Capturing and Automating Performance Diagnosis: the Poirot Approach

B. Robert Helm, Allen D. Malony, Stephen F. Fickas
University of Oregon

CIS-TR-95-02
February 1995

Abstract

Performance diagnosis, the process of finding and explaining performance problems, is an important part of parallel programming. Effective performance diagnosis requires that the programmer plan an appropriate method, and manage the experiments required by that method. This paper presents Poirot, an architecture to support performance diagnosis. It explains how the architecture helps automatically, adaptably plan and manage the diagnosis process. The paper evaluates the generality and practicality of Poirot, by reconstructing diagnosis methods found in several published performance tools.

Keywords: Performance tools, performance debugging, knowledge-based diagnosis, software engineering, parallel programming, scientific computing

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

1: Introduction

Our general goal is to make advances in parallel performance evaluation more useful for *performance debugging*. In performance debugging, the programmer writes an initial parallel version of a program, runs the program, analyzes performance data to find performance problems, and then transforms the program in response to those problems. We are particularly interested in the process of finding and explaining performance problems -- a process we call *performance diagnosis*. Performance diagnosis requires that one plan and carry out an effective *method* - a policy for setting up experimental program runs, collecting data, and analyzing and interpreting results. Programmers experienced with an application and its target environment can quickly isolate many performance bugs by careful choice of method -- of measurement tools, experimental conditions, and analysis techniques. Conversely, less experienced programmers can waste time and machine resources gathering data of low utility [15]. We thus believe that parallel performance research must identify appropriate diagnosis methods, and develop *automated, adaptable* support for applying those methods across classes of applications, architectures, and measurement environments.

This paper presents an architecture for planning and managing performance diagnosis. From the literature on performance diagnosis tools, we have identified common diagnosis methods, applicable across a wide range of languages, machines, and measurement environments. From the fields of expert systems, software process modeling, and semantic databases, we have developed techniques to automatically select and apply performance diagnosis methods. From this work we have derived *Poirot*, an architecture for performance diagnosis. This paper describes Poirot, and explains the support it provides to performance diagnosis. To suggest how Poirot can practically support both automation and adaptability, we show how it can *rationally reconstruct* (functionally reproduce) several automated performance diagnosis tools.

2: Approach

Performance diagnosis is a process of finding the main performance problems in a program, and explaining them sufficiently well to suggest program improvements. This task makes two kinds of demands on the programmer. First, performance diagnosis requires the programmer to *plan* -- pick performance experiments to do and analyses to perform. Second, the programmer must effectively *manage* performance diagnosis, setting up and running experiments, analyzing results, and organizing the numerous script, program, and data files generated along the way. Much of this management work is tedious, but must be done carefully to avoid incorrect or implausible results ("performance anomalies") due to mistakes in experiment setup or analysis [8].

Research has produced many performance tools that plan and manage significant parts of the performance diagnosis process. Tools like Quartz [1] and MTOOL [13], for instance prescribe particular kinds of experiments, automatically instrument programs to support those experiments, and manage most of the mechanics of data analysis. 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 "interesting" blocks in programs from an initial time profile. ChaosMon [6] automatically selects application-specific program visualizations to illustrate particular performance problems. ATExpert [18] automatically interprets performance data and offers the programmer "observations" on the likely causes of poor performance.

All of the tools listed above support performance diagnosis by prescribing and carrying out a particular performance diagnosis method -- a set of policies for experimentation and analysis. Performance tools gain several advantages by committing to a particular diagnosis method, and integrating that method tightly with particular performance analysis tools. First, they are highly *automated*. They can plan, making difficult decisions such as what to instrument in a program. They can also manage, relieving the programmer of many details of setting up, running, and analyzing performance experiments. However, by "hard-coding" a diagnosis method, and "hard-wiring" that method to particular analysis and measurement subsystems, performance tools sacrifice *adaptability*. No single diagnosis method is appropriate for all users, architectures, and applications. Certainly the choice of metrics to measure [9], and of techniques for data collection [6], [13] depends strongly on the target machine. Higher level choices, such as the type of behavior to investigate in an application [6], are equally dependent on the application and its target. Above all, the programmer's requirements for cost, accuracy, and precision in performance evaluation strongly influence the best choice of methods, and these requirements can vary considerably from project to project and moment to moment [23].

