Automating Performance Diagnosis: a Theory and Architecture

B. Robert Helm

CIS-TR-95-09
March 1995

Abstract

This prospectus describes research in the field of software engineering to simplify programming of parallel computers. It focuses specifically on *performance diagnosis*, the process of finding and explaining sources of inefficiency in parallel programs. Considerable research already has been done to simplify performance diagnosis, but with mixed success. Two elements are missing from existing research:

1.There is no general theory of how expert programmers do performance diagnosis. As a result, it is difficult for researchers to compare existing work or fit their work to programmers. It is difficult for programmers to locate products of existing research that meet their needs.

2. There is no automated, adaptable software to help programmers do performance diagnosis. Existing software is either automated but limited to very specific circumstances, or is general but not is automated for most tasks.

The research described here addresses both of these issues. The research will develop and validate a theory of performance diagnosis, based on general models on diagnostic problem-solving. It will design and evaluate a computer program (called *Poirot*) that employs the theory to automatically, adaptably support performance diagnosis.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

1.0 Introduction

The research outlined here is in the field of software engineering. Research in software engineering is similar to research in manufacturing engineering, in that it studies and tries to improve production processes -- the processes that produce computer software. My research has focused on software for distributed [10], [12], [13] and parallel [17] computer systems. Both parallel and distributed computer systems consist of multiple computers (and computer programs) that cooperate to some end. In distributed systems, the computers cooperate over long distances and with great autonomy. In parallel systems, the computers (called *processors* in this context) cooperate closely, generally executing different parts of a single program.

One reason to solve a problem with multiple processors, rather than a single processor, is to get high *performance*. The performance of a computer system is the speed with which it returns answers, and the efficiency with which it uses computer resources such as computing time and storage space. This research is motivated by an application that demands particularly high performance from computers, namely computational science. Scientists in several fields have identified "grand challenge" problems, whose solution depends in part on improvements in computing performance [27]. Parallel computers and software are being intensively studied as a way to get the necessary performance. In scientific parallel programs, which have severe performance requirements, "performance bugs" -- programming mistakes that make a program slow or inefficient -- attract a good deal of attention. Consequently, the job of the parallel programmer includes *performance debugging* -- finding and repairing performance bugs.

This paper describes research to understand and improve the process of performance debugging in parallel scientific programs. The research focuses specifically on the problem of locating and explaining performance bugs, which I call *performance diagnosis*. Expert parallel programmers often improve program performance enormously by running their programs experimentally on a parallel computer, then interpreting the results of these experiments to suggest changes to the program [9], [14]. Researchers in parallel computing have developed integrated suites of computer programs (called *performance diagnosis systems* in this paper) to collect and analyze performance data from performance experiments. However, performance diagnosis systems are not extensively used. Many obstacles prevent performance diagnosis systems from escaping research into practice [23], [28]. Two obstacles in particular motivate my current work:

1. *Lack of theoretical justification.* Researchers lack a theory of what methods work, and why. There is no formal way to describe or compare how expert programmers solve their performance diagnosis problems in particular contexts. There is no standard theory for understanding diagnosis system features and fitting them to the programmer's particular needs. As a result, researchers cannot easily compare and evaluate the systems they produce, and many potential users do not find systems that are applicable to their performance diagnosis problems.

2. *Conflict between automation and adaptability.* Performance diagnosis systems are not easily adaptable to new requirements. Highly automated systems, while providing considerable help to the programmer, are hard to change, hard to extend, and hard to combine with other systems [1]. As a result, programmers must do considerable work (re)programming systems, or converting data between systems, if existing automated systems do not fit their requirements. In fact, programmers generally ignore systems that do not fit their needs exactly. Instead, they collect and analyze data manually [28], or construct custom systems for their own projects [8].

Both of the obstacles above, I believe, could be mitigated by a formal theory of performance diagnosis processes. Such a theory should be able to describe how programmers solve problems, and how diagnosis systems help or hinder them. The theory could be used by researchers to systematically compare and evaluate performance diagnosis systems. In addition, the theory could be used to create automated performance diagnosis systems that are adaptable to particular requirements.

The research discussed here will make two contributions to the fields of software engineering and parallel computing. First, it will develop and evaluate a formal theory of parallel performance diagnosis, by applying models of diagnosis developed in the field of artificial intelligence. Second, it will develop and evaluate a novel diagnosis system that is both automated and adaptable. This will be a first step toward more usable, effective diagnosis systems.

