Future Generation Computer Systems 838 (2001) 1–12

FGCS
FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.nl/locate/future

A theory and architecture for automating performance diagnosis[☆]

Allen D. Malony*, B. Robert Helm

Department of Computer and Information Science, University of Oregon, 307 Deschutes Hall, Eugene, OR 97403-1202, USA

Abstract

This prospectus describes research to simplify programing 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 programers do performance diagnosis. As a result, it is difficult for researchers to compare existing work or fit their work to programers. It is difficult for programers to locate products of existing research that meet their needs.
2. There is no automated, adaptable software to help programers do performance diagnosis. Existing software is either automated but limited to very specific circumstances, or in general, not 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. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Performance; Diagnosis; Parallel

1. Introduction

Parallel computers and software are being intensively studied to solve “grand challenge” problems [24]. In scientific parallel programs, which have severe performance requirements, “performance bugs” — programing mistakes that make a program slow or inefficient — attract a good deal of attention. Consequently, the job of the parallel programer 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 we call *performance diagnosis*. Expert parallel programers 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 [9,12]. 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 [20,25]. Two obstacles, in particular motivate our current work:

☆ Supported by the US National Science Foundation and the Defense Advanced Research Projects Agency.

* Corresponding author. Tel.: +1-541-346-4407; fax: +1-541-346-5373.
E-mail address: malony@cs.uoregon.edu (A.D. Malony).

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 programers solve their performance diagnosis problems in particular contexts. There is no standard theory for understanding diagnosis system features and fitting them to the programer'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 programer are hard to change, hard to extend and hard to combine with other systems [1]. As a result, programers must do considerable work (re)programing systems or converting data between systems, if existing automated systems do not fit their requirements. In fact, programers generally ignore systems that do not fit their needs exactly. Instead, they collect and analyze data manually [25], or construct custom systems for their own projects [8].

Both the obstacles above, we believe, could be mitigated by a formal theory of performance diagnosis processes. Such a theory should be able to describe how programers 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.

We are now developing and evaluating a formal theory of parallel performance diagnosis by applying models of diagnosis developed in the field of artificial intelligence. We are incorporating that theory into the design of a novel diagnosis system that is both automated and adaptable. This paper presents initial results and planned research towards these ends.

1.1. Problem definition

To clearly define the research problem, we first present a hypothetical example of scientific programing on a parallel computer. The programer 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 programer turns to a parallel computer. To translate the scientific theory into a parallel program, the programer must make numerous decisions. In the neural net example, for instance, the programer must decide:

- How should the state of the neural net be represented computationally?
- What kinds of parallelism should be used?
- How should simulation work be scheduled among processors?

Performance diagnosis guides the programer to bad decisions made during programing. By finding and explaining the chief performance problems of the program, the programer determines which decisions had the worst performance effects, and how those effects might be repaired. As an example of performance diagnosis, suppose that the programer is studying the performance of the neural net program. The programer runs the program and finds that processors are idle much of the time. The programer 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. However, this does not tell the programer which decision should be changed. The programer hypothesizes that the current scheduling scheme — static scheduling — is the problem. This hypothesis implicates a particular decision — the scheduling scheme — and thus constitutes a useful diagnosis if it can be confirmed. The programer switches to dynamic scheduling and observes significantly improved load balance confirming the hypothesis.

Given this example, we now (re)define some terms. During performance diagnosis, the programer decided which performance data to collect, which features to judge significant, which hypotheses to pursue and what confirmation to seek. We 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, we have turned to problem-solving models from the field of artificial intelligence. Specifically, we 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 [4,19]. We 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 programer. We have initially validated this claim by showing how Poirot could *rationally reconstruct* or reproduce a set of existing performance diagnosis systems.

We summarize our research approach with the following claim:

> 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 claim. 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. 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 [23]. A knowledge-level theory of performance diagnosis must answer the question "What knowledge does a programer use to choose actions to meet performance diagnosis goals?" The theory here breaks the question down into two parts:

1. What methods do expert programers use?
2. How can we rationalize the programer's choice of methods?

We 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. Fig. 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

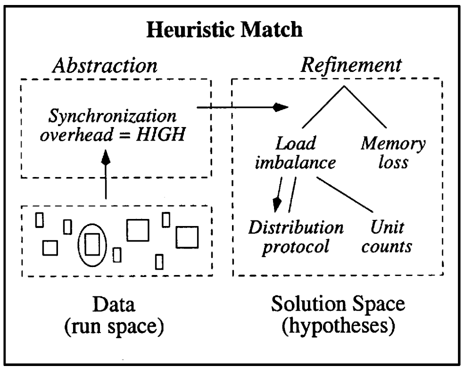

Fig. 1. Heuristic classification.

