CSRD report no. 680

Center for
Supercomputing Research and Development

Performance Analysis on the Cedar System

Kyle A. Gallivan
William Jalby
Allen D. Malony
Pen–Chung Yew

September 2, 1987

University of Illinois at Urbana–Champaign
104 S. Wright Street
Urbana, Illinois 61801

CONTENTS

Performance Analysis on the Cedar System[†]

Kyle A. Gallivan
William Jalby[‡]
Allen D. Malony
Pen–Chung Yew

September 2, 1987

ABSTRACT

To understand the complex interactions of the many factors contributing to supercomputer performance, supercomputer designers and users must have access to an integrated performance analysis system capable of measuring, analyzing, modeling, and predicting performance across a hierarchy of details and goals. The performance analysis system being developed for the CEDAR multiprocessor supercomputer embodies these characteristics and is discussed in this paper.

Keywords: *performance evaluation, performance analysis, parallel computation, hardware monitors, instrumentation, performance presentation, programming environments.*

This paper will appear as a chapter in the book **Performance Evaluation of Supercomputers**, edited by Dr. J.L. Martin, to be published by North–Holland in January, 1988.

[†] This work was supported in part by the National Science Foundation under Grant Nos. US NSF DCR84–10110 and US NSF DCR84–06916, the U.S. Department of Energy under Grant No. US DOE DE–FG02–85ER250001, a donation from the IBM Corporation, and the Center for Supercomputer Research and Development.

[‡] Present address: INRIA, domaine de Voluceau – Rocquencourt, B.P.105 78150 Le Chesnay, France.

1. INTRODUCTION

Over the last ten to fifteen years there has been an amazing increase in the availability and affordability of computing power. Initially, improvements in hardware were largely responsible for performance enhancement; first, by simply increasing the speed of standard component designs and later by allowing the cost–efficient implementation of novel architectural concepts. This advancement in hardware technology spurred the commercial development of software techniques, which had resided in academia for many years, thereby establishing improvements in software as a legitimate, albeit subservient, component in the march towards high–performance. Recently, the domination of hardware factors in improving computer performance has been reduced due to a decrease in semiconductor–based performance enhancements (which had been delivering an order of magnitude increase every seven years). This has allowed software improvements (both system and application software) to assume equal status with hardware. As a result, new high–performance general purpose systems, especially supercomputers such as CEDAR [1], must rely on the synergism of a complex combination of techniques from the areas of architecture, hardware, systems software (compilers and operating systems) and applications software for further performance gains.

Regrettably, during the early period of explosive growth in computing power, complacency arose concerning the development of performance analysis systems from which the supercomputing community is only now beginning to recover. In particular, commercially available supercomputers suffered a dearth of tools for users interested in determining the most effective way of exploiting the high–performance processing capabilities of the machines. The major reason for this was, of course, economic. There was little or no commercial incentive for manufacturers to include such tools; the market consisted of buyers (mostly government laboratories and large corporate research centers) that were willing to pay large sums of money for the computing power

and to shoulder the burden of not only designing and tuning new algorithms for their applications, but also, in some cases, designing and implementing their own operating systems.

This is not to say that performance evaluation and analysis was not pursued at all during this period. Much technically useful work on performance analysis systems was done; e.g. the C.mmp [2] and Cm* projects [3], and the Erlangen general purpose array [4]. Unfortunately, the institutions conducting this research lacked the economic wherewithal to proceed beyond the preliminary implementation stages. Without interest from industry, a successful transfer of technology was impossible.

Manufacturers and users of commercial supercomputers, during this time, were chiefly interested in performance evaluation (not analysis) for the purposes of comparing different machines in order to make marketing and purchasing decisions. Naturally, such work centered on choosing fair and appropriate metrics and developing benchmarking methodologies. Such work is important for determining the most efficient use of monetary resources, certainly a worthwhile cause, but it contains little technical merit with respect to systems design and application code tuning. For these latter purposes it is not important to ask, "Who won?"; but "How did they win?" (the architecture, system software and application code techniques used); and "Why did they win?" (an in–depth analysis of the interaction of the algorithm, architecture and system software that demonstrates why the techniques used were successful). The pursuit of answers to the last two questions is greatly facilitated by the inclusion of an integrated performance analysis and evaluation system as early as possible in the design stage of the supercomputer. Indeed, as supercomputers grow more complex it will be increasingly difficult to retrofit any system capable of yielding significant performance information.