1.1 Problem definition.

I first present a hypothetical example of scientific programming on a parallel computer. This will allow me to clearly define the research problem, and to give the reader enough information about parallel programming to understand issues that arise later. The programmer in the example must write a program that correctly and efficiently simulates an interconnected system of brain cells (a *neural net*) behaving according to some scientific theory. Large neural nets must be simulated over many different experimental conditions, so the programmer turns to a parallel computer. To translate the scientific theory into a parallel program, the programmer must make numerous decisions. In the neural net example, for instance, the programmer must decide:

- How should the state of the neural net be represented as numbers?
- How should the neural net be divided up for simulation by processors?
- What protocol should processors follow to obtain simulation work?

Performance diagnosis guides the programmer to bad decisions made during programming. By finding and explaining the chief performance problems of the program, the programmer determines which decisions had the worst performance effects, and how those effects might be repaired. As an example of performance diagnosis, suppose that the programmer is studying the performance of the neural net program. The programmer runs the program, and finds

that processors are idle much of the time. The programmer forms a hypothesis: there is a *load imbalance*. Some processors have more simulation work than others, so some processors are wasting time waiting for the overloaded processors to finish. This does not tell the programmer which decision should be changed, however. The programmer hypothesizes that the load imbalance is caused by the protocol that processors follow to obtain simulation work. This hypothesis implicates a particular decision -- the choice of work distribution protocol -- and thus constitutes a useful diagnosis if it can be confirmed. The programmer changes the protocol and observes significantly improved load balance, confirming the hypothesis.

Given this example, I now (re)define some terms. During performance diagnosis, the programmer decided which performance data to collect, which features to judge significant, which hypotheses to pursue, and what confirmation to seek. I define a *performance diagnosis method* to be the policies used to make such decisions. In these terms, a *performance diagnosis system* is a suite of programs that automatically supports some diagnosis method. The research problem is to define a theory of performance diagnosis methods, and to use that theory to create more automated, adaptable performance diagnosis systems.

1.2 Research approach and thesis statement.

To develop a theory of performance diagnosis, I have turned to problem-solving models from the field of artificial intelligence. Specifically, I have turned to the model of heuristic classification [5]. The first claim of the research is that a theory based on heuristic classification can effectively explain existing performance diagnosis systems and the way they are used.The theory has been used to characterize published case studies of performance diagnosis, providing initial validation of this claim.

Work in artificial intelligence has also focused on techniques to create adaptable, effective problem-solving systems [4], [22]. I have synthesized and extended several of these techniques in the design of a novel performance diagnosis system called Poirot. Poirot will encode, in a computer-interpretable form, the theory of performance diagnosis that this research develops. This will allow Poirot to automate many aspects of performance diagnosis, while remaining adaptable to changes in parallel computer, program, or programmer. I have initially validated this claim, by showing how Poirot could *rationally reconstruct* or reproduce a set of existing performance diagnosis systems.

The thesis statement below summarizes the research approach:

> A classification model of problem-solving is a sound basis for a theory of performance diagnosis, and for an automated, adaptable performance diagnosis system.

This paper describes current results, remaining work, and future work in support of this thesis. Section 2 presents current results and remaining work on the theory of performance diagnosis. Section 3 does the same for the diagnosis system, Poirot. Section 4 discusses future work on both of these topics.

2.0 A theory of performance diagnosis

To attack the first obstacle to performance diagnosis systems, lack of theoretical justification, this research develops a "knowledge-level" theory of performance diagnosis [26]. A knowledge-level theory of performance diagnosis must answer the question, "What knowledge does a programmer use to choose actions to meet performance diagnosis goals?" The theory here breaks the question down into two parts: (1) What methods do expert programmers use? (2) How can we rationalize the programmer's choice of methods? I have reconstructed or "reverse engineered" some answers to these questions from a survey of research papers on performance diagnosis systems, and from the case studies that appeared in those papers. The goal was to find methods and rationale that cut across a substantial number of diagnosis systems. Each performance diagnosis system was viewed as a collection of methods for system for heuristic classification. Similarities among systems were analyzed to identify general methods, and differences among systems were studied to extract rationale.

The result of the survey is a rationalized taxonomy of performance diagnosis systems, a systematic description of what methods performance diagnosis systems use, and why they use them. This section explains how heuristic classification was used to generate this performance diagnosis theory, summarizes the theory, and identifies remaining work to validate the theory.