Unfortunately, many existing performance tools make it difficult to change or extend their diagnosis methods, or to combine their built-in analysis and measurement facilities with those of other tools. As a result, end users must do considerable work (re)coding tools, or converting data between tools, if existing tools do not fit the requirements of a particular performance diagnosis project. In reaction, some researchers have developed general, highly adaptable toolkits for performance measurement and analysis [25], [28]. However, these tools sacrifice automation; the user must plan the entire performance diagnosis process, and write lengthy

scripts to manage the process.

We conjecture that both adaptability and automation are required to gain acceptance of performance diagnosis tools by users. We propose two novel design principles to make automated, adaptable performance diagnosis possible:

- Design Principle 1:*Plan methods to suit.* Give the programmer a general, extensible catalog of automated diagnosis methods, and a simple way to assemble good methods for a problem them into a coherent plan.
- Design Principle 2: *Separate methods from tools.* Link diagnosis methods to diagnosis tools through a high-level functional interface, one that hides details of tools and data irrelevant to managing the diagnosis process.

Poirot is an architecture for performance diagnosis that implements these two principles, by synthesizing research from artificial intelligence, software engineering, and performance evaluation. In particular, to build custom diagnosis methods, it relies on research in knowledge-based systems, which has produced a library of formal, general methods for diagnosis [5], [7], [19], [27] and has created implementation frameworks that can assemble methods to suit a particular purpose [10], [11], [22]. To create a high-level, flexible interface to performance tools, we draw on research in software development environments [17], [20], [24] and software databases [2], [26], [29], which provide techniques to access tools, programs, and data, independent of tool command syntax and data format.

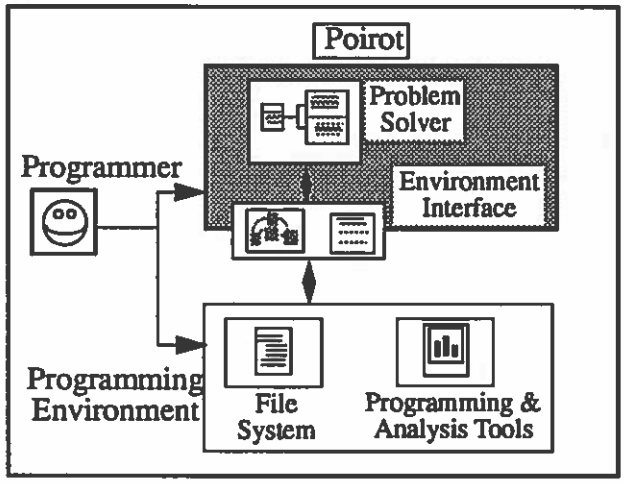
3: Architecture

In this section, we sketch the architecture of Poirot, and show how it can support automated, adaptable performance diagnosis by our two design principles, construction of custom methods, and separation of methods from tools. Figure 1 gives an overview of Poirot and its role in the programming process. Poirot consists of a *problem solver*, which constructs and executes custom performance diagnosis methods, and an *environment interface*, which links the problem solver to tool, programs, and data in the programmer's development environment. The architecture of Poirot is based on the Glitter program optimization system [11]. Like Glitter, Poirot acts as an assistant to the programmer. The programmer uses it to find useful diagnosis methods, to set up and carry out experiments for those methods, and to interpret and track the results of those experiments. While Poirot could conceivably operate autonomously, we expect that in most cases the programmer will perform the intellectually difficult tasks (such as interpreting performance analysis results) while Poirot performs the mundane ones.

3.1: Problem solver

Poirot's *problem solver* plans and manages diagnosis in conjunction with the programmer. It effectively implements Design Principle 1, planning diagnosis actions based on the current state of the diagnosis project and the user's preferences. The problem solver is a knowledge-based system; this simply means that it is structured around an interpreter (called the *engine*) which interprets a "program" called the *knowledge base* (Figure 2). The knowledge base is divided into two parts, the *method catalog* and the *control knowledge*.

Figure 1: The role of Poirot.



