

# Performance Analysis Strategies

Adam Leko

UPC Group, University of Florida

## Abstract

Analyzing the performance of programs is fundamentally the most important function a performance tool must have. There are a wide variety of different analysis techniques, ranging from relying entirely upon the user to perform analysis to semi-automated techniques that are able to automatically classify performance problems in code. This paper will classify and summarize existing analysis techniques, and give recommendations on which analysis techniques are most relevant to a performance tool targeting UPC and SHMEM.

## 1 Introduction

Performance analysis is a key issue that should be carefully considered when designing a new performance tool. How well a tool is able to analyze performance data can play a large part in the acceptance of a tool. Good performance tools should not only be able to display data in an intuitive manner, but should also help a user pinpoint interesting data so that any existing bottlenecks or performance problems can be easily identified.

An idealized work flow for tuning performance using a performance tool is shown in Figure 1. In the idealized work flow, a user first takes their unoptimized code, adds instrumentation code to it (preferably with little or no effort), runs it, and then feeds the data collected during runtime into a performance tool. The performance tool should help the user identify and correct performance problems in their code. This process is repeated until the user obtains the necessary amount of performance from their code. The instrumentation process should add negligible overhead to the program's execution, and should collect all data in such a way that all possible performance problems can be detected. In this ideal case, performance analysis

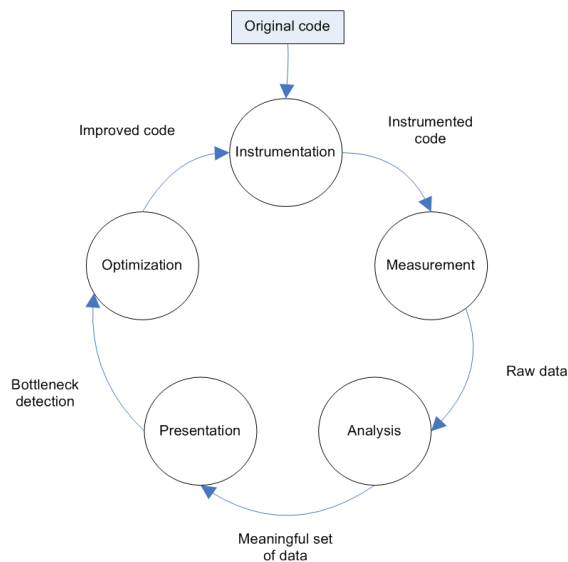


Figure 1: Idealized work flow

should be as fully automated as possible, and can be performed completely and accurately after running the program (post-mortem) since all relevant data has been collected during runtime.

In reality, no tool in existence uses this idealized work flow as-is; instead, existing tools tend to support parts of it, trading functionality for overhead and usability where appropriate. Current systems are so complex that collecting all information necessary to detect all possible performance problems during runtime is infeasible without perturbing an application’s performance, negating the usefulness of the information that was collected. In some cases, architectural properties such as out-of-order execution make it nearly impossible to know exactly what is going on at any given moment in hardware. Even when relatively complete data *can* be collected in a system without perturbing an application’s runtime characteristics, the large size and varying types of data create major information fusion problems that complicate data analysis.

Thus, real-world performance tools often have to make decisions about what kind of data they will collect, and may be forced to perform analysis and/or filtering when imperfect and incomplete information is available. Effectively, this means that filtering and analysis can (and sometimes must)

be performed at different phases of performance tuning, instead of at fixed times as in the idealized workflow of Figure 1.

The rest of this paper gives an overview of existing performance analysis strategies, which are categorized on how they tackle the problems mentioned above. Existing strategies will be categorized as follows:

- *Post-mortem vs. On-line vs. Pre-execution* – Pre-execution strategies perform all analysis before a program is executed, while on-line performance analysis strategies perform all analysis during runtime. Post-mortem strategies must have complete information collected during runtime before they can be applied.
- *Manual vs. Semi-automated* – Some analysis and bottleneck detection strategies can be automated in some fashion or another, while others rely upon the user to analyze and interpret data by themselves.

Note that the categories are not necessarily orthogonal; some strategies may use a combination of the above approaches. In these cases, we will categorize each strategy by its main approach used and note any deviations that occur from this main approach.

Scalability is another important issue that needs to be considered when discussing performance strategies. Strategies that work well for small systems but fall apart for systems larger than a few dozen processors are not very useful, as it is often the case that many performance problems do not appear until the system size reaches a certain point. Additionally, manual analysis techniques that rely on the user to perform most of the work can become very ineffective if the user has to wade through too much information as the system size grows. In our discussions of each strategy, we will also consider scalability, noting any possible problems one may encounter when applying the strategy to large systems.

The rest of this paper will be organized as follows: Section 2 will describe existing post-mortem analysis strategies. Section 3 will describe existing on-line analysis strategies, and Section 4 will describe strategies that take place before a program’s execution. Finally, Section 5 will give recommendations on which strategy or combination of strategies is most appropriate to use in a performance tool geared towards UPC and SHMEM.

## 2 Post-mortem analysis strategies

Post-mortem analysis strategies are applied after a program’s execution and use information collected during runtime, usually via tracing methods. These strategies have become the de-facto standard for most existing performance tools that support message-passing programming models. As mentioned before, in an ideal world, one would always use a post-mortem analysis strategy as long as all relevant data can be collected at runtime with negligible overhead. Since most message-passing applications are written using coarse-grained parallelism, most of the application’s behavior can be captured (or extrapolated from) from event traces containing all message-passing behavior of an application. Considering the popularity of existing message-passing systems (MPI) and their performance tools, it is not surprising most existing strategies fall under this category.

