this creates a choice point; the engine uses *control knowledge* to choose one of the competing methods. The chosen method then executes its body, which (typically) posts subgoals. If there is more than one subgoal, this creates another choice point: in which order should the goals be pursued? The engine uses control knowledge to select one of the goals. The cycle repeats.

The process terminates successfully when all goals have been solved. The state of this process -- the goals that have been posted, and the methods that have been proposed or carried out -- is stored in the problem-solver's *control database*.

Poirot's problem solving engine has no fixed policies for selecting among methods to execute or goals to pursue. These policies are defined by the control portion of its knowledge base. The control knowledge is structured as a set of condition-action rules. For each control decision, relevant rules are matched against the state of the control and domain databases. Rules with satisfied conditions add or delete assertions in the control database, which the engine then interprets to resolve the decision. The user may add assertions to the control database to influence the behavior of the system. This provides a mechanism for specifying application-, architecture-, and environment-specific diagnostic requirements to Poirot. This is discussed in more detail in section 4.2.1.

Poirot thus represents performance diagnosis methodology decisions explicitly as control decisions, and separates the formulation of those decisions (the task vocabulary) from the policies for resolving them (the control knowledge). This has several benefits for our implementation goals of extensibility, flexibility and retargetability. We describe these benefits in more detail in the sections 4.2 and 4.3.

4.0 Using Poirot: Three reconstructions

In this section we evaluate the sufficiency and usefulness of Poirot, by *rationally reconstructing* several proposed performance diagnosis methodologies. In rational reconstruction, we show how Poirot can formally encode a methodology, execute that methodology on a well-defined external interface, and produce comparable results. We do not make any claim about the appropriateness or usefulness of the methodologies we reconstruct; we simply show that Poirot can at least implement the methodologies the performance diagnosis field has devised to this point.

We first show how Poirot can re-implement the W3 search model of Hollingsworth and Miller [19]. We then demonstrate how Poirot can be retargeted to reconstruct the methodologies associated with ChaosMon [23] and PTOPP [10]. We chose these three because each includes one or more *idiosyncratic* techniques for performance diagnosis. Despite their idiosyncracies, our results suggest that these different methodologies share a substantial core of performance diagnosis knowledge, knowledge we can formally capture in Poirot in a general and portable form.

4.1 Example methodology: W3 model.

The first methodology we reconstruct is the W3 search model of Hollingsworth and Miller [19]. Our aim in this section is not to justify the W3 methodology. We simply present it as an example of the kind of methodology Poirot should be able to generalize, automate, and retarget. Our goal is to cast W3 in terms of more general diagnosis models, which supports both reuse across diagnosis contexts and integration with other methodologies.

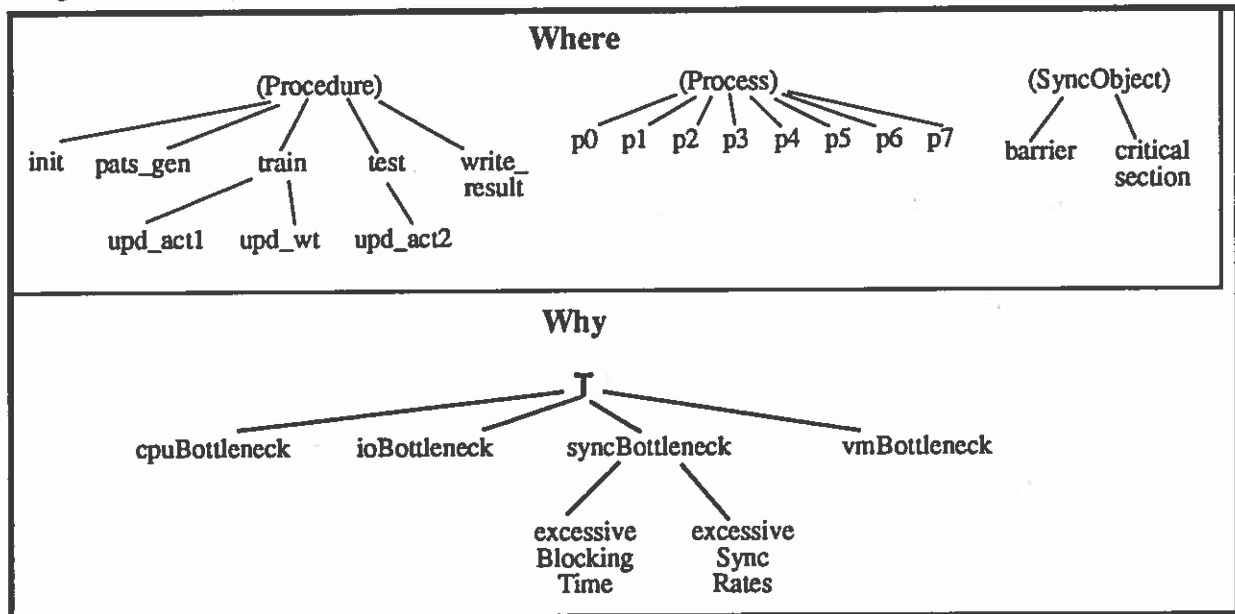
We first describe the W3 model and some aspects of its current implementation. W3 performs search in a space of possible diagnoses, called *hypotheses*, during an instrumented run of a program. Each hypothesis represents an abstract performance problem that may be present in the program. Search begins with the initial hypothesis that there is a performance problem. The W3 system investigates possible *refinements* of the hypothesis, where a refinement is a more specific hypothesis about the performance problem. Hypotheses have three dimensions, "Why", "Where" and "When" (thus the term W3). The "why" portion of the hypothesis describes the nature of the performance problem occurring. The "where" portion describes where in the program the problem is occurring, by giving coordinates of the form (procedure, process, object). Finally, the "when" portion of represents the time interval during a run that the performance problem is occurring. Each hypothesis is associated with test code that checks whether the performance problem it represents is present over a particular time interval of the run. The test enables sampled data collection from relevant (pre-inserted) event counters in the program, collects data, performs analysis, and returns results. The results are used during search to decide whether to refine the hypothesis, or to discard it.