The method catalog is an indexed library of performance diagnosis techniques. Each method in the catalog is effectively a small program that accomplishes a particular performance diagnosis task.The task is called the method's *goal*. Each method has a *body* that gives a list of diagnosis actions for accomplishing its goal. The actions in a method body fall into two types:

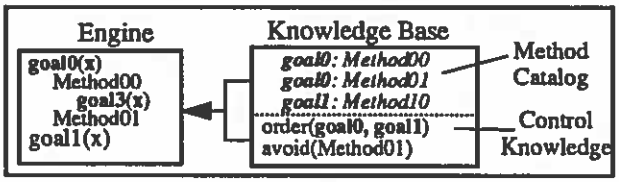
1. An action can start some subtask required to accomplish the method's goal (*post a subgoal*). This ultimately causes a method that can solve the subgoal to be called as a subroutine.
2. An action can send a command via Poirot's environment interface (*apply a transformation*). This is how Poirot can carry out low-level actions such as program instrumentation on behalf of the user. In some cases, applying a transformation will simply ask the user to supply some information or to take some action.

For example, the method catalog might contain the (high-level) method "Measure synchronization rate". The goal of this method is "**establish** that there is a synchronization bottleneck in routine R". Among its actions is one that posts the subgoal "**instrument** R for total time and synchronization

operation count"; this subgoal could lead to the invocation of a method that actually instrumented R, or one that retrieved an existing copy of R with the correct instrumentation. The method also includes actions that (1) run the instrumented program, (2) compute the synchronization rate of R, and (3) ask the programmer to judge whether the rate is excessive given the user's knowledge of R.

Note that more than one method may be defined for any given goal. Thus, when a subgoal is posted, there may be many candidate methods to invoke to address that goal. Poirot's problem solver has no fixed strategy for choosing methods to invoke; instead, it chooses methods based on its *control knowledge*. The control knowledge portion of the knowledge base defines policies for choosing methods, and ordering execution of goals, based on the current state of the diagnosis process. That state can include the user's current preferences, the target application and architecture, or the prior results of diagnosis. Control knowledge thus provides a mechanism for dynamically adapting general diagnosis methods from the catalog to the needs of a particular project. This is discussed in more detail in Section 4.

Figure 2: Problem solver components.

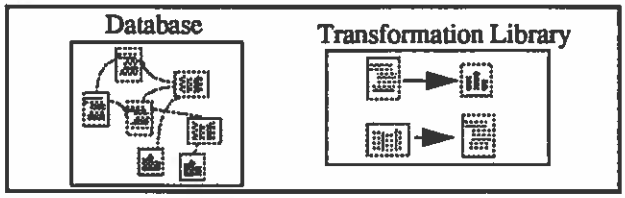


3.2: Environment Interface

The environment interface (Figure 3) provides high-level access to tools and data in the programming environment, in accord with Design Principle 2. The environment interface consists of a *transformation library*, which represents the tools, and a *database*, which represents the program and data files.

The transformation library is a set of primitives to be applied by performance diagnosis methods. Each transformation has an *interface* that characterizes its requirements and effects in terms of the database. Most transformations also have a *script*, that exchanges information with the user, or that sends commands to tools in the programming environment [12]. The script of a transformation encapsulates most environment-dependent details of a diagnosis action, such as the command syntax of the tools it invokes.

Figure 3: Environment interface components.



The database represents the state of the diagnosis project. It uses a uniform information model which represents such diverse items as data sets, programs, configurations files, and assertions about the target environment as *objects* with *attributes*. Transformations test and update the database as they are applied, allowing Poirot to track the state of the development environment as it changes. The user can also update the database, to document actions taken outside of Poirot.

To illustrate the functions of the environment interface, we return briefly to the method "Measure synchronization rate" discussed in section 3.1. That method leads to the application of several transformations, among them one that instruments a routine R for synchronization operation count and total time. The interface of this transformation requires that the database include an object representing routine R, and indicates that the transformation adds a new, instrumented version of R to the database. The transformation's script depends on the program environment and its supported tools; if, for example, the measurement tools use source instrumentation, the script might include version control and editing commands to add such instrumentation. However, if instrumentation were controlled by runtime argument flags, the script might do nothing, relying on a later transformation that runs the program to generate the necessary flags.