### 2.1 Manual techniques

Strategies in the manual analysis category rely on the user to perform most of the “heavy lifting” when searching for performance bottlenecks. The tools using this strategy depend entirely on presenting the user with appropriate visualizations and methods for manipulating those visualizations (zooming and searching) to find performance problems. Visualization of performance data can be a powerful technique, but without sophisticated filtering techniques, the amount of visualized data can easily become overwhelming.

To give a concrete example on how data in trace files can become overwhelming, the trace file obtained from running the NAS LU benchmark (class B workload) as visualized by Jumpshot-4 is shown in Figure 2 with no filters enabled. The trace file in question is well over 325MB for only roughly three minutes of execution time. As can be seen from the figure, it is very hard to discern what is going on due to the large amount of communication that is shown, even though a 32-processor system is considered a small system by today’s standards. To deal with this problem, tools that use trace files with the manual analysis strategy often provide searching, filtering, and zooming capabilities to restrict the amount of data being displayed. For example, many MPI trace file visualization tools allow the user to restrict the messages being displayed by specifying a specific tag or source and destination pairs. However, most tools that employ manual analysis using trace files do not have the ability to relate information back to the source code level, since

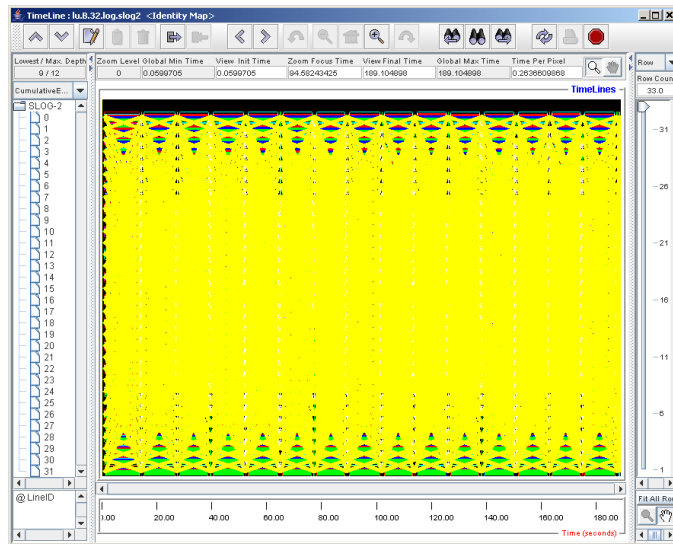


Figure 2: Jumpshot-4 visualization of NAS LU benchmark, 32 processors

it is difficult to show this information in a timeline-based view. It also also the responsibility of the user to choose filters that display useful information.

Another technique to cope with the problem of large amounts of information is the use of profiling instead of tracing. Profiling data can scale to large numbers of processors since it only provides summaries of statistical information. This has the advantage of having a very small measurement overhead and that summary information can easily be displayed inline with source code, but suffers from the problem that it is often hard to track down exactly what is going on and why performance is bad given only basic summary information.

Many users also use ad-hoc tracing and profiling methods based on manually placing `printf` at key points in their source code. This can be quite effective and in some cases can give similar information as trace-based methods, but also can impose a large overhead on the user since tracing or profiling code has to be written from scratch each time. A few existing tools are aimed at providing the simplicity of using `printf` while maximizing the amount of information the user can get out of them. For example, the NetLogger tool [1] provides a distributed logging facility that is in effect a distributed `printf`, and the SBI [2] library contains a very simple method to detect when pro-

processors enter a barrier and how long each barrier takes in MPI code.

Some of the first tools to support methods for detecting bottlenecks in parallel environments focused on shared-memory machines. In shared-memory machines, memory contention becomes an obvious bottleneck in parallel code, but this contention can be hard to detect (if not impossible) using simple methods such as inserting `printf` statements. Examining techniques used for detection of bottlenecks in shared-memory architectures is especially valuable in our project, since UPC and SHMEM are both designed for these types of machines.

An early performance tool from Stanford named Memspy uses simulation (through a simulator dubbed “Tango”) to identify problematic memory accesses and relate them back to the source code level [3]. Since collecting a complete trace of all memory access would lead to an incredible amount of information, Memspy uses statistical profiling in conjunction with symbol tables to relate detected bottlenecks back to the source code level. After running code through the tool, a simple textual description of variable names that caused large amounts of page faults is displayed to the user. Because of the use of simulation, Memspy can detect memory contention with a high degree of accuracy. However, since each memory access has to be caught and simulated, application runtimes are reduced by two orders of magnitude or more. A more recent tool that employs this strategy is the open-source Valgrind tool [4], which is able to simulate the memory hierarchy of several commercial processors such as the Intel Pentium 4.

Another early tool that performs shared memory profiling is MTOOL, which also was created by researchers at Stanford [5]. MTOOL uses a different strategy than what is employed by Memspy; instead of simulating memory access directly, code is instrumented and the time taken by basic code blocks (loops and subroutine calls) is recorded. These actual times are compared against ideal times, which are computed using knowledge of instruction latencies under the assumption that no pipeline hazards are encountered. Any extra time that is spent over the optimal time is attributed to memory contention. While this strategy was feasible in early single-issue RISC machines (and has been successfully applied with minimal overhead to long-running programs in the DCPI profiling system [6]), it would not be possible to statically predict instruction latencies in modern multiple-issue deeply-pipelined processors which have multiple functional units. Therefore, this technique cannot be used on modern machines, although the basic idea may be used at a higher level to identify bottlenecks in simple bus-based com-

munication networks (but not complex switched interconnection networks).