To illustrate the W3 model more concretely, we describe how it could (hypothetically) be used to diagnose performance problems in a example program. The program is a neural network simulation code written in Dataparallel C (DPC) [18], running on a Sequent Symmetry multiprocessor. Dataparallel C code compiles to single-program multiple data (SPMD) code; that is, the program is compiled to processes that share the same code but that operate on different portions of the data space. Processes communicate through the shared memory, and synchronize via barriers and critical sections.

Figure 3 shows a portion of the pre-enumerated "why" and "where" spaces that W3 uses to generate hypotheses for the neural net program. Diagnosis with W3 cycles between (1) initiating a run of the program to be diagnosed, (2) search for hypotheses whose tests are satisfied by data collected during that run, and (3) refining the satisfied hypotheses. W3 focuses initially on bottlenecks in one or more of synchronization, I/O, cpu utilization, and virtual memory for the entire program. Assume, for example, the synchronization bottleneck test succeeds. The system then refines the hypothesis to "critical section bottleneck" and "barrier bottleneck" and enables tests of these hypotheses. Assume the critical section hypothesis passes. W3 then refines further to iden-

tify the procedures in which problematic critical sections lie, and the phases during those procedure's runs where the problems are acute. The final result of a W3 diagnosis session is a list of satisfied hypotheses, each defining a particular performance bottleneck. The programmer's task is then to transform the program to remove these bottlenecks.

Figure 3. Why and Where axes for Dataparallel C neural net program.



4.2 Reconstructing W3

To illustrate the sufficiency of Poirot, we trace its operation as it reproduces W3's diagnosis behavior on the Dataparallel C neural net program (called "nnet") introduced in section 4.1. We show how Poirot could achieve a level of automation comparable to that of W3, while at the same time meeting our goals of flexibility, extensibility, and retargetability.

4.2.1 Flexibility and extensibility: responding to diagnostic requirements.

We stated in the introduction that methodologies should be chosen to meet explicit *diagnostic requirements*. In current performance diagnosis tools, users state diagnostic requirements by setting the values of a fixed set of control parameters. This often provides an overly restricted language for specifying diagnostic requirements. It also leads to a conflict: if the tool provides too few control parameters to express the user's requirements, the end users must recode the tool to get the control they need. Recoding tools difficult and time-consuming, even in those cases when source code is available. Conversely, if the tool provides too many control parameters, the tool is difficult to implement, learn and use. Poirot supports custom, extensible languages for stating diagnostic requirements. Poirot also mitigates the conflict between simplicity and flexibility by (1) separating control from other aspects of its knowledge base, and (2) by representing control in the form of rules. We use the initial part of the diagnosis session with the neural net program to illustrate these points.

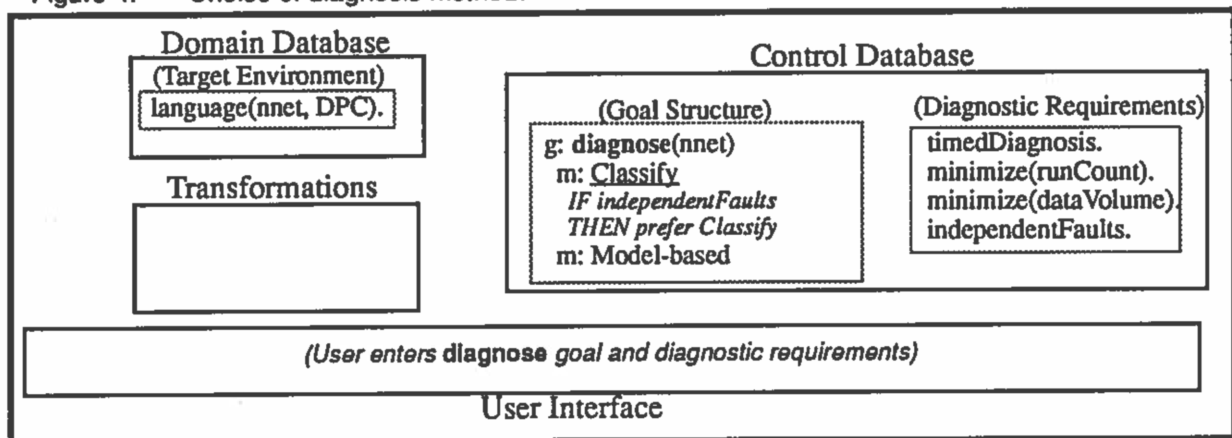
The user begins diagnosis of the neural net program by posting the goal **diagnose**(nnet). Goals include *achievement conditions* -- executable tests that query the domain database or the user to determine whether a goal has been met. The achievement condition of **diagnose**(nnet), for

instance, takes a proposed diagnosis (a hypothesis) from the database and asks the user whether the diagnosis is acceptable. The arguments of a goal (such as "nnet" in the **diagnose** goal), identify the objects the goal focuses on and distinguish different instances of the same goal.