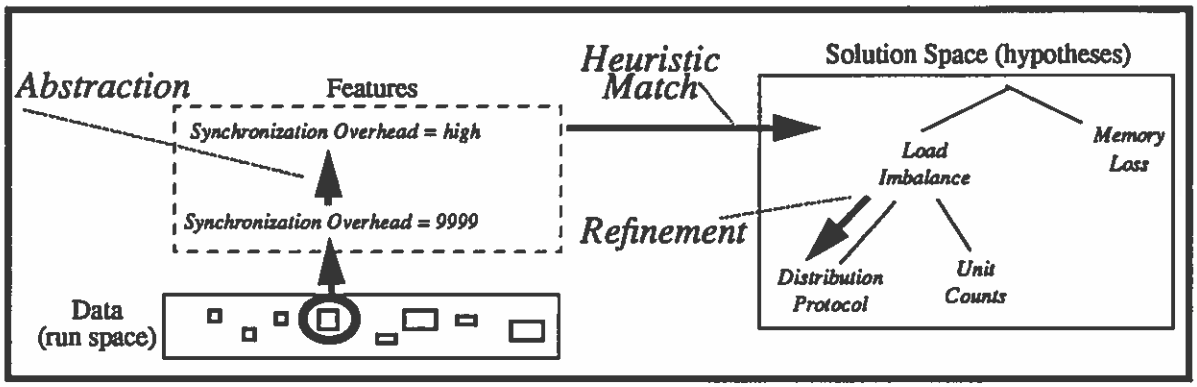
2.1 Heuristic classification in performance diagnosis

Heuristic classification is a way to solve problems by matching them to previously stored solutions. Clancey [5] identified heuristic classification as a critical process in many *expert systems*, computer programs that solve knowledge-intensive problems. Figure 1 depicts heuristic classification applied to performance diagnosis. Heuristic classification solves problems by looking up a solution in an exhaustive *solution space*. In the central step of heuristic classification, *heuristic match*, the problem solver matches the problem at hand (the *case)* to a stored solution in the solution space, based on the cases's essential aspects or *features*. For diagnosis, the solution space is the set of all *hypotheses* that could explain observed performance. The features of the case are extracted from raw information (*data*) by a process called *abstraction*. The solution selected by heuristic match is *refined* to fit additional features of the case. Abstraction, heuristic match, and refinement do not need to occur in any fixed order. The way the three processes are ordered or combined constitutes the *strategy* of the problem-solver.

One can see the basic elements of the heuristic classification model -- heuristic match, abstraction, refinement, and strategy in the example scenario of section 1.1. In that scenario, the programmer *heuristically matched* the essential *features* of the program's behavior to a well-known class of performance problem (load imbalance) that could explain those features. The features were *abstracted* from the *run space* of the neural net program, the space of performance data from all possible program runs. Abstraction involved computing summary data (such as total idle time) from a typical run. The generic hypothesis, load imbalance, was

refined into a more detailed explanation of the program's behavior (the poor work distribution protocol). The programmer's *strategy* was to perform abstraction, arrive at an initial hypotheses, refine that hypothesis, and then do diagnosis to confirm that hypothesis.

FIGURE 1. Heuristic classification in performance diagnosis.



2.2 Theory summary.

Table 1 lists a set of performance diagnosis systems that were re-examined as heuristic classification systems. Tables 2 through 5 summarize the methods identified for each element of heuristic classification. The most interesting point simply that many methods are shared across systems. A relatively large sample of performance diagnosis systems produces relatively few methods, if one examines how performance diagnosis systems do heuristic classification, as opposed to how they and their supporting software are implemented.

TABLE 1. Performance diagnosis systems surveyed.

System	Paper	System	Paper
AIMS	[33]	Paragraph	[15]
ATExpert	[19]	PPP (Parallel Performance Predicates)	[6]
ChaosMon	[20]	PTOPP	[8]
IPS-2	[24]	Quartz	[2]
MTOOL	[14]	SPT	[31]
Paradyne (Performance Consultant)	[18]		

The survey attempted to discover the rationale for diagnosis systems, the design goals that why explain why a diagnosis system adopts a particular method. As an example, I note that the goals on which systems disagreed most were precision, accuracy, and cost. In performance diagnosis, precise performance data allow one to more exactly and quickly check hypotheses.

However, data can be too precise; collecting data alters the behavior of the program being diagnosed, so excessive data collection leads to inaccurate results. Excessive precision can also impose large costs and turnaround times for data analysis and presentation, which may slow the diagnosis process overall. The part of heuristic classification most affected by these considerations was abstraction, particularly abstraction of time from runs. Systems whose designers were intolerant of cost and inaccuracy used time-summarized measures of program performance, while systems concerned with precision provided displays exhibiting behavior across time. A middle ground was provided by systems such as ChaosMon and AIMS. Those systems summarized behavior over time except when a hypothesis was matched in heuristic match; those systems would then enable collection of data for temporally distributed displays

TABLE 2. Hypothesis spaces.