Other existing tools implementing the post-mortem manual analysis strategy include trace file viewers such as Jumpshot-4 [7], Paraver [8], and VampireTrace [9]. Profile-based performance tools such as mpiP [10] and SvPablo [11] also fall under this category. In addition, an overview of analytical scalability analyses which only need the overall execution time of an application (Amdahl’s law, isoefficiency, and Gustafson’s law, among others) is presented in a survey written by Hu and Gorton [12].

## 2.2 Semi-automated techniques

Machines tend to be much better at sifting through large amounts of data than humans. With this in mind, several researchers have implemented prototype systems that try to automate the process of searching for performance problems as much as possible. There are several existing prototype tools that can perform at least a form of automated analysis of runtime performance data, and we give an overview of the more popular tools and methods in this section. Most strategies here use techniques from artificial intelligence such as knowledge-based expert systems and automatic classification or some form of modeling to identify performance problems.

IPS-2 [13] is an early tool that supports limited automatic analysis. When using the IPS-2 system, a trace file is collected after an instrumented version of the program is run. Two types of post-mortem analyses are supported by the tool: phase analysis and critical path analysis. Critical path analysis identifies the sequence of events that is responsible for the overall runtime length of the program. To perform the critical path analysis, a graph of the program’s activities is first constructed based on that program’s event trace, which represents the dependencies between different parts of the program’s execution. The graph is then analyzed using a modified shortest-path algorithm (which actually determines the longest path) to detect the critical path of the program’s execution. Phase analysis tries to detect different phases of a program’s behavior so that they may be analyzed separately. Once phases are identified, the critical path analysis can be used on the detected phases. The general motivation behind using critical path analysis is that it not only gives the user an idea of what parts of the program that take a long time, but what portions of the program will have the most impact in overall execution time. This idea has also been used in the trace-driven simulator, Dimemas [14].

An analysis approach that takes a much different approach to determining the methods that have the most overall impact on execution time is employed by the S-Check tool [15]. Instead of searching for the critical path of a program directly based on its trace file, artificial delays are inserted in methods of interest of a program (“synthetic perturbation”). Several permutations of delays between each method are run and the runtimes for each experiment are automatically recorded by S-Check’s user interface. Once the runtimes for the permutations are recorded, they are analyzed using a linear response model in conjunction with statistical methods. After the statistical analysis, each method is assigned a certain score which rates the relative influence that particular method has on overall execution time as compared to the other methods. The experimentation phase is fully automated, but no processing is done until after each variation of artificial delays is completely run. The tool also automatically computes the minimum number of experiments that must be run in order to capture the interplay between execution times of a particular set of methods.

Vetter’s automatic event classification borrows techniques from machine learning in order to automatically process event traces [16]. In Vetter’s technique, events in trace files are labeled under different categories such as late sender and late receiver. The novel idea in his approach is that events are analyzed using decision tree classification. When working on a new architecture, microbenchmarks are used to “train” the tree with examples of good and bad behavior. Vetter uses decision trees instead of more prevalent (and slightly more accurate) techniques such as neural networks. The advantage of using decision trees is that not only are events classified, but it is also possible to capture the reason why an event was classified a certain way. Vetter claims that this approach leads to better portability and robustness as compared to simpler fixed-cutoff schemes, since the technique is able to adapt to a changing environment automatically by retraining the decision trees. In addition, sometimes it is much easier to give examples of “good” or “bad” behavior without knowing the exact reason why a communication pattern performs well or not. In this sense, this analysis technique allows users to store knowledge-based information at a much higher level than other techniques, although it has the downside that extensive and computationally expensive training is necessary whenever a new type of bottleneck is discovered. Nevertheless, it is somewhat surprising that more tools have not adopted an automatic classification scheme based on machine learning techniques.

Scal-tool adopts a model-based approach for determining the scalability of

an application [17]. Similar to lost cycles analysis (see Section 3.2), program overheads are recorded during execution, although in Scal-tool these overheads are recorded directly using hardware counters. After the program has been run, speedup curves are produced based on the actual recorded runtimes of the unmodified program. Then, the program overheads from the hardware counters (such as L2 data cache misses) are fed into an empirical model that predicts scalability graphs based on the ideal case that these bottlenecks are completely eliminated. The authors acknowledge the limited accuracy of their chosen analytical model, but stress that it is not important for the model to give completely accurate results; rather, they claim that it is more important to give the user an idea of what kind of performance they might expect if they completely eliminate a particular bottleneck. This is an interesting technique, and similar to lost cycles analysis, performs the “optimal case analysis” we have been considering for inclusion into our performance tool. Another similar analysis technique is employed by the SCALEA tool [18] (first known as SCALA [19]), . Scalea automatically classifies bottlenecks by identifying the amount of time a program spends on communication in code regions, and relates them to the source code level using a tree-like view. It also supports comparing different executions directly by storing execution data inside an SQL database, and supports automatic instrumentation but requires compiler support (it operates over the abstract syntax tree). The instrumentation library is available as a standalone library, but only supports Java and Fortran programs.

The majority of semi-automated analysis methodologies that exist in literature adopt an knowledge-based approach to the problem of identifying performance problems in existing programs. These types of tools contain a knowledge base of known performance problems which are matched against complete trace files created out of program runs that are indicative of “normal” application behavior. Once bottlenecks are detected, some analysis techniques also include a method for indicating a few strategies on how to fix the performance problem based on information contained in the knowledge base.