Along with the goal **diagnose**(nnet), the user adds several assertions representing diagnostic requirements to the control database. In particular, the user specifies that (1) the diagnosis returned should identify the time when the performance problem is occurring (timedDiagnosis), (2) the system should minimize the number of times the program is run during diagnosis (minimize(runCount)), (3) the system should minimize data volume (minimize(dataVolume)), and (4) the system need not investigate interactions of multiple performance problems (independent Faults). The domain database already contains the assertion that "nnet" is a Dataparallel C program.

Figure 4 shows Poirot's initial response to the **diagnose**(nnet) goal, and introduces our notation for depicting Poirot's state in this paper. The figure shows the relevant parts of the domain database and the control database. It summarizes Poirot's interaction with the user, and with the environment (via transformations). The *goal structure* part of the control database indicates the goals Poirot is pursuing (indicated by "g:"), the methods it is considering (indicated by "m:"), and the control rules it is applying (indicated by italics). The methods Poirot retrieves for a goal are indented below the goal. Subgoals of a method (where they appear) are indented below the method. The methods that Poirot has chosen for a goal are underlined.

Figure 4. Choice of diagnosis method.



Poirot's problem-solving engine first searches for a method that can address the **diagnose** goal. Two methods, "Classify" and "Model-based" [7] are retrieved. Based on the user's assertion of "independentFaults", a control rule posts a preference for "Classify", which is then selected. Figure 4 shows this problem-solving state.

The choice between "Classify" and "Model-based" above corresponds (roughly) to the choice between two major subroutines in a tool. However, to make this choice sensitive to "independentFaults", it was not necessary for the user to locate the calling point(s) in the source, add a conditional statement and variable, and recompile. Instead, the parameter "independentFaults" was added into the control database schema, and the rule was added into the control database, immediately making the new behavior available. Poirot thus supports modular extension of its diagnostic requirement language, without requiring wholesale recoding of its knowledge base. It automatically selects diagnosis techniques based on requirements stated in the language.

4.2.2 Retargetability: Automating diagnosis using generic methods.

Poirot's knowledge base incorporates techniques for automated diagnosis developed in the field of expert systems and artificial intelligence [4][7]. Many of the techniques developed by these fields are "domain-independent"; that is, they rely on certain kinds of knowledge being available about the faults they are looking for, but whether faults represent diseases or short circuits does not affect the basic structure and processing of the technique. We can package these techniques as methods and retarget them to different platforms, as is shown in this section.

"Classify", the method selected for the **diagnose** goal, implements diagnosis by top-down classification [4], a common approach when faults can be investigated independently (thus the control rule discussed above). Applied to performance problems, it attempts to classify the program's behavior as reflecting one or more pre-enumerated classes of performance problems. It works by refinement, starting with a most general class "performance problem" and then gathering information to refine the classification. We can use this generic method to reimplement W3 by treating W3's hypotheses as classes.

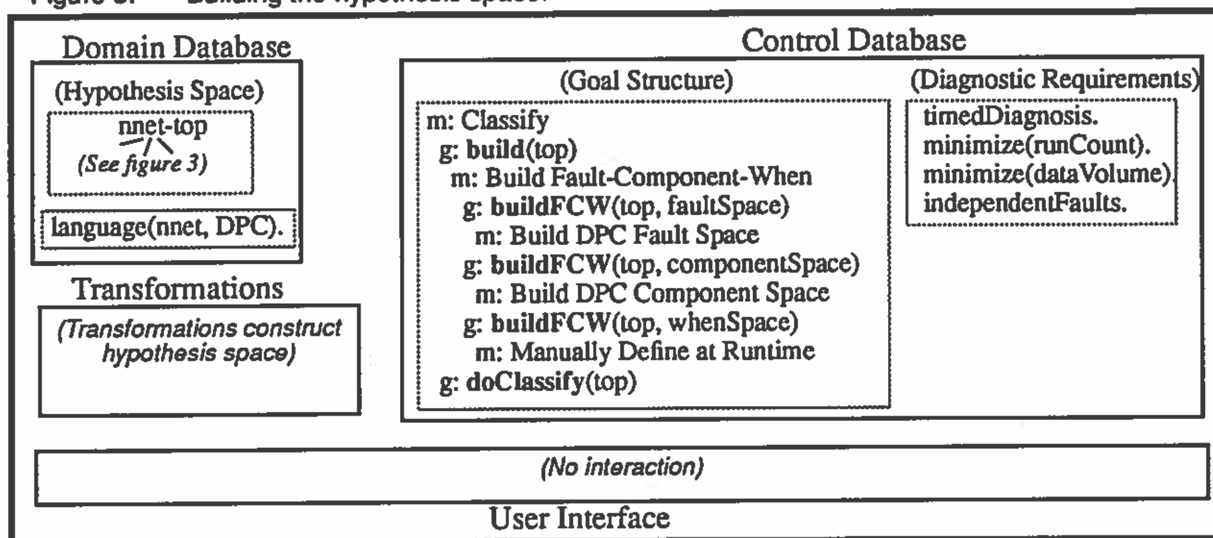
As shown in figure 4, Poirot has chosen "Classify", based on its control knowledge. The body of the "Classify" method calls a transformation to add the most general hypothesis (called "top"), then automatically posts an ordered list of subgoals: **build**(top), and **classify**(top). Poirot calls the achievement condition of the first goal, **build**, which calls for the pre-enumeration of the hypothesis space. As in W3, hypotheses in this space have "why", "where", and "when" dimensions. Following more general diagnosis terminology [7], we call the "why" and "where" dimensions "fault" and "component", respectively.