Heuristic Match
- What is the hypothesis (solution) space?
  - What is the fault taxonomy?
    - Programming model/architecture-determined faults (ATExpert, MTOOL, PPP, Paradyne, Paragraph, PTOPP, Quartz, SPT)
    - Algorithm-specific faults (ChaosMon, Quartz, PPP)
    - Not specified/Unclear (IPS-2)
  - What is the component structure?
    - Invocation structure (AIMS, ATExpert, IPS-2, MTOOL, Paradyne, PTOPP, Quartz, SPT)
    - Process/thread structure (AIMS, IPS-2, MTOOL, PPP, Paradyne, Paragraph, Quartz)
    - Phases (AIMS, ChaosMon, IPS-2, Paragraph)

TABLE 3. Methods of hypothesis retrieval.

Heuristic Match
- How are features used to retrieve the most significant hypothesis?
  - Direct measurement (all).
  - Perturbation (PTOPP, SPT).
  - Conflict detection (ATExpert, MTOOL).

TABLE 4. Methods of abstraction.

Abstraction
- How are features abstracted from individual runs?
  - Profiles (ATExpert, IPS-2, MTOOL, PPP, Paradyne, PTOPP, Quartz)
  - Time histograms (AIMS, ChaosMon, IPS-2, Paradyne, Paragraph)
  - Trace displays (AIMS, ChaosMon, IPS-2, Paragraph)
- How are features abstracted across runs?
  - Speedup profiles (MTOOL, PPP, PTOPP)
  - Response metrics (PTOPP, SPT)

The survey also shed some light on the lack of adaptability of performance diagnosis systems, by analysis of the sources of knowledge in those systems. For example, consider the solution space required by heuristic classification. How can it be practically enumerated for perfor-

mance diagnosis, when, in principle, every performance bug in every possible parallel program must be covered? The surveyed diagnosis systems solve this problem by restricting their hypothesis spaces. In particular, most systems deal only with types of performance problems that are characteristic of a particular programming language and parallel computer architecture. Only a few systems attempt to systematically support program-specific hypotheses, and only ChaosMon and PPP support custom or user-defined hypotheses. This observation explains the lack of adaptability of many diagnosis systems -- in order to have a small hypothesis space, they commit to a limited target computer and language.

TABLE 5. Methods of refinement.

Refinement
How are hypotheses explained by program behavior at the appropriate level of system description?
- Source analysis (AIMS, ATExpert, IPS-2, MTOOL, PTOPP, Quartz, SPT)
- Syncrhonized trace analysis (AIMS, Paragraph)
- Linked explanations (AIMS, ATExpert, ChaosMon, Paradyn, Paragraph)

TABLE 6. Characterizing strategies.

Strategy
To what extent is diagnosis data-driven (vs. hypothesis-driven)?
- Data-driven (ATExpert, IPS-2, PPP, PTOPP, Paragraph, Quartz)
- Phased (MTOOL, SPT)
- Predicate (AIMS, ChaosMon)
- Hypothesis-driven (Paradyne)
To what extent is diagnosis performed online (vs. offline)?
- Offline (AIMS, IPS-2, Paragraph)
- Online abstraction, offline match (ATExpert, MTOOL, PPP, PTOPP, Quartz, SPT)
- Online predicate (AIMS)
- Online (ChaosMon, Paradyne)