So with the advent of complex computing systems which combine advances in architecture, hardware, systems software and applications algorithms and software to achieve high

performance has come a renewed interest in the development of performance analysis capabilities to aid in improving the dismal ratio of the performance achieved by ordinary users to the potential performance of the system. A recent step toward improving the performance analysis and evaluation situation in supercomputing is the approach proposed by Kuck and Sameh [5]. They discuss the use of a hierarchy of codes, from kernels to full applications, to experimentally characterize the workload and performance of supercomputers. They also recommend the establishment of a public database which would contain: results of benchmarks obtained for the various combinations of codes and machines; detailed information on the techniques used to obtain the reported performance; and, most importantly, analyses concerning the algorithm/architecture/hardware/software interaction. The creation of such a database can build on the benchmarking effort, which has so preoccupied the supercomputing community recently, and serve as a vehicle by which a supercomputing benchmarking discipline can develop. Due to the inclusion of the detailed discussion of the techniques used and the analysis of the interaction of the four components of the system, it will provide a public forum for information which will stimulate the development of more sophisticated performance analysis systems for supercomputers.

The degree of success that such efforts achieve depends largely upon the availability of powerful performance analysis tools. The approach to performance analysis on the CEDAR system is presented in this paper. Discussions of the performance analysis strategy used as well as the measurement tools and future plans are included.

2. CEDAR DESCRIPTION

The CEDAR system being developed at the Center for Supercomputing Research and Development at the University of Illinois is a parallel supercomputer that combines advances in vector processing, multiprocessor parallel architectures, hierarchical memory systems, high–speed interconnection networks, restructuring compiler technology, and scientific applications algorithm design. The following sections describe the CEDAR architecture and hardware, the software components, and the applications development.

2.1. CEDAR Architecture and Hardware

The overall hierarchical architecture of the CEDAR system is shown in Figure 1. At the lowest level of the hierarchy are the Computational Elements (CE's) which are pipelined vector processors. The next level of the hierarchy comprises several clusters of CE's. Each cluster is a set of CE's and a local cluster memory. The highest level of the CEDAR architecture is formed by coupling several clusters together via a global shared memory.

At present, a single CEDAR cluster is a slightly modified ALLIANT FX/8. The ALLIANT FX/8 architecture combines vector and parallel capabilities with a two–level memory structure (cache and main memory). Each CE (up to eight are possible) is a register–oriented CRAY–1–like vector processor. The instruction set supported is that of the Motorola 68020 enhanced with a set of high–level vector instructions.

A separate concurrency control bus shared by the eight CE's facilitates rapid synchronization. Special instructions which use this bus allow the eight CE's to perform self–scheduling of parallel loops by dynamically assigning loop iterations to available processors at run time and provide a fast mechanism for enforcing data dependencies between loop iterations.