Given that the goal **build**(top) has not been achieved at this point, Poirot retrieves the method "Build Fault-Component-When" (BFCW), the only method for **build** that satisfies the user's requirement "timedDiagnosis". Poirot chooses and executes BFCW, which adds the hypothesis space to the domain database. Figure 5 summarizes the goal structure and results of the BFCW method. Based on the assertion that "nnet" is a Dataparallel C program, BFCW invokes transformations that construct the space shown in Figure 3. Note that all but the lowest-level methods used to achieve the **build** goal are independent of the target platform.

4.2.3 Retargetability: Separating local policies from reusable techniques.

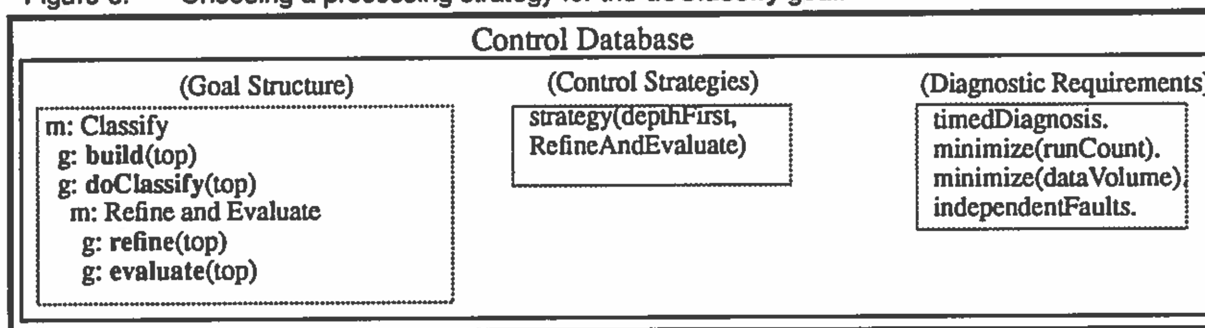
A programmer often makes performance diagnosis decisions based on concerns that may be called *local*: the current application's demands, the target machine's resources, the programming environment's tools, or the programmer's own tastes. Current automated diagnosis tools often mix local decisions with those based on more general concerns. The result is that one cannot effectively retarget the tool, even though many of its methods are not, in principle, limited to a particular local environment. Poirot attempts to remedy this problem by allowing users to separate knowledge that addresses local concerns from knowledge that is more general. In particular, Poirot can encode local policies in control knowledge, so that the task vocabulary remains maximally reusable.

Figure 5. Building the hypothesis space.



This is illustrated by Poirot's initial strategy for processing the **doClassify** goal (Figure 5). Poirot selects the generic method "Refine and Evaluate" for this goal. It has two types of subgoals: **refine**(hypothesis), which refines a hypothesis, and **evaluate**(hypothesis), which collects data to evaluate a hypothesis. "Refine and Evaluate" does not, however, specify an order in which to do these two goals. Recall that the user asserted "minimize(dataVolume)", likely because of limited resources in the target machine or in the monitor. The control rules respond to this assertion by implementing a strategy for "Refine and Evaluate" that evaluates one hypothesis at a time, minimizing the data generated. Specifically, Poirot proceeds depth-first under the constraint of minimum data volume. It pursues **evaluate** for each hypothesis, until one is confirmed. It then recursively investigates the refinements of the confirmed hypothesis, following the same strategy.

Figure 6. Choosing a processing strategy for the **doClassify** goal.



Poirot's use would be limited in environments that tolerated high data volumes, if its problem-solving engine hard-wired the depth-first strategy. One way around this would be to write multiple versions of the "Refine and Evaluate" methods that follow different strategies. However, there are potentially a plethora of variants of this method, one for each conceivable interleaving of refining and evaluating hypotheses. Poirot allows the tool builder separate general diagnosis strategies from strategies that address local concerns, by placing the latter in control knowledge. This helps keep its task vocabulary small and widely applicable.

4.2.4 Retargetability: Separating tool interfaces from tool implementations.

We rely on our library of transformations to insulate the performance diagnosis methodologies in Poirot from the details of the tools in the environment in which Poirot is embedded. The next section shows how this occurs in our example diagnosis session with "nnet".

By pursuing depth-first search from the "top" hypothesis, Poirot eventually focuses on the goal **evaluate**(syncBottleneck). We follow the processing of this goal in further detail to show how we model the W3 measurement methodology. Figure 7 outlines the goal structure of this process. Figure 8 shows the successive hypotheses generated during this part of the search.

W3 tests for a bottleneck by measuring a performance metric that directly correlates with that bottleneck. For instance, given the hypothesized bottleneck, "syncBottleneck", W3 would measure a metric that indicates the level of synchronization activity. Poirot selects a method called "Directly Measure" for the goal **evaluate**(syncBottleneck) that implements W3's method of hypothesis testing.