The example illustrates how the environment interface supports automated diagnosis while separating methods from tools (Design Principle 2). The environment interface defines a set of performance diagnosis primitives, specified in terms of their effects on a high-level database. The actual implementation of these primitives, and the format of data and programs in the database, is hidden from the diagnosis methods and so existing methods can be adapted unchanged to new programming environments and diagnosis tools. The result is that we can adapt knowledge about *what* steps to take in performance diagnosis into contexts where *how* those steps are taken differs significantly.

4: Example

Poirot is an architecture for automated, adaptable diagnosis using general diagnosis methods and a high-level interface to the programming environment. We next present a short hypothetical processing example using the architecture. The example serves two purposes. First, the example

shows in detail how the features of the architecture enable it to automate performance diagnosis while remaining adaptable. In addition, in section 5 we use the example to assess the practicality of adaptable performance diagnosis. In particular, we estimate the level of effort needed to functionally reproduce several existing performance diagnosis tools, starting from the knowledge base used in the example.

The programmer in our example is looking for performance problems in a neural net simulation program called `nnet`. Initially, the programmer decides to study the program's speedup on a typical data set. The programmer wants to examine the subroutines `init`, `pats`, and `train`, which correspond to the major phases of the program, and two subroutines of `train` -- `acts` and `wts` -- which contain the bulk of the program's computation. The first step in performance diagnosis with Poirot is to encode these decisions in Poirot's control knowledge. This entails translating the programmer's preferences into rules for selecting methods and ordering goals, illustrated by English paraphrases in Figure 4.

In Figure 4, the programmer first specifies the general method of performance diagnosis, by specifying the method to select for the **diagnose** goal. The method chosen, "Establish-Refine", is a generic diagnosis framework originally applied in a medical setting [4]. The Establish-Refine method treats performance diagnosis as search in a space of possible performance problems. Each point in the search space is called a *hypothesis*, and represents a performance problem as a *fault* (a type of performance problem that is occurring), and one or more relevant *components* (the program locations or phases where the problem is occurring). The Establish-Refine method consists of *establishing* and *refining* hypotheses. Establishing a hypothesis means finding evidence for the hypothesis; that is, finding evidence that a particular performance problem is having a significant impact on the program. Refining a hypothesis means generating the possible explanations for that hypothesis -- the possible causes (or more precise descriptions) of the performance problem. Each explanation generated is itself a hypothesis, which is then diagnosed recursively by the Establish-Refine method. Poirot's Establish-Refine method posts two types of subgoals, **establish** and **refine**, which initiate evidence-gathering and explanation generation, respectively.

Figure 4: Control knowledge for nnet example.

1. Use "Establish-Refine" method for **diagnose** goals
2. Key hypotheses have the form "Unknown fault in subroutine C", where C is one of {nnet, init, pats, train, acts, wts}.
3. Use "Speedup" method to **establish** key hypotheses
4. Use "RefineComponent" to refine key hypotheses.
5. Prefer methods that add measurements to previously planned experiments over methods that create new experiments.
6. Plan all experiments concerning key hypotheses before running any such experiments.

With this explanation, we can explain the effect of the remaining control rules in Figure 4. Rule 2 lists a set of key hypotheses; Rule 3 states that speedup analysis should be used to gather evidence for the key hypotheses. Rule 4 states that Poirot should try to refine the location of key performance problems before refining the type of fault occurring. Rules 5 and 6 implement a general policy for planning experiments: pack as many measurements into an experiment as possible. The effect of all these rules is to order a speedup profile of the program, with speedup analyses for the whole program and each of the key subroutines.

We briefly illustrate the operation of Poirot on the example problem. Poirot follows a process of goal refinement, guided by control knowledge. The user supplies an initial goal. The problem-solving engine then retrieves methods that are indexed to the goal. If more than one method is retrieved, the engine chooses one of the competing methods. The chosen method then executes its body, which (typically) applies some transformations and posts subgoals. The engine chooses one of the subgoals, and the cycle repeats. Each goal includes a test, which examines the diagnosis state to see whether the goal has been solved; the process terminates successfully when all goal tests evaluate to true. The choices in this process -- the choice of a goal to work on, and the choice of a method for a goal -- are made by consulting the control knowledge.