Of all the knowledge-based approaches described in the literature, the earliest attempt was the Poirot system [20], developed by researchers at the University of Oregon. The Poirot system never had a fully-functional implementation, but contained many ideas that have been prevalent in most knowledge-based systems since its inception. In the Poirot system, a program is analyzed using regular tracing methods. The trace file is matched against

an existing knowledge base containing bottlenecks, and any bottlenecks that are detected are presented to the user along with a set of solutions for that particular program. The whole process is user-guided, but the goal of the Poirot system is to minimize user interaction by capturing and automating the performance diagnosis process. The authors of Poirot still recognize that the user has to play a significant part in the optimization process, and so tried to build their tool around that assumption. However, Poirot never got much farther than an early research prototype, and has since been dropped by the researchers in favor of the TAU performance analysis tool.

One of the more popular knowledge-based tools is the KAPPA-PI tool [21, 22], which is geared towards MPI and PVM applications. KAPPA-PI uses a fixed database of performance problems along with a database of suggestions for remedying those problems. It can detect a wide range of communication-related performance problems, and relates these problems back to the source code level. The integrated source browser also suggests (in English) ways to correct the performance problems, but this step is not automated. The Merlin tool [23] is very similar to KAPPA-PI, except that it is targeted towards beginners as a way to pass performance-specific knowledge from more experienced users to less experienced ones. In Merlin parlance, the knowledge database is called a “performance map,” which relates performance problems and their corresponding solutions. The FINESSE tool [24, 25] adopts a similar knowledge-based approach, and is geared towards automatically-parallelized codes such as those found in OpenMP and High-Performance Fortran. The FINESSE tool, like Scal-tool, shows speedup curves as its main visualization. FINESSE requires the ability to parse the source code (its abstract syntax tree) and is focuses on detecting problems that inhibit automatic parallelization. Like KAPPA-PI, there is no visible way for a user to add or remove performance problems to the knowledge base. FINESSE supports an IDE-like environment that includes version control management and experiment management, two features that we should strongly consider including in our tool since they greatly reduce the work the user has to perform to track how much they have improved their program. Finally, another very popular analysis tool that uses the knowledge-based strategy is EXPERT [26, 27], part of the KOJAK performance analysis tool suite. EXPERT current supports detection of communication-based problems, and has an extensible architecture to allow users to add performance problems to the knowledge base.

When applying a knowledge-based approach to semi-automatically de-

tecting performance problems, it is often very useful to let the user add their own entries into the knowledge base of performance problems. While many tools use fixed sets of bottlenecks, several support extending the core set by the user. There are three such approaches described in the literature. The first approach was created by the APART group, a consortium of academic and industry members whose goal was to create a foundation for automatic performance analysis tools. The APART group defined ASL, the APART Specification Language, which is nothing more than a standard grammar for specifying performance problems (“properties”) [28]. While the ASL is a powerful domain-specific language, no current systems have implemented a complete ASL parser and matching system, although a stripped-down version that has a syntax close to Java has been implemented and tested [29]. The second approach to representing performance problems in a knowledge base is taken by the EXPERT tool. Instead of creating a domain-specific language to describe performance problems, researchers created EARL [30], a standard API which provides high-level operations over trace file events. EXPERT can be extended to detect new performance problems by writing Python scripts or C++ objects (which use the EARL API) that implement a specific event-based callback interface. This callback interface is similar to event-based XML parsers and provides a powerful way to add detection capabilities for arbitrary performance problems. A third approach to the knowledge-based problem is the Event Description Language (EDL) [31], which is based on extended regular expressions. EDL allows a compact representation of many performance problems, but suffers from the problem that since it is based on regular expressions (finite automata), it cannot handle state-based performance problems.

### 3 On-line analysis strategies

Instead of deferring analysis to the end of a program’s execution, a few tools attempt to perform this analysis at runtime. This has the advantage that large trace files can be avoided, but also has the drawback that the analysis has to be performed quickly and with limited data.

On-line analysis strategies tend to adopt the approach of dynamic instrumentation; that is, only recording subsets of available performance information at any given time and changing the recorded data dynamically during runtime as dictated by whatever performance analysis strategy is in

use. Making the wrong decision about what data to record at runtime can often be *more* costly than standard event tracing, since to get that required data back it is usually necessary to re-run the application code. Therefore, it is not surprising that most tools that attempt analysis of performance data use event tracing in combination with post-mortem analysis. Thus, existing literature on this topic is more limited than the previous section.

### 3.1 Manual techniques

Requiring the user to perform manual interaction during runtime is usually not a good idea, for several reasons. First of all, the user most likely does not know exactly what they are interested in recording (otherwise they would probably not be using a performance tool). Secondly, program phases can change rather rapidly, so it may be extremely difficult for users to keep up with a program’s execution. Because of these reasons, the vast majority of tools that perform runtime analyses rely on the process to be automated by the performance tool.

However, an existing tool named PATOP [32, 33] requires the user to pick out which information they want *while* their program is executing. This has the advantage that no data is ever recorded that the user will not look at, but suffers from the exact problems previously mentioned. According to its authors, PATOP is able to classify 4 categories of bottlenecks, which are:

- Insufficient coding of sequential parts
- Insufficient parallelism
- Delay of execution streams in accesses to remote system components
- Delay of execution streams due to an overloaded communication network

The author of the PATOP system claims that all these bottlenecks result in idle processor cycles, so they can be detected by processor utilization graphs only. While the claim that all bottlenecks result in idle processor cycles is essentially true, it is not very helpful to rely completely on processor utilization, since this alone does not tell you what kind of bottleneck is occurring.

PATOP does support recording other metrics, and even has a feature-laden dialog to help decide which metrics to record on which processors. The dialog allows you to record “attributed measurements,” which is a way

to specify a range of machines you wish to record a metric across given a high-level description like “record the average I/O delay for all processors spending more than 50% of their time in external I/O.” However, since the tool does not parse these descriptions in natural language, the interface for specifying these attributed measurements is very convoluted.