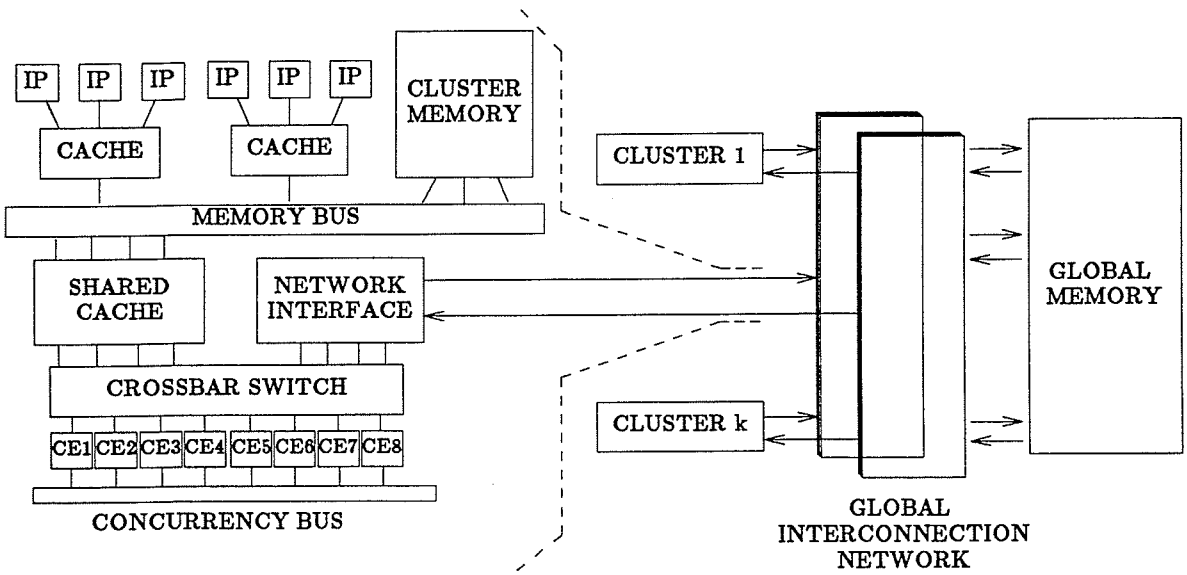

Figure 1. The CEDAR Architecture

The eight CE's share a cache and cluster memory in order to provide fast access to local data and reduce traffic to the global memory. A crossbar switch is used as a cluster interconnection network which affords each CE access to the cache and to ports to the global interconnection network (one port for each CE in the cluster). At the interface between a CE and its global network port, there is a data prefetching facility, so array data can be prefetched from the global memory into a data buffer.

The global interconnection network consists of two unidirectional interconnection networks. One connects processors to the shared global memory. The other connects the global memory back to the processors. The networks are packet–switching omega networks. They are pipelined and use wide data paths for packet transmission. Buffers are provided in each switching element to increase network bandwidth.

The globally shared memory comprises multiple memory modules. Data are interleaved across memory modules to allow high memory bandwidth. The global memory is used to store globally shared data and instructions. There is no hardware–managed cache between a processor and the global memory. Instead, data can either be demand paged into cluster memory from the global memory by operating system, or moved into cluster memory via a move instruction in a user's program. A synchronization processor is present in each memory module which implements a set of synchronization operations providing efficient low level synchronization capabilities between processors in different clusters and a method of enforcing data dependences. The indivisibility of these synchronization operations is maintained in each synchronization processor.

2.2. CEDAR Software Components

The CEDAR system supports three levels of parallelism: vector, loop, and task parallelism. Task parallelism is large grain parallelism that allows parts of a program to execute asynchronously across CEDAR clusters. Medium grain parallelism is present when the cluster processors cooperate in the execution of a DO loop. The finest level, vector parallelism, allows multiple elements of an array to be worked on using a single instruction. To exploit the parallel processing capabilities of the CEDAR architecture, a multitasking operating system, parallel languages, and restructuring compilers are being developed.

2.2.1. Xylem Operating System

The CEDAR operating system, **Xylem**, is a modification of Alliant's Concentrix™ operating system extended for multitasking and virtual memory management of the CEDAR memory hierarchy [6]. A Xylem process consists of one or more *cluster–tasks*. Each cluster–task executes on a single CEDAR cluster. Multiple cluster–tasks execute asynchronously across the CEDAR system. Xylem provides system calls for starting and stopping tasks, and waiting for tasks to

finish. System calls are also provided for coarse–grained inter–task synchronization. In addition to multitasking, Xylem supports multiprogramming whereby multiple processes can be executing simultaneously.

The Xylem virtual memory system provides convenient access to the CEDAR physical memory hierarchy [7]. Each cluster–task of a Xylem process has its own virtual address space made up of fixed size pages. Each page has attributes that indicate *accessibility* (**shared** or **private**) and *locality* (**global** or **cluster**). Xylem implements run–time mechanisms for dynamic memory allocation, memory attribute manipulation, and caching of global memory data in cluster memory with incoherency detection. The Xylem assembler, *xas*, supports the definition of virtual memory sections, the assignment of attributes to sections, and the placement of data in sections [8]. The mechanisms implemented by Xylem will be used primarily by libraries and compilers, providing convenient and appropriate interfaces and services for user programs.

2.2.2. Languages

Fortran is the focus of language and compiler development for CEDAR because of its dominance in scientific programming. However, other languages such as C, Lisp, and Prolog are also being pursued. All CEDAR languages are being designed and implemented to exploit the hierarchical structure of CEDAR and the multitasking capabilities of the Xylem operating system.

CEDAR Fortran is derived from Alliant FX/Fortran with extensions for memory allocation, concurrency control, multitasking, and synchronization [9]. New data type specification statements reflect the Xylem memory access and locality structure. Vector concurrency is available through array section notation, conditional vector statements, and vector reduction functions. DOALL and DOACROSS constructs specify parallel execution of loop iterations on proces-

sors within a single cluster task or spread across multiple cluster tasks. Multitasking routines provide an interface between CEDAR Fortan and Xylem for task creation and control. A set of synchronization functions allows access to the CEDAR hardware synchronization primitives. Cray–style synchronization operations are also provided. Multitasking and synchronization routines are implemented as part of a CEDAR Fortran run–time library.