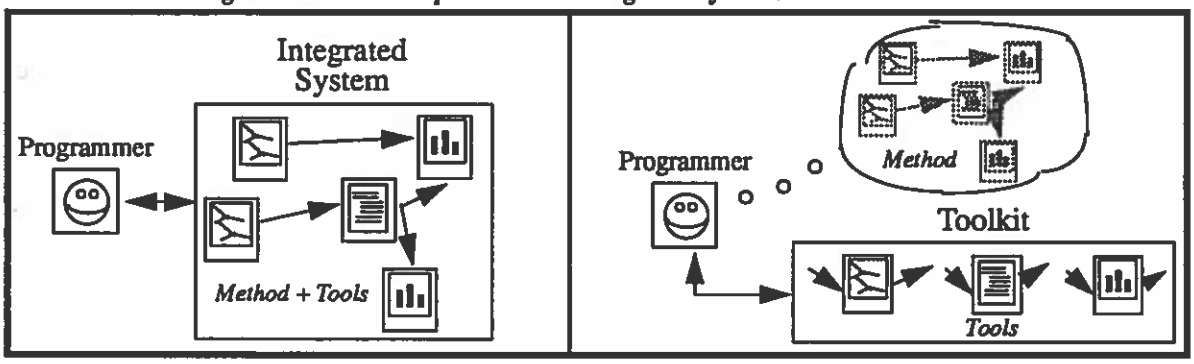
2.3 Remaining work

The theory provides novel framework for characterizing and comparing performance diagnosis systems, and systematically organizes the knowledge that performance diagnosis systems use. However, the theory is based on case studies reported by performance diagnosis researchers, rather than directly by scientific programmers. These case studies may not accurately represent scientific programming problems. I am currently seeking additional case studies from developers of scientific applications [21]. Also, the theory developed so far is insufficiently formal to make testable predictions about performance diagnosis processes. My approach to formalizing and validating the theory is to build it into a computational model -- the Poirot diagnosis system -- and demonstrate that the model can competently solve performance diagnosis problems. The next section describes Poirot and the work intended for it.

3.0 Poirot: an architecture for performance diagnosis

In this section, drawn partially from [17], I sketch Poirot, a performance diagnosis system based on the theory discussed above. Poirot is designed to overcome the second obstacle to acceptance of diagnosis systems: their poor combination of automation and adaptability. Existing performance diagnosis systems go to one of two extremes (Figure 2). *Integrated* systems aim for high automation by committing to a particular performance diagnosis method, and to programs for performance data collection, analysis, and presentation (*tools*) that are specialized for that method. However, as discussed in section 2.2, the method and tools are unlikely to be effective outside of a narrow range of parallel computers, programs, and programmers; integrated diagnosis systems are therefore not adaptable. In reaction, some researchers have developed *toolkits*, which supply a general set of tools, and a programming system for combining tools [25], [30]. However, toolkits sacrifice automation; the programmer must decide on a performance diagnosis method and write additional programs to carry it out.

FIGURE 2. Integrated and toolkit performance diagnosis systems.



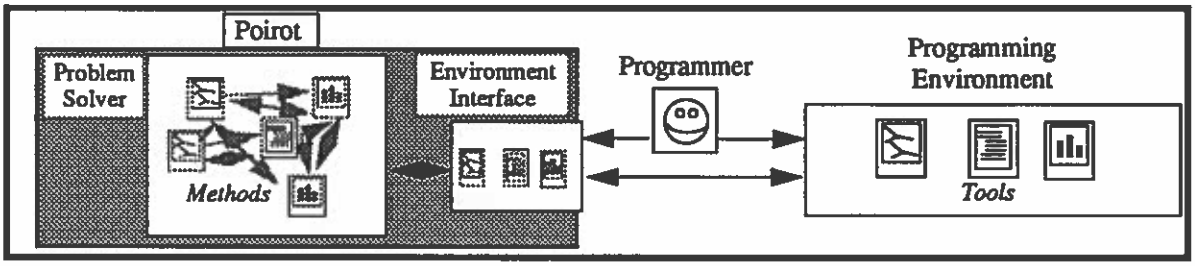
Poirot offers a third alternative (Figure 3). The programmer using Poirot neither accepts a canned method, as in integrated systems, nor builds a custom method from scratch, as in toolkits. Instead, the programmer defines policies, which are interpreted by Poirot's *problem-solver* to choose a performance diagnosis method. The programmer also helps Poirot use whatever tools are available, by extending Poirot's *environment interface*. This section explains how Poirot works, presents evidence of its feasibility, and identifies additional work to implement and evaluate it.

3.1 Overview of Poirot

Poirot is an extension and redesign of the Glitter system [11]. It consists of a *problem solver*, and an *environment interface*. The problem-solver selects and carries out performance diagnosis actions. It is guided by a formal encoding of the theory of performance diagnosis, which supplies multiple performance diagnosis methods and their rationale. The problem-solver

does not, however, interact with tools directly. Instead, it performs abstract diagnosis actions that are translated into commands by the environment interface.

FIGURE 3. Overview of Poirot.



The problem solver is the most novel component of Poirot. It is a knowledge-based system; which means that it is structured around a program called the *engine*, which carries out instructions in a *knowledge base*. The knowledge base is divided into two parts, the *method catalog* and the *control knowledge*. The method catalog encodes methods from the theory of performance diagnosis. The control knowledge encodes the rationale of these methods, along with the programmer's preferences for method selection.