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 programer *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 scheduling scheme). The programer's *strategy* was to perform abstraction, arrive at an initial hypothesis, refine that hypothesis, and then do diagnosis to confirm that hypothesis.

Table 1
Surveyed systems

System	Paper citation
AIMS	[30]
ATExpert	[16]
ChaosMon	[17]
IPS-2	[21]
MTOOL	[12]
Paradyne (performance consultant)	[15]
Paragraph	[13]
PPP (parallel performance predicates)	[6]
PTOPP	[8]
Quartz	[2]
SPT	[28]

2.2. Thesis summary

Table 1 lists a set of performance diagnosis systems that were re-examined as heuristic classification systems. Table 2 lists a set of questions we developed, based on the heuristic classification model, to characterize performance diagnosis systems.

Fig. 2 summarizes the results of our survey. We draw two main points from these results about performance diagnosis methods in general:

1. Many methods are shared among multiple diagnosis systems. While systems differ in implementation details, they frequently support similar patterns of reasoning to arrive at a diagnosis.
2. Diagnosis systems considered as a whole are fairly diverse. While the set of all methods is

Table 2
Characterizing diagnosis systems

Aspect	Questions
Heuristic match	What is the hypothesis (solution) space? What is the fault taxonomy? What is the component structure? How are features used to retrieve the most significant hypothesis?
Abstraction	How are features abstracted within individual runs? How are features abstracted across runs?
Refinement	How are hypothesis explained by program behavior at the appropriate level of systems description?
Strategy	What direction (data found to hypothesis, hypothesis to data needed) does reasoning flow? How much is diagnosis overlapped with experimentation?

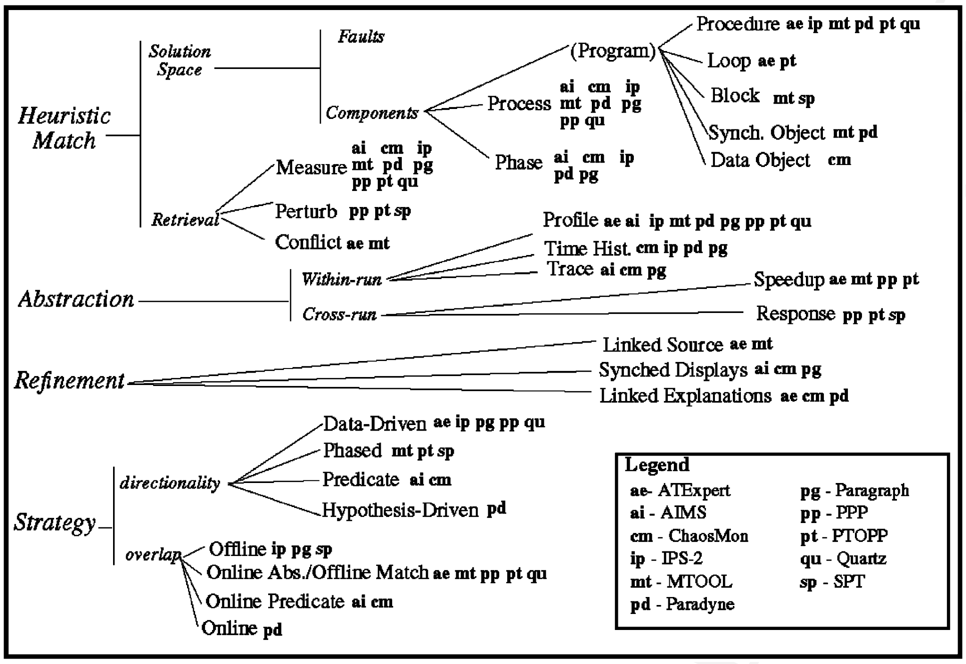

Fig. 2. Performance diagnosis systems as heuristic classification methods.

comparatively small, each system surveyed used a different combination of methods. Systems differ particularly in their methods of abstraction and in their strategies. Each represents a different set of trade-offs among precision, accuracy, machine resource cost, and programer set-up and analysis time.