## 3.2 Semi-automated techniques

Performing semi-automated analysis at runtime is much more valuable to the user if it can be performed in an automated fashion. A few existing tools support semi-automated analysis techniques that are performed at runtime. To support these techniques, most tools require a method of turning instrumentation for specific parts of programs on and off. The most popular method of doing this is dynamic binary instrumentation, as provided by libraries such as DynInst [34], although other techniques exist (such as temporary relocating dynamic link tables [35]).

Paradyn is an extremely popular tool which only supports runtime analysis [34]. Paradyn supports a mode similar to PATOP in which a user selects which metrics they wish to view, but also has an automated search routine named  $W^3$  that automatically searches an application for a set of known bottlenecks. The  $W^3$  model searches for

- *Where* a program performs badly,
- *When* a program performs badly, and
- *Why* a program performs badly

To perform the  $W^3$  automated search, Paradyn starts out with a singular hypothesis and refines it based on available runtime information. For example, if a method is currently taking up a large amount of time in synchronization, Paradyn may label that method with a *ExcessiveSyncTime* label, which may be later refined to a specific method in the program’s source code. Paradyn uses DynInst to dynamically instrument executable images while a program is running. Paradyn also uses a client-server architecture where a single Paradyn daemon runs on one host, and all instrumented copies of the application send performance information to the main daemon. The daemon can request parts of the application be instrumented on each host, and it is with this ability that it performs its automated search. Hypothesis are

refined with further information from dynamic instrumentation, or rejected outright. While the search method is fairly effective, because the search is performed at runtime Paradyn only has a handful of bottlenecks that it is able to identify, and Paradyn seems to take a very long time to identify bottlenecks. In addition, after a bottleneck is identified, no effort is made to help the user fix the bottleneck (it is strictly an identification tool). Paradyn’s search method originally used a simple heuristic to order search nodes, but has recently been extended to use the application’s call graph [36] and call stack [37]. These heuristics attempt to search parts of a program’s execution that are likely to have performance bottlenecks first.

Two other existing systems that adopt an approach similar to Paradyn are Peridot [38] and OPAL [39]. OPAL originally used selective tracing in conjunction with an incremental approach to recording runtime information, but was later extended to do automatic analysis at runtime [40]. OPAL uses hypothesis, refinements, and proof rules similar to Paradyn, except that refinements cause new executions of a program to be started with new data to be collected via selective tracing. Peridot takes a much more ambitious approach than Paradyn in supporting large-scale systems. Peridot uses a software agent-based approach with dedicated parts of the computational infrastructure reserved for recording runtime information, and uses ASL to encode performance problems. While Peridot is probably more suited to grid computing, the overhead for recording this information could potentially be extremely high. In addition, Peridot’s website has not been updated in over 3 years, and no prototype implementing the design of Peridot has been constructed. It is likely that Peridot, like Poirot, is a system that looks good on paper, but many sacrifices have to be made for it to work on real-world systems such that its actual usefulness is limited.

Lost cycles analysis [41] is another analysis technique that relies on sampling a program’s state to determine whether it is doing useful work or not. Lost cycles does not require dynamic instrumentation; instead, instrumentation code is inserted (by the user or a tool) that sets flags describing the current state of a program’s execution. Any time the program spends that is not directly related to computation is categorized as “lost cycles.” Lost cycles are broken down into 5 categories: load imbalance, insufficient parallelism, synchronization loss, communication loss, and resource contention. The classification of program cycles is done at runtime based on predicates which directly sampled state information. For example, there exists work imbalance in a program if, at a particular point in time, one processor is

doing useful work while the others sit idle waiting on synchronization. Lost cycles analysis has an advantage that it can also be used to predict program scalability with a decent amount of accuracy, but suffers from the drawbacks associated with sample-based analysis: namely, it can be difficult or impossible to relate information back to the source code level. Also, for detection most types of lost cycles, the sample-based methodology has a rather simple implementation, since all that has to be recorded at runtime is a sample of the program’s current state.

One existing system that is geared towards grid computing is the Autopilot [42] tool and its corresponding visualization environment, Virtue [43]. Autopilot uses the Globus grid computing system [44], and uses the Nexus global address space toolkit [45]. Autopilot is composed of distributed *sensors* and *actuators* which register with a global name service. The purpose of the sensors is to record performance information. Actuators query the name servers to find out which sensors are available, and can make steering decisions such as selecting an appropriate algorithm variation depending on the runtime environment’s current properties. One other interesting aspect of Autopilot is that decisions can be made using fuzzy logic, which the authors claim allows a user to “elegantly balance potentially conflicting goals.” For shared-memory performance analysis in which the environment stays relatively stable, online steering of an application most likely has too much overhead to be useful. However, if we wish our tool to support wide-area distributed systems, it may be worthwhile to examine online application steering in the future.

A novel approach to runtime analysis which aims to greatly reduce event traces uses statistical methods to perform “clustering” in event traces [46]. The authors of this method noted the popularity of the Single-Program, Multiple-Data (SPMD) and data-parallel coding styles, which leads to equivalence classes of processors. With most SPMD and data-parallel programs, even though a large number of processors are executing at a given time, there are generally only a handful (or less) of distinct operations going on at once. Therefore, instead of tracing all processors, a representative from each equivalence class of processors is chosen, greatly reducing the amount of tracing data obtained. To select a representative, sliding windows of all metrics are recorded using a relatively short window size, which acts as a smoothing filter over the data. All metrics are then normalized between a fixed range. Conceptually, these metrics are projected into an  $n$ -dimensional space, and groups of processors that are close to each other form clusters.