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*).
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 (called *transformations*) 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.

The engine of Poirot chooses and executes methods. The programmer supplies an initial diagnosis goal, which represents some diagnosis task to perform. The engine then retrieves methods indexed to that goal, selecting among alternative methods based on the control knowledge. The engine carries out the actions of the method's body in sequence. If actions post subgoals, the engine chooses one of the subgoals (again using control knowledge), retrieves methods indexed to that subgoal, and repeats the cycle. The engine halts when all goals have been accomplished. It interrupts the cycle when it cannot proceed due to a gap in the knowledge base or environment interface. In such cases, the programmer fills the gap, by supplying missing information or by performing some diagnosis action manually.

Diagnosis actions send commands to tools via the environment interface. The environment interface consists of a set of *transformations* that represent primitive diagnosis actions, and a

database that represents stored performance data and program versions. The purpose of the environment interface is to support adaptable diagnosis, by separating diagnosis methods from the software that support those methods. It specifies transformations in terms of their effects on the high-level database. Methods can thus apply transformations and track their effects without knowing what commands are sent to tools, or how data and programs are stored in files. As a result, general methods can be adapted unchanged to new tools. One can reuse knowledge about *what* steps to take in performance diagnosis in contexts where *how* those steps are taken differs significantly.

3.2 An example

I briefly illustrate the operation of Poirot on an example. The programmer in the example is looking for performance problems in the neural net simulation program, which is called `nnet`. The programmer has added information on available tools to the environment interface, and specified the control knowledge as a set of rules for selecting goals and methods (Figure 4). Rule 1 selects the overall method for performance diagnosis, an implementation of heuristic classification called "Establish-Refine" [3]. The Establish-Refine method has two subgoals, **establish** (establish a hypothesis) and **refine** (refine a hypothesis). Establishing a hypothesis means finding evidence for the hypothesis; in terms of heuristic classification, this means doing abstraction and heuristic match to check whether the hypothesis is valid for the program. Refining a hypothesis, as in the heuristic classification model, means generating the possible explanations for that hypothesis. Each explanation generated is itself a hypothesis, which is then processed in its turn by the Establish-Refine method.

FIGURE 4. Control knowledge for neural net 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. Add measurements to previously planned experiments rather than create new experiments.
5. Plan all experiments concerning key hypotheses before running any such experiments.

Figure 5 shows a trace of the goals and methods processed during the initial part of a diagnosis session. 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 selects "Establish-Refine", which posts its subgoals. The **establish** subgoal is processed first; the engine retrieves two methods, "Speedup" and "TotalTime". Each method represents a way to gather evidence for performance problems in nnet; measure its run time with increasing numbers of processors, or simply measure its run time for comparison to the programmer's expectations. Rule 3 causes the engine to select the "Speedup" method. That method sets up an experiment

to measure the speedup of the program as a whole. The experiment does not run, however. Rule 5 defers the experiment until the goal **refine**(h0) has been processed. This goal leads to the posting of **diagnose** goals for the subprograms `init`, `pats`, and `train`, initiating Establish-Refine three more times (not shown). The effect is to add the three subprograms to the speedup experiment. Finally, the experiment runs and the results are presented to the user.

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 Poirot. First, Poirot can potentially make the diagnosis process highly automated. Even if the programmer carried out all the steps corresponding to transformations, Poirot still helps organize the diagnosis process; the goal/subgoal structure serves a form of "to-do" list, while the database keeps track of performance data and their functions in the diagnosis process. If most transformations have automated implementations, then Poirot can perform works autonomously, guided only by the control knowledge.In addition, Poirot achieves automation adaptably. The programmer can change Poirot's diagnosis method relatively easily, by changing control rules. Poirot separates methods from the tools via the environment interface, so most of the methods in the example scenario could be adapted to other tools by changing only the innards of the transformations.

3.3 Feasibility

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. I claim that Poirot can, in fact, be made practical, by reusing knowledge across multiple contexts. To demonstrate this, one can assess how Poirot could *rationally reconstruct* several published performance diagnosis systems. In rational reconstruction, one shows 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 Poirot can rationally reconstruct diverse systems without wholesale changes to the knowledge base and environment interface, this suggests that it may be made practical. One could develop a single, core version of Poirot, that a programmer could incrementally modify for a particular set of requirements.