For heuristic classification, an issue of particular importance is the solution space: how does one pre-enumerate the set of possible performance diagnoses (hypotheses)? Clearly, this is impossible in general, since every parallel application has its own set of possible performance problems. However, the systems surveyed did generally list a set of standard problems to look for, and the measurement, analysis, and displays of each system were integrated to check those hypotheses. Fig. 3 summarizes such "fault taxonomies" for the systems we surveyed. In a few cases, systems look for faults associated with the application currently being diagnosed (ChaosMon) or on the class of parallel algorithm being used (Quartz). However, most systems list performance problems tied to particular programing language or programing model constructs. As a result, the system can in theory diagnose any application, albeit at a fairly low semantic level.

In case studies, the programer generally fills in the missing details, refining an initial, low-level hypothesis into a high-level explanation in terms of application behavior. So, for example, MTOOL points a programer to a memory problem in a block of code, but the programer then "eyeballs" that block to explain why it leads to poor memory performance. Some systems provide particular assistance to this refinement process (Fig. 3). The most obvious form of support, in MTOOL and elsewhere, is "source code clickback", which links a display of a bottleneck to the associated source code. In trace-based systems, such as Paragraph, programers in case studies use the global synchronization of displays as a similar form of "clickback". By examining traces of communication patterns over a time interval, together with performance displays to that interval, the programer can relate program behavior to its performance consequences. Finally, a few systems go further, linking canned refinements to particular bottlenecks. For ex-

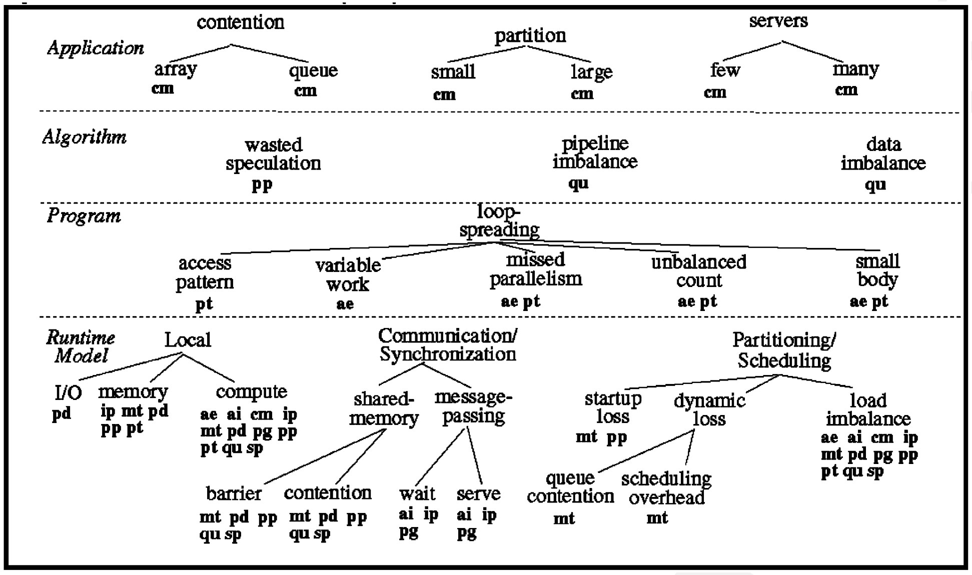

Fig. 3. Fault taxonomies of surveyed systems.

ample, ATExpert provides the programer with text “observations” that explain observed bottlenecks in terms of likely program behavior.

The forgoing discussion suggests one reason why performance diagnosis systems are not widely used, they are not adaptable to a wide variety of contexts. To help arrive at an initial diagnosis, performance diagnosis systems define a limited fault taxonomy, a finite set of performance problems to look for. To date, systems have derived this set from the workings of the programing language and runtime system they support. It follows that the diagnosis systems are limited to a particular class of target machines and environments. Often this class is fairly restricted; MTOOL, for example, requires a machine architectures with fixed latency instructions. Of course, the implementation of any diagnosis system will also frequently limit its applicability. However, we argue that existing systems have conceptual limitations that prevent them from acquiring a large base of potential users.

2.3. Remaining work

We have developed an initial theory of performance diagnosis, as a process of heuristic classification. The theory provides novel framework for characterizing and comparing performance diagnosis systems, and systematically organizes the knowledge that performance diagnosis systems use. We have used the theory to survey existing systems and identify some important distinctions and limitations in their designs. However, the theory is limited. It is based on the case studies reported by performance diagnosis researchers, rather than directly by scientific programers. These case studies may not accurately represent scientific programing problems. We are currently seeking additional case studies from developers of scientific applications [18]. Also, the theory is insufficiently formal to make testable predictions about performance diagnosis processes. To formalize and evaluate the theory, we are building it into Poirot, a novel performance diagnosis system. In the next section, we discuss the architecture of Poirot and our initial results.