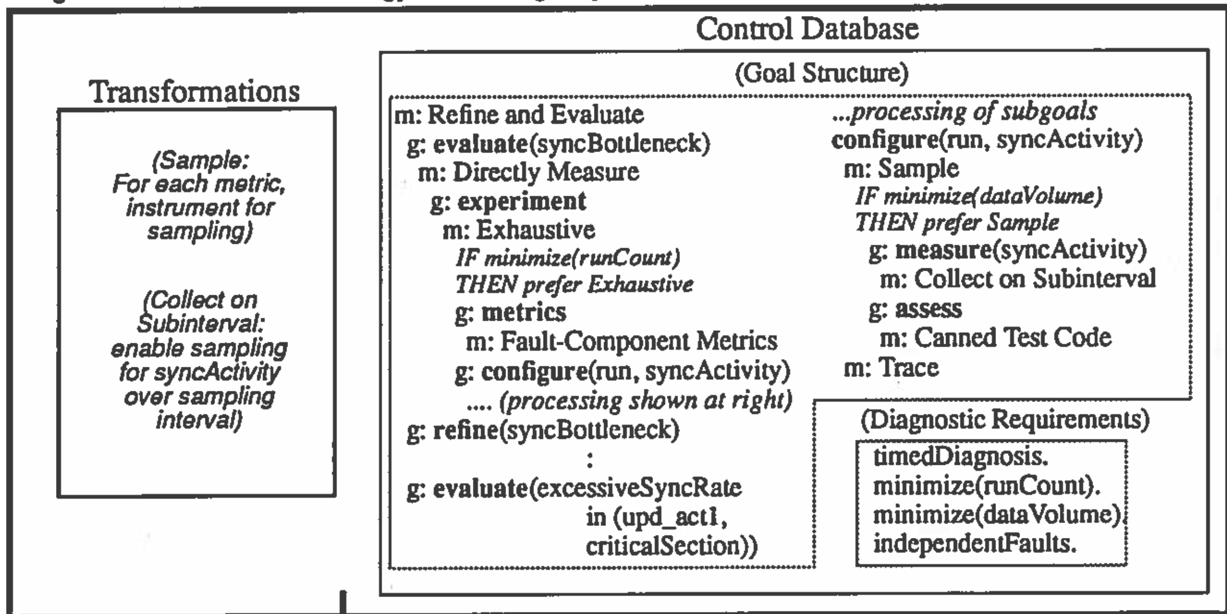
The "Directly Measure" method posts the subgoal **experiment**, which asks for an experiment to perform the required measurements. Guided by the requirement to minimize the number of runs, Poirot selects a method, called "Exhaustive", that collects data from a single program run to test all possible bottlenecks. The hypothesis space (pre-enumerated by the "Classify" method earlier) serves to identify which faults and components are possible in the target architecture. Working with this hypotheses space, the "Fault-Component Metrics" method invokes transformations that, given a fault and a component, select an appropriate metric for estimating the severity of the fault in that component. In the example, a transformation posts "syncActivity" as the metric for the hypothesis "syncBottleneck".

When metrics have been selected, the "Exhaustive" experimentation method then generates a run object [34] to represent the experimental configuration. For each metric, "Exhaustive" posts a subgoal **configure**(run, metric), which requests that the run be instrumented to gather data for the metric. Poirot chooses the method "Sample" for all **configure**(run, metric) goals, as sampling generates the least data (satisfying minimize(dataVolume) and can handle the metrics in question. The "Sample" method configures the run for data collection.

When configuration is complete, "Exhaustive" posts the **measure**(syncActivity) goal for each metric. Processing this goal confirms the "syncBottleneck" hypothesis. As a result, the depth-first strategy begins to investigate the refinements of this hypothesis. Figure 8 shows the hypotheses generated by that search.

We stated that the method "Sample" configures the run with instrumentation for each metric. It does so with a standard set of transformations from Poirot's environment interface. Depending on the environment, each of these transformations may operate by inserting source-level counters, by linking the program to an instrumented runtime system, or by asking the user to insert instrumentation points. Each implementation presents the same transformation interface to the "Sample" method. As a result, this "Sample" method remains reusable across any programming environment that can implement the transformations invoked by the method.

Figure 7. Measurement strategy for locating a synchronization bottleneck.



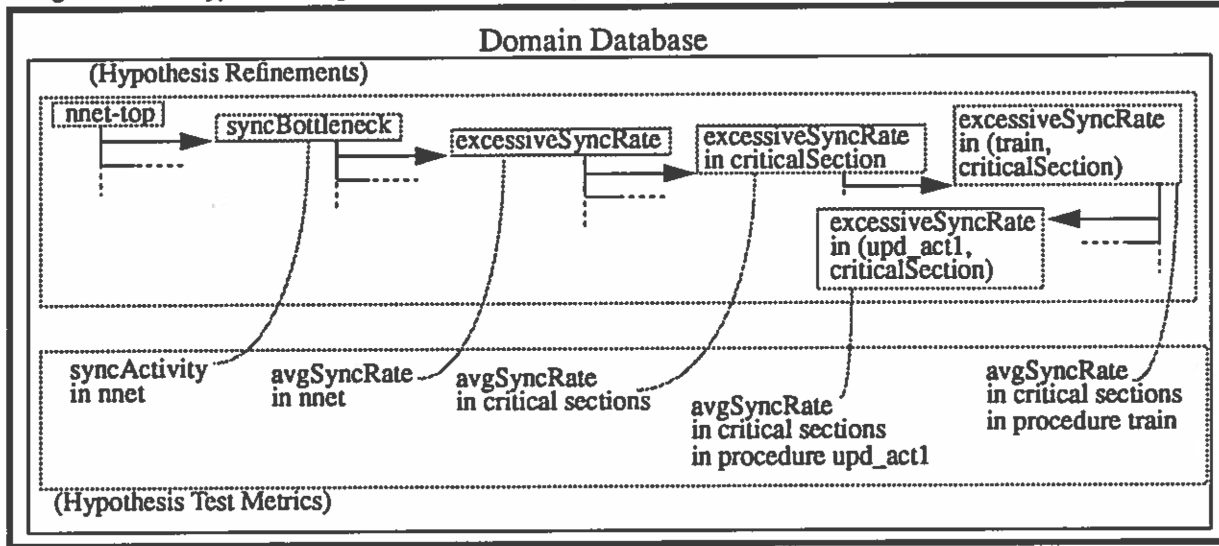
4.2.5 Flexibility: changing diagnosis strategies for changing circumstances.