Table 7 summarizes the changes a programmer would have to make to Poirot to implement three performance diagnosis systems that appeared in section 2.2. The table assumes a version of Poirot that could automatically perform the scenario of section 3.2. To summarize, the system Paradyne implements exactly the Establish-Refine method of diagnosis, with refinement to several kinds of program components and phases. Both Paradyne and ChaosMon establish hypotheses on-line, during a run of the program, they differ only in that ChaosMon is continually attempting to establish all hypotheses in the hypothesis space, while Paradyne establishes only selected hypotheses. PTOPP differs significantly from these two systems, but still reuses the "Establish-Refine" and "Time Profile" methods from the example scenario.

TABLE 7. Reconstruction of three diagnosis systems.

Paradyne	**ChaosMon**	**PTOPP**
Add method for **establish** that evaluates hypotheses on-line. Add transformations to collect and interpret on-line time histograms.	Add method for **establish** goals that evaluates hypotheses on-line. Add transformations to collect and interpret on-line metric histograms.	Add method for **establish** that uses perturbation. Add transformations for PTOPP's perturbations.
Add methods for **refine** that refine hypotheses to processes, synchronization objects, and phases.	Add a method for **refine** that queries the user for application-specific hypotheses and adds them to the database.	Add methods for **refine** that refine hypotheses to loops.
Add control rules for depth-first search, on-line **establish**, Paradyne "hints".	Add control rules for exhaustive search, on-line **establish**.	Add control rules for initial **establish** with time profiles. High-scoring loops become key hypotheses as in example scenario.

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

3.4 Summary and remaining work.

Poirot's is a novel diagnosis system, although it is grounded in previous work in parallel processing and artificial intelligence. Poirot will be developed into a working program, to validate its design and the theory of performance diagnosis that underlies it. This validation will take place in two phases. In phase 1, Poirot will be used to diagnose multiple test programs for

two different platforms (parallel computers and programming languages). The test programs will have known performance bugs on the two platforms; the task of Poirot will be to find these bugs. Poirot will operate as an advisor, suggesting diagnosis actions to the experimenter, who will carry out the suggested actions and report results. During phase 1, Poirot will be evaluated on two dimensions:

1. *Competence.* How useful is the guidance Poirot provides to the experimenter? How much control knowledge must be added to find performance bugs autonomously?
2. *Cost.* How much knowledge must be added to Poirot to get competence on a new platform, given that Poirot has achieved competence on a previous platform?

In phase 2, the experiments of phase 1 will be repeated, but Poirot will now interact directly with data collection and analysis tools. In this phase, Poirot will be judged on slightly different criteria:

1. *Cost.* How much must be added to the environment interface of Poirot to link it to a platform? How much of the interface must be changed when Poirot is moved to a new system?
2. *Automation.* How many manual steps can Poirot replace, given a particular environment interface? How does Poirot's time-to-solution and error rate compare with manual operation?

High competence and automation in these experiments will demonstrate the validity of the theory of performance diagnosis Poirot embodies. If that competence and automation can be obtained at a reasonable cost, this will demonstrate Poirot's practical adaptability and validate its design.

4.0 Conclusions and Future Work

This research is developing a novel, knowledge-level theory of expert performance diagnosis. It will validate that theory both by further examination of case studies, and by direct testing of Poirot, a computational model. Poirot is also a performance diagnosis system, constructed on novel principles. The research will evaluate the ability of Poirot to competently and cost-effectively automate performance diagnosis in contexts more diverse than existing performance diagnosis systems.

The anticipated results have obvious limits that future work should target. There are at least two points on which the theory of performance diagnosis outlined here fails. Such failures do not render the theory valueless: it is incomplete, but it is sufficiently detailed and explicit that one can say how it is incomplete. The first failure concerns clustering of components. In Clancey's original framework, the hypothesis space is pre-enumerated both in the sense that it is entirely determined before reasoning begins, and in the sense that it is not case-specific. Neither statement is true for performance diagnosis. In one case study, for example [14], a programmer notes that certain subprograms are sufficiently alike in structure and behavior,

that the most time-intensive member of the set can be diagnosed, and the others assumed to operate similarly. This represents a clustering of the solution space, lumping multiple hypotheses into a single hypothesis based on the facts of the specific case. As such, it does not fit the model of heuristic classification that underlies the present theory.

In addition, many diagnoses concluded in case studies appear difficult to anticipate and store in advance, as required by the theory. For example, in another case study [32], the programmer identifies unexpected delays in a sequence of program events. The key word is "unexpected" -- the programmer has expectations of how the sequence of events should play out, and those expectations are specific to the case (program). With the exception of ChaosMon, none of the surveyed systems provide systematic support for detecting violations of case-specific expectations of behavior, and even the facilities of ChaosMon provide limited support for checking sequences of events. Performance diagnosis research may have to recapitulate diagnosis research in AI, and turn increasingly to "first principles" models of structure and behavior [7].