2.2.3. Compilers

Development of automatic language restructuring tools is a major component of CEDAR's software design as a result of the work on the **Parafrase** restructurer [10] [11]. Compiler optimizations for vectorization, parallelization, and memory allocation are being developed for the CEDAR machine. The restructuring and compilation strategy is depicted in the organization of the CEDAR Fortran compiler shown in Figure 2. The parallel language program together with restructurer directives are input to a source–to–source restructurer which produces a new source program that can be compiled and executed. The restructuring process is iterative, allowing the user to select additional or different optimizations if desired. The final restructured source program is passed through a CEDAR language pre–processor to a back–end compiler.

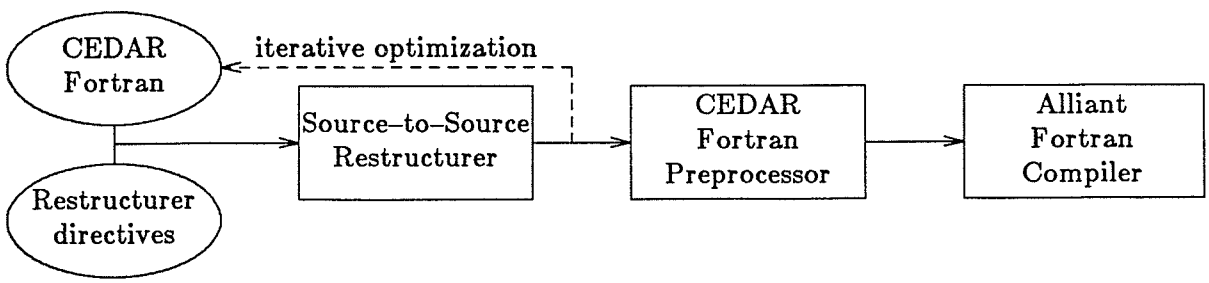

Figure 2. The CEDAR Fortran Compiler

2.3. Applications

The driving force of supercomputing performance is, of course, various applications in engineering and science. The goal of the CEDAR system is to provide high performance over a wide range of applications, in contrast to the high peaks and low valleys delivered by traditional vector machines.

A number of application areas have been selected for consideration: lattice gauge computations (QCD), quantum chemistry, weather simulation, computational fluid dynamics, adjustment of geodetic networks, inverse problems, structural mechanics, electronic device simulation, and circuit simulation. These range from purely research topics in physics through physical simulations of great practical interest to various engineering design problems. From these applications, a number of basic algorithms have been extracted: sparse linear system solvers, algorithms for linear least squares (direct and iterative), nonlinear algebraic system solvers, sparse eigenvalue problem solvers (standard and generalized), FFT, rapid elliptic problem solvers, multigrid schemes, stiff O.D.E. solvers, Monte–Carlo schemes, and integral transforms. These basic algorithms are typically 100 to 1000 lines of Fortran code each and form the computationally intensive kernels of the applications mentioned above.

The present effort on the CEDAR project with respect to applications comprises two parallel but intimately related tasks. The first is the analysis of the behavior of the basic algorithms and the implementation of CEDAR versions via CEDAR–specific high–performance kernels. These algorithms and kernels form libraries which are used as building blocks for new applications codes. An example of this type of activity is the analysis of the BLAS3 primitives and their use in *block* algorithms for dense linear algebra algorithms [12] [13]. The second task investigates specific application codes to identify the computationally intensive constituent algorithms, their interaction and the data/computation flow of the code and then transforms the code

appropriately to achieve high–performance. These transformations can range from the replacement of the old versions of the computationally intensive kernels with CEDAR library versions (possibly implementing a new kernel if it does not already appear in the library), to complete redesign of the data structure and computational flow of the code, or even to the development of the a new approach to solving the problem. These two tasks require a large range of performance analysis capabilities and the CEDAR performance analysis system must accommodate these needs.

3. CEDAR PERFORMANCE ANALYSIS STRATEGY

The primary goal of CEDAR's design and construction is to demonstrate that supercomputers of the future can deliver high performance across a wide range of applications and be accessible to ordinary users [14]. The CEDAR performance analysis strategy to aid in achieving this goal comprises three main tasks: establish a general methodology for performance investigation on the CEDAR system; design and build integrated performance measurement and analysis tools; effectively combine the methodology with the tools to form a cohesive performance analysis system integrable with other code development tools such as debuggers, restructuring compilers, and applications expert systems, into a complete supercomputing programming environment. Each of these tasks are considered in turn in this section.