We have argued that different diagnosis strategies are appropriate in different environments. It is often the case that, in addition, it is useful to change diagnosis strategy within a single environment. For instance, one might change strategy in response to new data. Poirot can implement such dynamic shifts of methodology, as is illustrated by the next part of our "nnet" example.

After several iterations of measurement and refinement (summarized in Figures 7 and 8), Poirot posts the goal **refine**("excessiveSyncRate in (procedure upd_act1, criticalSection)"). The user decides to investigate the dynamic behavior of this bottleneck. To do this, the user manually invokes a method to refine "excessiveSyncRate in (procedure upd_act1, criticalSection)" along the "when" axis, and partitions the remaining run time of "upd_act1" into four time intervals. This results in four new excessiveSyncRate hypotheses for these time intervals, and corresponding **refine** and **evaluate** goals.

These new timed **refine** and **evaluate** goals, and the user's requirement to minimize the number of runs, motivate a change in goal selection policy. Each timed hypothesis requires data to be collected during a particular time interval; if that hypothesis is not tested at the appropriate time during the current run, Poirot will have to re-run the program to test the hypothesis. To avoid this, Poirot's control rules change its strategy from depth-first to a time-sensitive strategy: pursue the goal **evaluate**(hypothesis) when (and only when) the time of the current run matches the time of the hypothesis. Thus, if the current program run is at time 1, Poirot will pursue the goal **evaluate**("excessiveSyncRate in (procedure upd_act1, criticalSection) during interval 1") over other goals. This gives a better chance of catching the incoming data.

Figure 8. Hypotheses generated in locating a synchronization bottleneck.



With this change, Poirot confirms the hypothesis "excessiveSyncRate in (procedure upd_act1, criticalSection) during interval 3". We assume the user accepts this as a diagnosis, ending the session. Note that the user intervened only three times during the diagnosis session: (1) the user posted the initial **diagnose** goal, (2) the user initiated dynamic analysis of the excessiveSyncRate bottleneck, and (3) the user accepted the final diagnosis. Only the second action required the user to use a tool other than Poirot itself. During the session, Poirot saved the intermediate results in the domain database including the solution path, the instrumented program, and the collected data, making them available for later queries. This illustrates the extent to which Poirot can support automation of selection, use and management, when provided with sufficient knowledge.

4.3 Reconstructing PTOPP and ChaosMon.

To assess the degree to which Poirot supports extensibility and retargetability, we briefly sketch how the knowledge base described above could be extended to reproduce other performance diagnosis methodologies. We focus in particular on those associated with PTOPP [9], and ChaosMon [23]. Further details and other reconstructions from our literature search may be found in [19].

The PTOPP methodology was designed to support tuning of parallelized Fortran programs for the Cedar multiprocessor. It has several interesting features. First, it has a well-defined set of faults and metrics, described in [10]. It uses perturbation analysis, a generalization of speedup analysis, to detect performance problems. Finally, it addresses the issue of managing programs and data. Naming conventions are used to identify whether a source program is (for instance) an uninstrumented sequential version or a 32-processor instrumented version. PTOPP includes a facility for automatically building appropriate executables based on these conventions. PTOPP it also includes a database for storing performance data.

We can represent the PTOPP methodology in Poirot as follows:

- Add Cedar Fortran equivalents of the DPC-specific methods for constructing the fault and component spaces (see Figure 5).

- Define a perturbation method (a sibling of "Directly Measure" in Figure 7) for processing **evaluate** goals.
- Add control knowledge that tests whether the target runtime system is Cedar Fortran-like, and to choose the appropriate methods if it is.

The management functions of PTOPP are well-supported by Poirot. Program objects can have attributes describing their state, obviating the need to for naming conventions and the build facility. The database schema can be extended for performance data.

It is thus possible to reuse components of the W3 knowledge base to handle significant features of PTOPP. We also consider retargeting Poirot to integrate the distinctive features of the ChaosMon monitoring environment. Chaos Mon's methodology is driven by an application-specific *monitoring model*. The monitoring model describes the structure of the program in terms of communicating objects with attributes. The methodology allows the user to create *abstract views* that filter and aggregate performance data. Abstract views obtain their input data from *monitoring views*, high-level data collection programs that describe how to update the monitoring model from program variables during execution. ChaosMon provides a compiler which generates instrumentation code from view specifications. This compiler includes a planning module which automatically selects between tracing or sampling for data collection.