In practice, clusters are partitioned using square error clustering. Clusters are recomputed at fixed time intervals, which must be short enough to capture changing program behavior but long enough so that recomputing cluster members does not introduce a large overhead. Finally, to reduce the “jitter” that may occur when different members of a cluster are selected as that cluster’s representative, the member of the cluster that has been there the longest is always chosen as the representative. Statistical clustering can be used to obtain 1-2 orders of magnitude reduction on trace file sizes (depending on how many processor equivalence classes there are), but the resulting trace files are only an approximation of what is occurring at runtime. In addition, the clustering process does introduce computational overhead, especially if high-quality clustering is to be performed, and even if perfect clustering is achieved, a lot of information such as dependencies between passing messages around cluster members is lost.

## 4 Pre-execution analysis strategies

For many applications, attempting to analyze the performance of a program *before* it is executed is too error-prone to be useful. However, several strategies such as modeling and simulation are well-accepted techniques of doing performance analysis, although they usually have interesting tradeoffs.

### 4.1 Manual techniques

As previously mentioned, the most accepted way of doing performance analysis before a program’s execution is to model a program’s performance using analytical or simulative techniques. An overview of modeling techniques is presented elsewhere (TODO: ref to modeling) and will not be described here.

Simulation is an entire subject in and of itself; thus, only a short overview will be given here. There are many techniques to perform simulation, including discrete-event simulation, Monte-Carlo simulation, and continuous-time simulation. Discrete-event simulation is a very generic methodology that is able to model systems that change states at discrete moments in time, and thus is often used to model discrete and clocked systems such as processors and communication networks. A key problem with simulation is trying to balance the expensive cost of performance simulation with the fidelity obtained from running the simulations. The FASE (Fast and Accurate Simulation

Environment) project of the Modeling & Simulation group at the University of Florida's HCS lab is an attempt to tackle this problem by providing a simulation environment which combines pre-simulation to obtain traces with simulation to predict performance. The trace-driven simulation approach is also adopted by other tools such as Dimemas [14] and PACE [47], although these systems use much coarser models for simulating communication than what is employed by FASE.

While modeling and simulation can be very powerful methods for analyzing bottlenecks in a program's execution, it seems they are best suited to complement other performance analysis techniques. Even though great lengths have been made in the FASE project of speeding up simulation time while retaining fidelity, simulation times still can be several orders of magnitude slower than running the program on real hardware. Analytical modeling offers a compromise on accuracy and speed, but it can be difficult to keep evaluation times low while still extrapolating useful information. Needless to say, this is a research area where a lot of work is still going on, and it would be very beneficial for our project to further research different analytical modeling and simulation techniques in later phases of the project.

## 4.2 Semi-automated techniques

Because of the relative difficulty in analyzing a program's performance automatically before its execution, there are few semi-automated techniques explored in existing literature. Most techniques that attempt to accomplish a form of automated analysis concentrate only on optimization (see TODO: ref to optimization report for an overview).

However, there is one knowledge-based analysis tool proposed by Li and Zhang [48] which is meant to be applied automatically to source code before it is executed. PPA (Parallel Program Analyzer) first parses program source code and creates an abstract syntax tree. This tree is matched against an existing knowledge base, which uses a domain-specific language for encoding common-sense knowledge about good performance methodologies. Using this knowledge base, suggestions are given to the programmer on how to improve the performance of their code. PPA was initially proposed in 1996, but no additional work has been done since the initial proposal. It is also not clear whether a prototype has been created which implements the ideas presented in the original paper. Given the difficulty in processing source code and representing high-level programming techniques and the lack of case studies

used in available literature on PPA, it seems this technique has not gained widespread acceptance because of implementation barriers.

## 5 Conclusions and recommendations

This paper has presented an overview of existing analysis techniques, which range from simple techniques which rely entirely on the user to complicated ones which borrow concepts from artificial intelligence. It is interesting to note that of all the methodologies presented here, the vast majority of tools in existence use the simplest of all strategies, which is to provide user-driven filtering capabilities in a visualization environment. Three notable exceptions to this rule are Dimemas, EXPERT (KOJAK), and Paradyn. One common feature between these three exceptions is that they have *working* implementations of complex ideas, even if the current implementations are just scaled-down versions of the original idea. There seems to be a strong correlation between the livelihood of projects that have strong implementations versus tools that exist just as proof-of-concept vehicles.

Of all the strategies mentioned, the strategy that seems most amenable to both user utility and implementation ease is the trace-driven post-mortem analysis. Most tools seems to recognize the utter importance of involving the user in the analysis of a program. While several strategies exist that attempt to (eventually) exclude the user from the tedium of analyzing and optimizing their code, the problem space of optimization is not fully understood. Therefore, any analysis method beyond simple scalability analyses *should* expect user interaction, and thus a tool implementing a strategy should strive to provide a usable interface to ease the interaction between the user and the tool.

For our UPC and SHMEM performance tool, we should write a tool or pick an existing one that is able to perform post-mortem trace-based and profile-based analysis. While runtime analysis strategies (and other exotic strategies such as trace file clustering) are interesting and can potentially speed up analysis, in the end an accurate analysis is much more valuable to the user than an incomplete analysis that finishes quickly. However, our tool should support an incremental selective tracing approach for large-scale applications that would be very expensive to trace on a full scale.

Lost cycles analysis seems like a nice initial approach to take, since it would be relatively easy to implement and can be used to do performance

prediction and scalability analyses. Once portions of the program have been identified that have poor performance, these regions can be analyzed further using full tracing or more detailed profiling. In addition, lost cycles provides a very easy way to compare program performance against an optimal case (as in Scal-tool), since speedup curves could be easily produced for an optimal version of the program that is the actual program's execution time minus any lost cycles.