3. Poirot architecture

In this section, drawn partially from [14], we 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

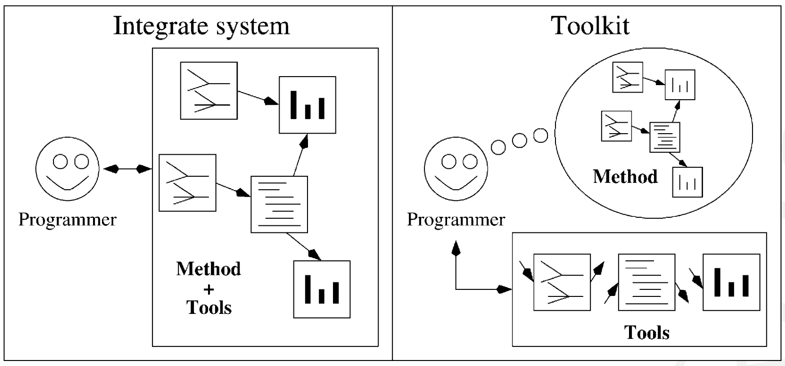

Fig. 4. Integrated systems and toolkits.

adaptability. Existing performance diagnosis systems tend to one of two types (Fig. 4). *Integrated systems* (the type surveyed in the previous section) 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 programers; integrated diagnosis systems are therefore not adaptable. In reaction, some researchers have developed *toolkits*, which supply a general set of tools, and a programing system for combining tools [22,27]. However, toolkits sacrifice automation; the programer must decide on a performance diagnosis method and write additional programs to carry it out.

Poirot offers a third alternative (Fig. 5). The programer using Poirot neither accepts a canned method,

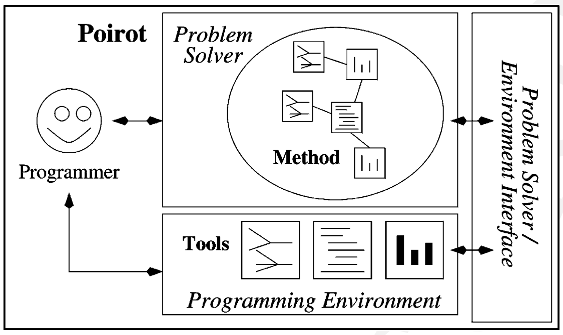

Fig. 5. Overview of Poirot.

as in integrated systems, nor builds a custom method from scratch, as in toolkits. Instead, the programer defines policies, which are interpreted by Poirot's *problem solver* to choose a performance diagnosis method. The programer also helps Poirot to 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.

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 programer'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 programer 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 programer 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

We briefly illustrate the operation of Poirot as an example. The programer in the example is looking for performance problems in the neural net simulation program, which is called `nnet`. The programer 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 (Fig. 6). 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

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.