We add support for ChaosMon in Poirot as follows:

- Create new methods for building the fault-component space (Figure 5) that uses the application-specific monitoring model, rather than the program and runtime system.
- Add a new counterpart of the method "Fault Component Metrics" (Figure 7) that invokes ChaosMon's tools for editing views. For each fault and component, the method creates a view to test for the fault in the component.
- Add a method for the **configure** goal that compiles views. This method has two separate subgoals. The **plan** subgoal causes the planning module to run and post its conclusions to the database. Control rules then use these conclusions to select between "Trace" and "Sample" for each instance of the **instrument** subgoal.

Note that we have subdivided Chaos Mon's view compiler two independent pieces. This allows us to substitute submodules in for one of the pieces when it does not apply. On a system that only supports sampling, for instance, we can cut use a control rule to cut the planner out of the loop. By providing facilities for flexible assembly of methodologies, Poirot encourages the creation of more fine-grained, special-purpose tools. While this is in contrast with the current trend toward integration of tools, we believe it is likely to lead to greater simplification of tools and their interfaces.

5.0 Status and Future Work

We are currently implementing an initial version of the Poirot solver, and a prototype of its environment interface. Our near-term goal is to construct a knowledge base consisting of the three methodologies described here, plus several others from our initial literature review (e.g., those

from Quartz and Mtool). We will test these methodologies in an advisory role (not initially requiring them to interact with tools in the programming environment) on an actual tuning project. This is intended to test the problem-solving aspects of the Poirot architecture, to provide a core task vocabulary, and to help formulate an appropriate environment interface. Our first "applied" implementation of the environment interface will be targeted to the pC++ programming environment [3]. The long-term goal of this effort is to work with end users of pC++ to experimentally evaluate the benefits of the approach on selected code tuning and porting projects.

We view Poirot as a first step towards our long-term research plans of formalizing and automating methodologies for parallel performance evaluation and optimization. Our research focuses not only on automation of performance diagnosis, but also on the general principles that enable it -- knowledge-based system organization, generic problem-solving techniques, and high-level interaction with environment data and tools. We believe these could equally benefit other aspects of performance engineering. For instance, work on the PTOPP methodology [10] encompassed performance debugging proper, capturing process information for performance tuning as well as diagnosis through transformational directives; this has close relation to Glitter's original target application [13]. Additionally, our work on a framework for knowledge-based performance optimization will benefit from the Poirot results.

Finally, we believe that formalizing methodology in a framework like Poirot's may benefit researchers on performance evaluation, independent of its value to programmers. In particular, it provides a means of documenting results in the field: formally characterize the issues (goals) a performance tool addresses, identify the positions (methods) it takes on those issues, and specify the rationale (control rules) for the choices it makes and its use. The result is a detailed encoding of methodology that may be used to compare competing approaches. In addition, Poirot's ability to define methodologies independent of tool implementations suggests a new, "need-driven" [9] approach to performance tool design and development: formulate diagnosis methodologies based on the diagnostic requirements, and then create new tools, or adapt existing tools, to support the methodology. This approach could produce tools that more directly meet the needs of programmers by allowing them to create application-specific diagnosis assistants.

6.0 References

[1] Thomas Anderson and Edward Lazowska, "Quartz: a tool for tuning parallel program performance", *Proceedings 1990 ACM SIGMETRICS*, May 1990, pp. 115-125.

[2] Dennis G. Allard and David S. Wile, "Aggregation, persistence and identity in worlds", in John Rosenberg and David Koch (eds.), *Persistent Object Systems*. Berlin: Springer-Verlag, 1990, 161-174.

[3] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, B. Mohr, "Implementing a parallel C++ runtime system for scalable parallel systems", *Supercomputing '93* (Portland, OR, November 1993), to appear.

[4] H. Burkhart and R. Millen, "Performance measurement tools in a multiprocessor environment", *IEEE Transactions on Computers*, Vol. 38, No. 5, May 1989, pp. 725-737.

[5] B. Chandrasekharan, and T. Johnson, "Generic Tasks and Task Structures: history, Critique, and New Direction", in Jean-Marc David, Jean-Paul Krivine, Reid Simmons (eds.) *Second Generation Expert Systems*. Berlin: Springer-Verlag, 1992, pp. 232-272.

[6] Mark Crovella and Thomas J. LeBlanc, "Performance debugging using performance predicates", *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1993, pp. 140-150.

[7] Johan de Kleer, Brian Williams, "Diagnosing Multiple Faults", *Artificial Intelligence*32, 1987, pp. 97-130.

[8] Peter J. Denning, "What is experimental computer science?", *Communications of the ACM*, Vol. 23, No. 10, October 1980, pp. 543-544.

[9] Rudolf Eigenmann and Patrick McClaughry, "Practical tools for optimizing parallel programs", Technical Report 12-76, Center for Supercomputing Research & Development, Urbana-Champaign, IL, 1992.

[10] Rudolf Eigenmann, "Toward a methodology of optimizing programs for high-performance computers", Technical Report 11-78, Center for Supercomputing Research & Development, Urbana-Champaign, IL, 1992.

[11] L. Erman, P. London, S. Fickas, "The design and example use of Hearsay-III", in *IJCAI-7* (Vancouver, BC, 1981), pp. 409-415.