One could draw a strong parallel between a two- or three-level memory hierarchy and the programming model UPC and SHMEM present to the user. Since it is important to present the user with information that is relevant to the programming model in use, and the most popular method for presenting information related to the memory hierarchy is the use of profiling, a logical choice would be to present all implicit communication by showing profile information ("miss rates" as non-local accesses, communication frequency), and use event tracing to capture any explicit communication (SHMEM calls and UPC library calls). The tool should definitely contain at least a rudimentary experiment management facility so that the user does not have to work hard to keep a record of performance information so that traditional scalability analyses (speedup curves, etc) can be easily performed.

Once a solid implementation for the basic instrumentation and visualization of UPC and SHMEM programs has been created, it could be extended to use semi-automatic knowledge-based analysis capabilities by using the EARL trace file analysis library. EARL seems a logical choice since it already has an existing implementation (unlike ASL) and can be easily extended by writing Python or C++ code (unlike KAPPA-PI and other knowledge-based tools). In addition, the knowledge base can be written entirely in C++, which should help speed the analysis phase.

One advantage to this approach is that supporting other programming models such as MPI, OpenMP, or a combination of them can be accomplished without a complete rewrite of the tool. It would also provide an ideal research vehicle for exploring other research topics, such as fully automatic performance analysis, using fuzzy logic in the performance knowledge base, and other as yet undiscovered techniques.

## References

- [1] B. Tierney, W. E. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter, “The netlogger methodology for high performance distributed systems performance analysis,” in *HPDC*, pp. 260–267, 1998.
- [2] E. Novillo and P. Lu, “A Case Study of Selected SPLASH-2 Applications and the SBT Debugging Tool,” in *First International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD), in conjunction with the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, April 22–26 2003.
- [3] M. Martonosi, A. Gupta, and T. Anderson, “Memspy: analyzing memory system bottlenecks in programs,” in *SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pp. 1–12, ACM Press, 1992.
- [4] N. Nethercote and J. Seward, “Valgrind: A program supervision framework,” in *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, (Boulder, Colorado, USA), July 2003.
- [5] A. J. Goldberg and J. L. Hennessy, “Performance debugging shared memory multiprocessor programs with MTOOL,” in *Supercomputing '91*, pp. 481–490, 1991.
- [6] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, “Continuous profiling: where have all the cycles gone?,” *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 357–390, 1997.
- [7] A. Chan, D. Ashton, R. Lusk, and W. Gropp, *Jumpshot-4 Users Guide*, 2004.
- [8] E. C. for Parallelism of Barcelona (CEPBA), *Paraver – Parallel Program Visualization and Analysis Tool – Reference Manual*, November 2000.
- [9] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, “VAMPIR: Visualization and analysis of MPI resources,” *Supercomputer*, vol. 12, no. 1, pp. 69–80, 1996.

- [10] J. Vetter and M. McCracken, “Statistical scalability analysis of communication operations in distributed applications,” in *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, 2001.
- [11] L. A. de Rose and D. A. Reed, “Svpablo: A multi-language architecture-independent performance analysis system,” in *ICPP '99: Proceedings of the 1999 International Conference on Parallel Processing*, p. 311, IEEE Computer Society, 1999.
- [12] L. Hu and I. Gorton, “Performance evaluation for parallel systems: A survey,” Tech. Rep. UNSW-CSE-TR-9707, Department of Computer Systems, University of NSW, Sydney, Australia, October 1997.
- [13] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. S. Lim, and T. Torzewski, “Ips-2: The second generation of a parallel program measurement system,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 2, pp. 206–217, 1990.
- [14] S. Girona, J. Labarta, and R. M. Badia, “Validation of dimemas communication model for mpi collective operations,” in *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 39–46, Springer-Verlag, 2000.
- [15] R. Snelick, “S-Check: a tool for tuning parallel programs,” in *Proceedings of the 11th International Parallel Processing Symposium (IPPS'97)*, pp. 107–112, 1997.
- [16] J. Vetter, “Performance analysis of distributed applications using automatic classification of communication inefficiencies,” in *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pp. 245–254, ACM Press, 2000.
- [17] Y. Solihin, V. Lam, and J. Torrellas, “Scal-tool: pinpointing and quantifying scalability bottlenecks in dsm multiprocessors,” in *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, p. 17, ACM Press, 1999.
- [18] H.-L. Truong, T. Fahringer, G. Madsen, A. D. Malony, H. Moritsch, and S. Shende, “On using scalea for performance analysis of distributed

- and parallel programs,” in *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pp. 34–34, ACM Press, 2001.
- [19] T. Fahringer, B. Scholz, and X.-H. Sun, “Execution-driven performance analysis for distributed and parallel systems,” in *WOSP '00: Proceedings of the second international workshop on Software and performance*, pp. 204–215, ACM Press, 2000.
- [20] B. R. Helm, A. D. Malony, and S. Fickas, “Capturing and automating performance diagnosis: the poirot approach,” in *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pp. 606–613, IEEE Computer Society, 1995.
- [21] A. Espinosa, T. Margalef, and E. Luque, “Automatic performance analysis of parallel programs,” in *6th EUROMICRO Workshop on Parallel and Distributed Systems*, 1997.
- [22] A. E. Morales, *Automatic Performance Analysis of Parallel Programs*. PhD thesis, Department d’Informàtica, Universitat Autònoma de Barcelona, 2000.
- [23] S. W. Kim, I. Park, and R. Eigenmann, “A performance advisor tool for novice programmers in parallel programming,” in *Proceedings of 13th International Workshop on Languages and Compilers for Parallel Computing 2000 (LCPC00), Lecture Notes in Computer Science 2017, Springer*, pp. 274–288, 2000.
- [24] N. Mukherjee, G. D. Riley, and J. R. Gurd, “Finesse: A prototype feedback-guided performance enhancement system,” in *Proceedings 8th Euromicro Workshop on Parallel and Distributed Processing 2000 (PDP2000)*, pp. 101–109, IEEE Computer Society Press, Jan. 2000. ISBN 0769505007.
- [25] N. Mukherjee, *On the Effectiveness of Feedback-Guided Parallelization*. PhD thesis, Department of Computer Science, University of Manchester, 1999.
- [26] F. Wolf, B. Mohr, J. Dongarra, and S. Moore, “Automatic search for patterns of inefficient behavior in parallel applications.” submitted to *Concurrency Practice and Experience* by Jack Dongarra, February 2005.