Fig. 6. Control knowledge for neural net 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)
```

Fig. 7. Goal–method–subgoal trace of example.

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.

Fig. 7 shows a trace of the goals and methods processed during the initial part of a diagnosis session. In Fig. 7, 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 programer 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 number of processors, or simply measure its run time for comparison to the programer'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. However, the experiment does not run. 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.

The preceding scenario illustrates two features of Poirot. First, Poirot can potentially make the diagnosis process highly automated. Even if the programer carried out all the steps corresponding to transformations, Poirot still helps to 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 programer 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. We 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 envi-

Table 3
Reconstruction of three diagnosis systems

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

ronment interface, this suggests that it may be made practical. One could develop a single, core version of Poirot, that a programer could incrementally modify for a particular set of requirements.

Table 3 summarizes the changes a programer 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.

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 programing 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. 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 [12], a programer 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 [29], the programer identifies unexpected delays in a sequence of program events. The key word is "unexpected" — the programer 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, and to validate the theory by direct observational studies of working programers. With such a theory in hand, researchers may be able to close the gap between current performance diagnosis systems and their potential users.

Uncited references

[10,26].

References

[1] M. Abrams, A. Batongbacal, R. Ribler, D. Vazirani, Chitra 94: 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, E. Lazowska, Quartz: a tool for tuning parallel program performance, in: Proceedings of the 1990 ACM SIGMETRICS, May 1990, pp. 115–125.

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

[4] B. Chandrasekharan, T. Johnson, Generic tasks and task structures: history, critique, and new directions, in: J.-M. David, J.-P. Krivine, R. Simmons (Eds.), Second Generation Expert Systems, Springer, Berlin, 1992, pp. 232–272.

[5] W. Clancey, Heuristic classification, Artif. Intell. 27 (1985) 289–350.

[6] M. Crovella, T. LeBlanc, Performance debugging using performance predicates, in: 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, Artif. Intell. 32 (1987) 97–130.
[8] R. Eigenmann, P. McClaughry, Practical tools for optimizing parallel programs, Technical Report 12-76, Center for Supercomputing Research and 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 and Development, Urbana-Champaign, IL, 1992.
[10] M. Feather, S. Fickas, B. Helm, Composite system design: the good news and the bad news, in: Proceedings of the Fourth Annual KBSE Conference, Syracuse, 1991.
[11] S. Fickas, Automating the transformational development of software, IEEE Trans. Softw. Eng. 11 (11) (1985).
[12] A.J. Goldberg, J.L. Hennessy, Mtool: an integrated system for performance debugging shared memory multiprocessor applications, IEEE Trans. Parall. Distr. Syst. 4 (1) (1993) 28–40.
[13] M. Heath, J. Etheridge, Visualizing the performance of parallel programs, IEEE Softw. 8 (5) (1991) 29–39.
[14] B. Helm, A. Malony, S. Fickas, Capturing and automating performance diagnosis: the Poirot approach, in: Proceedings of the Ninth International Parallel Processing Symposium, Santa Barbara, CA, April 1995.
[15] J. Hollingsworth, Finding bottlenecks in large-scale parallel programs, Ph.D. Thesis, Department of Computer Sciences, University of Wisconsin–Madison, 1994.
[16] J. Kohn, W. Williams, ATExpert, J. Parall. Distr. Comput. 18 (1993) 205–222.
[17] C. Kilpatrick, K. Schwan, ChaosMon — application-specific monitoring and display of performance information for parallel and distributed systems, in: Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, May 1991, pp. 48–59.
[18] A. Malony, B. Helm, Call for collaboration: performance diagnosis processes, Technical Report 95-01, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403.
[19] J. McDermott, Preliminary steps toward a taxonomy of problem-solving methods, in: S. Marcus (Ed.), Automating Knowledge Acquisition for Expert Systems, Kluwer Academic Publishers, Boston, MA, 1985.
[20] P. Messina, T. Sterling (Eds.), System Software and Tools for High Performance Computing Environments, SIAM, Philadelphia, PA, 1993.
[21] B. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, T. Torzewski, IPS-2: the second generation of a parallel program measurement system, IEEE Trans. Parall. Distr. Syst. 1 (2) (1990) 206–216.
[22] B. Mohr, Standardization of event traces considered harmful or is an implementation of object-independent event trace monitoring and analysis systems possible, Adv. Comput. 6 (1993) 103–124.
[23] A. Newell, The knowledge level, AI Magazine 2 (1981) 1–20.
[24] Grand challenges 1993: high performance computing and communications, Office of Science and Technology Policy (OSTP), National Science Foundation, 1992.
[25] C. Pancake, C. Cook, What users need in parallel tools support: survey results and analysis, in: Proceedings of the Scalable High Performance Computing Conference, May 1994, pp. 40–47.
[26] D. Reed, Performance instrumentation techniques for parallel systems, in: L. Donatiello, R. Nelson (Eds.), Models and Techniques for Performance Evaluation of Computer and Communication Systems, Lecture Notes in Computer Science, Springer, Berlin, 1993.
[27] D. Reed, R. Aydt, R. Noe, P. Roth, K. Shields, B. Schwartz, L. Tavera, Scalable performance analysis: the Pablo performance analysis environment, in: A. Skjellum (Ed.), Scalable Parallel Libraries Conference, IEEE Computer Society Press, 1993.
[28] R. Snelick, J. Ja' Ja', R. Kacker, G. Lyon, Using synthetic-perturbation techniques for tuning shared-memory programs, Technical Report NISTIR 5139, US Department of Commerce, Technology Administration, National Bureau of Standards and Technology, Gaithersburg, MD 20899.
[29] P. Worley, J. Drake, Parallelizing the spectral transform method, Concurr.: Prac. Exp. 4 (4) (1992) 269–291.
[30] J. Yan, Performance tuning with AIMS — an automated instrumentation and monitoring system for multicomputers, in: Proceedings of the 27th Hawaii International Conference on System Sciences, Vol. II, January 1994, pp. 625–633.