[12] Domenico Ferrari, "Considerations on the insularity of performance evaluation", *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 6, June 1986, pp. 678-683.

[13] S. F. Fickas, "Automating the transformational development of software", *IEEE Transactions on Software Engineering*, Vol 11, No. 11 (November 1985).

[14] G. Fox, "Performance engineering as part of the development life cycle for large-scale software systems" In: *Proc. of the 11th Int. Conf. on Software Engineering*, (Pittsburgh, PA). IEEE Computer Society Press, 1989, pp. 85-94.

[15] Mark A. Gisi and Gail E. Kaiser, "Extending a tool integration language", in *1st International Conference on the Software Process: Manufacturing Complex Systems* (Redondo Beach, CA, October 1991), pp. 218-227.

[16] Aaron J. Goldberg and John L. Hennessy, "Mtool: an integrated system for performance debugging shared memory multiprocessor applications", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 1, January 1993, pp. 28-40.

[17] M. T. Heath and J. A. Etheridge, "Visualizing the performance of parallel programs," *IEEE Software*, Vol. 8, no. 5, September 1991, pp. 29-39.

[18] Philip Hatcher and Michael Quinn. *Dataparallel Programming on MIMD Computers*. Cambridge, MA: MIT Press, 1991.

[19] B. Robert Helm, "A bestiary of performance diagnosis methodologies", Technical Report 93-24, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403.

[20] Jeffrey K. Hollingsworth and Barton P. Miller, "Dynamic control of performance monitoring on large scale parallel systems", 1993 *Proceedings of the International Conference on Supercomputing*, July 19-23, 1993.

[21] Karen Huff and Victor R. Lesser, "A plan-based intelligent assistant that supports the software development process", in Peter Henderson (ed.), *SIGPLAN Notices*, Vol. 24, No. 2, February 1988.

[22] J. R. Josephson, B. Chandrasekharan, J. Smith, M. Tanner, "A mechanism for forming complex explanatory hypotheses", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-17, No. 3, May/June 1987, pp. 445-454.

[23] Gail E. Kaiser, Peter H. Feiler, and Steve S. Popovich, "Intelligent assistance for software development and maintenance", *IEEE Software*, Vol. 5, No. 3, May 1988, pp. 40-49.

[24] Carol Kilpatrick and Karsten Schwan, "ChaosMon -- Application-specific monitoring and display of performance information for parallel and distributed systems", *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991, pp. 48-59.

[25] J. Laird, A. Newell, P. Rosenbloom, "SOAR: an architecture for general intelligence", *Artificial Intelligence*33(1): 1987, pp. 1-64.

[26] Allen D. Malony, "Performance observability", Ph.D. dissertation, University of Illinois at Urbana-Champaign, October 1990.

[27] Allen Malony, Daniel Reed, and Harry Wijshoff, "Performance Measurement Intrusion and Perturbation Analysis", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 4, July 1992, pp. 433--450.

[28] Dan Marinescu, James Lumpp, Thomas Casavant, and H.J. Siegel, "Models for monitoring and debugging tools for parallel and distributed software", *Journal of Parallel and Distributed Computing*, Vol. 9, No. 2, June 1990, pp. 171-184.

[29] Margaret Martonosi, Anoop Gupta, and Thomas Anderson, "MemSpy: analyzing memory system bottlenecks in programs", *Proceedings 1990 ACM SIGMETRICS*, May 1992, pp. 1-12

[30] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, Timothy Torzewski, "IPS-2: the second generation of a parallel program measurement system", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990, pp. 206-216.

[31] Naftaly H. Minsky and David Rozenshtein, "A software development environment for law-governed systems", *ACM SIGSOFT Software Engineering Notes*, Vol. 15, No. 6, 1990, pp. 65-75.

[32] David Ogle, Karsten Schwan, and Richard Snodgrass, "Application-dependent dynamic monitoring of distributed and parallel systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 7, April 1993, pp. 762-778.

[33] Yun Peng and James A. Reggia, "A probabilistic causal model for diagnostic problem solving part II: diagnostic strategy", *EEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-17, No. 3, May/June 1987, pp. 395-406.

[34] Daniel A. Reed, "Performance Instrumentation Techniques for Parallel Systems", in L. Donatiello and R. Nelson (eds.), *Models and Techniques for Performance Evaluation of Computer and Communication Systems*. Berlin: Springer-Verlag, Lecture Notes in Computer Science, 1993.

[35] Zary Segall, Ajay Singh, Richard T. Snodgrass, Anita K. Jones, Daniel P. Siewiorek, "An integrated instrumentation environment for multiprocessors", *IEEE Transactions on Computers*, Vol. C-32, No. 1, January 1983, pp. 4-14.

[36] Zary Segall and L. Rudolph, "PIE: a programming and instrumentation environment for parallel processing", *IEEE Software*, November 1985, pp. 22-37.

[37] Connie U. Smith. *Performance Engineering of Software Systems*. Reading, MA: Addison-Wesley, 1990.

[38] Richard T. Snodgrass, "A relational approach to monitoring complex systems", *ACM Transactions on Computer Systems*, Vol. 6, No. 2, May 1988, pp. 157-196.

[39] Cui-Qing Yang and Barton Miller, "Critical path analysis for the execution of parallel and distributed programs", *Proceedings 9th International Conference on Distributed Computing Systems*, June 1988, pp. 366-373.