3.1. Methodology

Achieving CEDAR's goal by effectively balancing the influence of its architecture, hardware, software, and applications critically depends on identifying the key performance parameters of the system, measuring and analyzing their interactions, synthesizing observed behavior into performance models, and applying performance hypotheses to predict the effects of new designs. Although the architecture, hardware, software, and applications designers sometimes have conflicting performance priorities (for example, throughput versus response–time), a comprehensive performance analysis methodology establishes a framework for reconciling these differences in order to approach the ultimate performance goals of the CEDAR system. The general approach adopted for CEDAR is organized into four strongly coupled areas: experimentation, analysis, modeling and prediction. The subtleties of the definitions of each of these areas and the interaction between them varies with the point in the design cycle under consideration.

The experimentation phase essentially comprises two tasks. The first is to verify or refute assertions made in the latest prediction phase with respect to the model and/or system components under consideration. The second is to gather information indicating which system parameters are the dominant influences on the performance metric being used. The completion of these tasks provides the analysis phase with information upon which recommendations for changes to the applications code, architecture or system software can be based. It also provides information which indicates how the model of the complex system being used needs to be updated to include more significant parameters and eliminate those which are increasing the complexity of the investigation unnecessarily. The forms of experimentation, which depend on the type of model used, and its subphases are discussed below in relation to the tools required for this phase.

The analysis phase is the investigative part of the methodology which attempts to understand and qualify the results of the latest experimentation phase. It is here that the determination is made as to whether or not the assertions concerning the model and system performance as well as the other decisions mentioned in the experimentation phase description are accurate. The analysis activity is intimately related to the data reduction and presentation subphase of experimentation (see below). Indeed, when designing tools for the two activities it becomes clear that their division is sometimes an artificial conceptual convenience. A highly–interactive analysis environment is crucial. In fact, expert system technology could be exploited by guiding analysis efforts from a knowledge base composed of basic notions of good and bad performance behavior. The analysis phase is also responsible for the assimilation of relevant performance information from the activities of the other design groups; for example, at some point the applications group must take into account any changes that have occurred in the architecture or new hardware performance capabilities.

The modeling and prediction phases are so closely related that they will be discussed in tandem. At each iteration of these phases, a model or combination of models is postulated with a set of performance metrics and a set of assertions about the various levels of performance expected and the way in which these levels differ from the set of the previous iteration. (Presumably this difference will be an improvement in the sense that the new assertions are closer to the global performance goals than those of the previous iteration.)

This activity varies considerably in form and purpose over the design process of a system. Initially, when the paper design of the major components of the system is underway, this activity tends to consist of models which are inputs to performance simulators such as Parafrase and the CEDAR Performance Prediction Package (see below) and assertions, which, although qualitative in nature, are much more detailed than those that will appear later in the design cycle. As components are designed in detail and are implemented in hardware the models progress to input for extremely detailed behavior simulators, e.g. circuit simulation packages, and finally to the most exact model of all, the piece of hardware itself. The prediction phase assertions grow correspondingly detailed.

When the major components are implemented in hardware and the hands–on application code development begins, the form the activity changes again. The models and the predictive assertions assume a dual nature each consisting of a component relating to the extant hardware implementation and a component relating to a simplified qualitative model whose purpose is not to provide the detailed behavior information of the simulations of the earlier portion of the design cycle, but to serve as a simplified conceptual model upon which algorithm and kernel integration can be based. As a result, the corresponding experimentation phase takes on this dual aspect. Each experiment consists of performance benchmarks on the implemented version of the system (a process much more detailed and system–specific than the normal machine com-

parison benchmarking activity) and evaluation of the qualitative model over a sufficiently fine mesh in the design space. (Automated experimentation tools do much to guarantee that the sampling mesh is dense enough by relieving the incredible tedium that usually accompanies such an activity.) The analysis phase then decides how accurate the model is and under what conditions it can serve as an element in a performance characteristic database.

The development of such qualitative models based on empirical observation is a crucial component of the CEDAR performance analysis strategy for developing high–performance application codes. Such models are usually based on some set of simplifying assumptions specific to the kernel, algorithm, architecture and the interaction thereof. Examples of such modeling efforts are contained in [13], [15] and [16]. A collection of these models of algorithm and kernel performance along with similar models for the performance of certain key components of the system form a performance encyclopedia of building blocks. This information can be used to facilitate speculation concerning the benefits of altering the implementation of an algorithm or application code thereby saving valuable time by pruning nonproductive avenues of investigation from the set of alternatives.