Figure 5 shows a trace of the goals and methods processed during the initial portion of scenario. In Figure 5, goals are **boldface**, the methods proposed for a goal are indented below the goal, and the subgoals posted by a method are indented below the method. An asterix marks methods chosen and goals solved during the scenario. The programmer initiates diagnosis by manually posting a goal **diagnose**(h0), where h0 is a hypothesis stating "There is an unspecified performance problem in the main program (`nnet`)". The engine retrieves methods for **diagnose** from the method catalog; Rule 1 causes it to choose "Establish-Refine". "Establish-Refine" posts its subgoals. We assume by default the **establish** goal is processed first; the engine retrieves two methods, "Speedup" and "TotalTime". Each method represents a way to gather evidence for performance

problems in nnet; use speedup analysis, or simply measure the total time of `nnet` and compare it to the programmer's expectations. Rule 3 causes the engine to select the "Speedup" method.

The "Speedup" method has four subgoals. The **plan_speedup** goal plans an experiment that measures the speedup of nnet. This could mean adding `nnet` to an already-planned speedup experiment; Rule 5 says to do this whenever possible. However, no speedup experiment has yet been planned, so the CreateSpeedup method is called to set up the experiment. This method posts an **apply** goal, which invokes a transformation createSpeedup. This transformation asks the programmer for parameters of the speedup experiment (such as numbers of processors, the version of the program), and adds a new object representing the experiment to the database. The second subgoal of "Speedup" is an **apply** goal that instruments the program used in the new experiment to measure `nnet`'s total run time. The remaining subgoals of "Speedup", **run_speedup** and **assess_speedup**, would run and present results of the speedup experiment to the user. However, Rule 6 defers these goals until the goal **refine**(h0) has been processed. This goal leads to the posting of **diagnose** goals for the subroutines `init`, `pats`, and `train`, initiating three recursive calls to Establish-Refine.

Figure 5: Goal-Method-Subgoal trace of example.

```
diagnose(h0="fault=unspecified, component=nnet") *
  Establish-Refine *
    establish(h0) *
      Speedup *
        plan_speedup(h0) *
          CreateSpeedup *
            apply(createSpeedup(h0)) *
        apply(instrumentTime(component(h0)))
        run_speedup(h0)
        assess_speedup(h0)
      TotalTime
    refine(h0)
      RefineFault
      RefineComponent *
        apply(findParts(component(h0))) *
        diagnose(fault=unspecified, component=init)
        diagnose(fault=unspecified, component=pats)
        diagnose(fault=unspecified,component=train)
```

The preceding scenario illustrates two features of the Poirot architecture. First, it can potentially make the diagnosis process highly automated. We observe that even if the programmer carried out all the steps corresponding to transformations, Poirot still provides some value organizing the diagnosis process. The goal/subgoal structure serves a form of "to-do" list, while the database keeps track of files and their functions in the process. This can help avoid slips such as omitting an instrumentation point, or comparing the performance of the wrong program versions. If most transformations have automated implementations, then Poirot can perform considerable amounts of work autonomously, guided only by the policies stated in the control knowledge.

Second, Poirot achieves automation adaptably, due to the two design principles it incorporates. In the scenario, Poirot followed a strategy (speedup analysis) with particular cost/accuracy/precision trade-offs. Poirot's method representation, and in particular its separation of methods from control knowledge, make it relatively easy to add methods or change control rules to set up other strategies achieving different trade-offs (Design Principle 1). For example, changing Rule 3 in Figure 4 to prefer "TotalTime" would produce an ordinary time profile, rather than a speedup profile, reducing the number of program runs while losing some useful information. Poirot also separates methods from the programming environment via the environment interface (Design Principle 2). As a result, most of the methods and transformations invoked in the example scenario could be adapted to other programming environments (or updated to take advantage of new facilities in an existing environment) by changing only the transformation scripts.