- [27] F. Wolf and B. Mohr, “Automatic performance analysis of mpi applications based on event traces,” in *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pp. 123–132, Springer-Verlag, 2000.
- [28] T. Fahringer, M. Gerndt, B. Mohr, F. Wolf, G. Riley, and J. L. Träff, “Knowledge specification for automatic performance analysis: APART technical report, revised edition,” Tech. Rep. FZJ-ZAM-IB-2001-08, Forschungszentrum Jülich GmbH, Aug. 2001.
- [29] T. Fahringer and C. S. Jùnior, “Modeling and detecting performance problems for distributed and parallel programs with javapsl,” in *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pp. 35–35, ACM Press, 2001.
- [30] F. Wolf and B. Mohr, “Earl - a programmable and extensible toolkit for analyzing event traces of message passing programs,” in *HPCN Europe '99: Proceedings of the 7th International Conference on High-Performance Computing and Networking*, pp. 503–512, Springer-Verlag, 1999.
- [31] P. Bates, *Debugging Programs in a Distributed System Environment*. PhD thesis, University of Massachusetts, February 1986.
- [32] O. Hansen, “A method for optimizing large scale parallel applications,” in *28th Hawaii International Conference on System Sciences (HICSS'95)*, pp. 192–201, 1995.
- [33] O. Hansen, “A tool for optimizing programs on massively parallel computer architectures,” in *HPCN Europe 1994: Proceedings of the nternational Conference and Exhibition on High-Performance Computing and Networking Volume II*, pp. 350–356, Springer-Verlag, 1994.
- [34] J. K. Hollingsworth, *Finding bottlenecks in large scale programs*. PhD thesis, University of Wisconsin Madison, 1994.
- [35] M. González, A. Serra, X. Martorell, J. Oliver, E. Ayguadé, J. Labarta, and N. Navarro, “Applying interposition techniques for performance analysis of openmp parallel applications,” in *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, p. 235, IEEE Computer Society, 2000.

- [36] H. W. Cain, B. P. Miller, and B. J. N. Wylie, “A callgraph-based search strategy for automated performance diagnosis (distinguished paper),” in *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pp. 108–122, Springer-Verlag, 2000.
- [37] P. C. Roth and B. P. Miller, “Deep start: A hybrid strategy for automated performance problem searches,” in *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pp. 86–96, Springer-Verlag, 2002.
- [38] M. Gerndt, A. Schmidt, M. Schulz, and R. Wismler, “Performance analysis for teraflop computers – a distributed approach,” in *Proceedings of the 10th Euromicro PDP Workshop*, IEEE Computer Society Press, Jan. 2002.
- [39] M. Gerndt, A. Krumme, and S. Özmen, “Performance analysis for SVM-Fortran with OPAL,” in *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '95)*, Aug. 1995.
- [40] M. Gerndt and A. Krumme, “A rule-based approach for automatic bottleneck detection in programs on shared virtual memory systems,” in *HIPS*, pp. 93–, 1997.
- [41] M. E. Crovella and T. J. LeBlanc, “Parallel performance using lost cycles analysis,” in *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pp. 600–609, IEEE Computer Society Press, 1994.
- [42] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed, “Autopilot: Adaptive control of distributed applications,” in *HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, p. 172, IEEE Computer Society, 1998.
- [43] E. Shaffer, D. A. Reed, S. Whitmore, and B. Schaeffer, “Virtue: Performance visualization of parallel and distributed applications,” *Computer*, vol. 32, no. 12, pp. 44–51, 1999.
- [44] I. Foster and C. Kesselman, “Globus: A metacomputing infrastructure toolkit,” *Intl J. Supercomputer Applications*, vol. 11, no. 2, pp. 115–128, 1997.

- [45] I. Foster, N. Karonis, C. Kesselman, G. Koenig, and S. Tuecke, “A secure communications infrastructure for high-performance distributed computing,” in *6th IEEE Symp. on High-Performance Distributed Computing*, pp. 125–136, 1997.
- [46] O. Y. Nickolayev, P. C. Roth, and D. A. Reed, “Real-time statistical clustering for event trace reduction,” *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, pp. 144–159, 1997.
- [47] D. J. Kerbyson, E. Papaefstathiou, J. S. Harper, S. C. Perry, and G. R. Nudd, “Is predictive tracing too late for HPC users?,” in *High-Performance Computing*, pp. 57–67, Kluwer Academic, 1999.
- [48] K.-C. Li and K. Zhang, “A knowledge-based performance tuning tool for parallel programs,” in *Proc. ICA3PP’96 - 2nd IEEE International Conference on Algorithms and Architectures for Parallel Processing*, pp. 287–294, IEEE Press, 1996.