Future research will extend the theory of performance diagnosis to fill these gaps. Future work is also needed to extend the theory beyond performance bugs in parallel systems to similar bugs in distributed systems [16], and to validate the theory by direct observational studies of working programmers. With such a theory in hand, researchers may be able to close the gap between current performance diagnosis systems and their potential users.

5.0 References

[1] M. Abrams, A. Batongbacal, R. Ribler, D. Vazirani, "Chitra94: A tool to dynamically characterize ensembles of traces for input data modeling and output analysis", Technical Report TR 94-21, Department of Computer Science, Virginia Polytechnic Institute and State University, 1994.

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

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

[4] 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.

[5] W. J. Clancey, "Heuristic classification", *Artificial Intelligence* 27 (1985), 289-350.

[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] M.S. Feather, S. Fickas., B. R. Helm, "Composite system design: the good news and the bad news", Proceedings of Fourth Annual KBSE Conference, Syracuse, 1991

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

[12] S. Fickas, B. R. Helm, "Automating Specification of Network Applications", *IFIP International Conference on Upper Layer Protocols, Architectures and Applications*, May, 1992, Vancouver, B.C.

[13] S. Fickas, B. R. Helm, "Knowledge representation and reasoning in the design of composite systems", *IEEE Transactions on Software Engineering*, Vol. 18, No. 6, pp. 470-481.

[14] 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.

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

[16] B. R. Helm, S. Fickas, "Scare Tactics: evaluating problem decompositions using failure scenarios", *Proceedings of the Workshop on Change of Representation and Problem Reformulation* (Asilomar, CA, April 1992). Technical Report FIA-92-06, NASA Ames Research Center, Moffett Field, CA 94025, p. p. 85-93.

[17] B. R. Helm, A. D. Malony, S. Fickas, "Capturing and automating performance diagnosis: the Poirot approach", *Proceedings of the Ninth International Parallel Processing Synposium* (Santa Barbara, CA, April 1995). Los Alamitos: IEEE Computer Society Press, to appear.

[18] J. K. Hollingsworth. *Finding Bottlenecks in Large Scale Parallel Programs*. Ph. D. Thesis, Department of Computer Sciences, University of Wisconsin-Madison, 1994.

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

[20] 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.

[21] 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.

[22] J. McDermott, "Preliminary steps toward a taxonomy of problem-solving methods", in S. Marcus (ed.), *Automating Knowledge Acquisition for Expert Systems*. Boston, MA: Kluwer, 1985.

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

[24] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, T. 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.

[25] B. Mohr, "Standardization of event traces considered harmful or is an implementation of object-independent event trace monitoring and analysis systems possible", in *Proc. CNRS-NSF Workshop on Environments and Tools for Parallel Scientific Computing*, (Saint Hilaire du Touvet, France, 1993). Elsevier Science Publisher, Advances in Computing Vol. 6, 1993, pp. 103-124.

[26] A. Newell, "The knowledge level", AI Magazine 2 (1981), pp. 1-20.

[27] Office of Science and Technology Policy (OSTP), "Grand challenges 1993: high performance computing and communications", National Science Foundation, 1992.

[28] C. M. Pancake and C. Cook, "What users need in parallel tools support: survey results and analysis", *1994 Scalable High Performance Computing Conference*, May 1994, pp. 40-47.

[29] 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.

[30] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz and L. F. Tavera, "Scalable performance analysis: the Pablo performance analysis environment", in A. Skjellum (ed.), *Scalable Parallel Libraries Conference* Los Alamitos: IEEE Computer Society Press, 1993.

[31] R. Snelick, J. Ja' Ja', R. Kacker, and G. Lyon, "Using synthetic-perturbation techniques for tuning shared-memory programs", Technical Report NISTIR 5139, U. S. Department of Commerce, Technology Administration, National Bureau of Standards and Technology, Gaithersburg, MD 20899.

[32] P. H. Worley and J. B. Drake, "Parallelizing the spectral transform method", *Concurrency: Practice and Experience* 4(4), June 1992, pp. 269-291.

[33] J. C. Yan, "Performance tuning with AIMS - an automated instrumentation and monitoring system for multicomputers", *Proceedings of the 27th Hawaii International Conference on System Sciences* (Jan 1994), Vol. II, pp. 625-633.