5: Rational reconstructions

The previous section showed that Poirot can diagnose performance automatically and adaptably. However, there are some practical obstacles. To support diagnosis in diverse contexts, numerous methods and control strategies must be encoded in the knowledge base, and numerous tools and file formats must be linked to the environment interface. We claim that Poirot can, in fact, be made practical, by reusing knowledge across multiple contexts. To demonstrate this, we informally assess how Poirot could *rationally reconstruct* several published performance diagnosis systems. In rational reconstruction, we show how Poirot can formally encode a system, mimic the problem-solving of that system on a well-defined external interface, and produce comparable results. If we can rationally reconstruct diverse systems without wholesale changes to the knowledge base and environment interface, this suggests that our approach may be made practical. One could develop a single, core version of Poirot, that a developer could incrementally modify for a particular set of requirements.

5.1: Performance Consultant

The first system we reconstruct is the Paradyne Performance Consultant [15]. Our goal is to show how Poirot's knowledge base could be extended to functionally reproduce the Performance Consultant's behavior. We note first that the Performance Consultant implements exactly the Establish-Refine method of diagnosis. As in our example, each hypothesis describes a performance problem in terms of a

fault type (called the "why" of the hypothesis in [15]), and a component where the fault is occurring. The Performance Consultant supports several types of hypothesis refinement, allowing components to be procedures, processes, or synchronization objects. The user can also specify a "when" coordinate for a hypothesis, corresponding to a time interval during program execution. Hypotheses are established by analyzing time histograms of key performance metrics, computed during the run of the program. Each hypothesis is associated with test code that enables relevant instrumentation points, collects and analyzes the histograms from those points, and judges the significance of the results. Hypotheses are ordered using stored "hints", and the hypothesis space is searched depth-first (a hypothesis is refined as soon as it is established).

We briefly outline the steps required for a developer to reconstruct the Performance Consultant in Poirot:

- Add a method for **establish** goals that evaluates hypotheses using time histograms. This method invokes the Paradyne instrumentation interface via transformations to collect and interpret on-line performance data.
- Add methods for **refine** goals that refine hypotheses to particular processes and synchronization objects. Also add a method for **refine** that interacts with the user (via a transformation) to specify "when" coordinates for hypotheses.
- Add rules to the control knowledge for depth-first search, on-line establishment of hypotheses, and any useful "hints".

We note first that none of the methods we discussed in section 4 need to be modified, although some (such as the "Speedup" method) are cut out of processing by the new control knowledge. The reconstruction reuses the "Establish-Refine" and "RefineComponent" methods, although the transformations applied by the latter method may need new scripts. Thus, by providing a catalog of general methods (Design Principle 1), and separating those methods from particular tool implementations, Poirot enables a developer to port the methods of the Performance Consultant to a different set of supporting tools.

5.2: ChaosMon

We also consider retargeting Poirot to integrate the distinctive features of the ChaosMon system [6]. In ChaosMon the user develops a *monitoring model*, essentially a set of application-specific hypotheses together with criteria for establishing those hypotheses. The criteria for testing a hypothesis are encoded in a corresponding *abstract view* that interprets performance data and becomes *active* when its hypothesis is established. When a view is active, ChaosMon displays one or more user-defined visualizations of the data that activated the view. Abstract views obtain their input data from *monitoring views*, high-level data collection programs that describe how to update abstract views from program variables during execution. Like Paradyne, ChaosMon diagnoses performance "on-line", during the program run. It provides a compiler that automatically generates optimized instrumentation code from view specifications.

We can sketch a process by which a developer could adapt Poirot to ChaosMon:

- Add a method for **refine** that queries the user for application-specific hypotheses and adds them to the database.
- Add a method for **establish** that checks the view for a hypothesis while the program is running. A subgoal of this method looks for a view definition for the hypothesis, invoking the editing tools and compiler via transformations if no definition yet exists.
- Add rules to the control knowledge that (1) **refine** all hypotheses to the greatest extent possible before the program is run, and (2) continually check the **establish** goal for each hypothesis during the program run, marking the goal solved if the view for the hypothesis becomes active during the run.

ChaosMon, like the Performance Consultant, reuses the "Establish-Refine" method. It also shares with the Performance Consultant some transformation interfaces concerned with on-line data sampling.

5.3: PTOPP.

Finally, we examine the system supported by the PTOPP (Practical Tools for Parallel Programming) tool suite [8], [9]. The system 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 [9]. It uses perturbation analysis, a generalization of speedup analysis, to detect performance problems. Finally, PTOPP provides extensive facilities for managing diagnosis, such as an automated mechanism for relating programs to the performance data they produced, and a database for storing that data.

A developer can represent the PTOPP system in Poirot as follows:

- Add a method for **refine** that support loops and loop nests as components.
- Define a perturbation method (a sibling of "Speedup" in Figure 5) for processing **establish** goals.

- Add control rules (similar to those in the Section 4 example) that initially establish which hypotheses correspond to the most time-consuming loops in the program. These become the key hypotheses, and are diagnosed before any other hypotheses.

Poirot takes over many of the management functions of PTOPP. It interprets the goal and hypothesis structure to relate programs to their associated performance data. The data stored by PTOPP are similar to those stored in the database in the example in Section 4.

5.4: Summary of reconstructions

Overall, the results of these cursory reconstructions are encouraging. We find substantial sharing and reuse of knowledge among the method catalogs of the three reconstructed systems. There is also some reuse of environment interface components among the three systems. Most of the effort in reconstructing the three systems is confined to the control knowledge, and the transformation implementations. A core knowledge base and environment interface might therefore suffice to make Poirot practically adaptable in diverse contexts.

6: Status and Future Work

We are currently implementing an initial version of Poirot, to confirm our initial impressions of the architecture with practical experience. Our near-term goal is to construct a knowledge base consisting of multiple performance diagnosis methods drawn from an extensive literature review [14]. We will test these methods 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-solver, to provide a core method catalog, 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 goal of this effort is to work with end users of pC++ to experimentally evaluate the level of automation and adaptability that can practically be achieved. The pC++ version of Poirot will finally be retargeted to another environment, to assess whether it can be cost-effectively adapted to multiple environments.

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 first task is to acquire a more complete picture of performance diagnosis as it is practiced. To date, our work has been based primarily on case studies supplied by tool developers. We are currently pursuing additional case studies from application developers [15].

Also, our research focuses not only on automation of performance diagnosis, but 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 [9] 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 [11].

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 a method that may be used to compare competing approaches. In addition, Poirot's ability to define methods independent of tool implementations suggests a new, "need-driven" [8] 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.

7: References

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

[2] D. G. Allard and D. S. Wile, "Aggregation, persistence and identity in worlds", in J. Rosenberg and D. 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), 1993, pp. 588-597.

[4] T. Bylander and S. Mittal, "CSRL: A language for classificatory problem-solving and uncertainty handling", *AI Magazine*, August, 1986, pp. 66-77.

[5] B. Chandrasekharan, and T. Johnson, "Generic tasks and task structures: history, critique, and new directions", in Jean-Marc David, Jean-Paul Krivine, Reid Simmons (eds.) *Second Generation Expert Systems*. Berlin: Springer-Verlag, 1992, pp. 232-272.

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

[7] J. de Kleer, B. Williams, "Diagnosing multiple faults", *Artificial Intelligence*32, 1987, pp. 97-130.

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

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

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

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

[12] M. A. Gisi and G. 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.

[13] A. J. Goldberg and J. 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.

[14] 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.

[15] A. D. Malony, B. Robert Helm, "Call for collaboration: performance diagnosis processes", Technical Report 95-01, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403.

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

[17] K. Huff and V. 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.

[18] J. Kohn and W. Williams, "ATExpert", Jounral of Parallel and Distributed Computing, Vol. 18, 1993, pp. 205-222.

[19] 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.

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

[21] C. Kilpatrick and K. 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.

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

[23] P. Messina, T. Sterling (eds.). *System Software and Tools for High Performance Computiung Environments.* Philadelphia, PA: SIAM, 1993.

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

[25] B. Mohr, "Performance evaluation of parallel programs in parallel and distributed systems" In: *Proc. COMPAR Joint Int'l Conf. on Vector and Parallel Processing* (Zurich, Switzerland, 1990), Lecture Notes in Computer Science 457. Berlin: Springer-Verlag, 1990, pp. 176-187.

[26] D. Ogle, K. Schwan, and R. 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.

[27] Y. Peng and J. 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.

[28] D. 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.

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