3.2. Performance Analysis Tools

In the past, designers have developed performance analysis tools primarily on an as–needed basis resulting in a random collection of incomplete, inconsistent tools instead of an integrated performance analysis package. The complex interaction of the components of new supercomputing systems renders such a collection of limited value. The design of performance analysis tools must be carefully considered to provide the foundation of an effective performance analysis system. In this section, the characteristics and structure required of tools in the CEDAR performance analysis system as well as the tradeoffs in their implementation are discussed. Later sections will describe the tools presently in the CEDAR performance analysis system which attempt

to satisfy these requirements.

The most important characteristic of the CEDAR performance toolset is that it is integrated. Essentially, this means that the toolset is a basic component in the design of the CEDAR system, it contains tools that interact in a coordinated fashion, and it provides a framework for the addition of future tools. An equally important characteristic is adaptability. Not only must the toolset be able to accommodate the different viewpoints of the major design groups (hardware, software, and applications), but it must be able to adjust to the requirements of any particular performance analysis investigation. This implies that the tools cover a hierarchy of performance analysis issues ranging from low–level details (e.g., cycle–by–cycle processor activity) to system–level questions (e.g., the performance of cluster task scheduling). For many performance analysis investigations, passivity of the tools is a critical requirement to insure accuracy. Because the performance analysis sophistication of the users can also vary, the performance tools should also strive to be highly–interactive, user–friendly, and able to present performance information in various levels of detail. Finally, the performance analysis system should be designed in such a way as to anticipate its inclusion in a general supercomputing programming environment.

The automation of modeling, prediction, and analysis tools requires that the knowledge of supercomputer designers be captured in an expert system framework. This has yet to happened, however steps can be taken in this direction by collecting and merging systems of various types present in the design, development, and use of a supercomputer. Such an effort is underway on the CEDAR project.

Most performance analysis tools are concerned with the experimentation phase of the methodology. Conceptually, there exist two types of experimentation tools: simulative and empirical. Throughout the system development process, there is a need to be able to work with

the design in the abstract. Simulative tools are those that model and simulate the architecture, hardware, software, and applications at various levels of logical and operational detail without requiring measurement of actual system components. Such tools are common for circuit and board–level simulation of hardware designs, and general–purpose modeling and simulation packages also exist.

For the CEDAR system, a large spectrum of simulative tools are required, from very high–level architectural and software modeling tools, crucial early in the system's lifecycle, to detailed simulators more closely reflecting the evolving characteristics of various component designs, to low–level emulations of actual operation. The use of such tools is not relegated to certain periods of the design cycle but continue to be used and refined throughout the system's development. For example, the Parafrase vectorizing/parallelizing compiler developed at the University of Illinois, was an early architecture and software design tool for the CEDAR system. It continues to be important in studying source code restructuring for parallelism and is currently being adapted to serve as a performance optimization component of the CEDAR compilation process. Similarly, various simulators, including the CEDAR Performance Prediction Package (CPPP) [17], assisted in the investigation of tradeoffs in the CEDAR architecture during initial processor, interconnection network, and memory design; and recently its powerful trace generation facility has been used in cache simulations and task scheduling studies.

The most common notion of performance analysis tools, those that involve experiments with actual components of the system, are referred to here as empirical experimentation tools. Generally, empirical experimentation involves three components: specification, instrumentation/data collection, and data reduction/presentation. The specification phase defines the events and/or performance characteristics which are to be investigated in the experiment, possibly in an abstract fashion. The performance–directed events defined in the

specification must be realized by instrumentation tools that monitor system operation and collect performance data associated with event occurrence and duration. Implicit in this process is the translation of event definition into the appropriate combination of monitoring points and actions in order that information concerning the specified events can be recovered in the data reduction and presentation phase. For instance, if a user desires the computational rate of a kernel in floating point operations per second, this abstract performance metric must be translated in the instrumentation phase into floating point operation counts and elapsed kernel execution time measurement. The value of the metric is recovered in the data reduction and presentation phase. Since events manifest themselves in various ways, the instrumentation and data collection tools must be flexible and extensible so as not to restrict the specification. Critical to the empirical experimentation process is the development of tools for rendering the collected performance information in meaningful ways. This involves both reduction and presentation of the monitored data with respect to the higher–level event description contained in the specification. Clearly, user interaction is critical in the empirical experimentation process in order to guide performance characterization and optimization efforts.

As in all aspects of supercomputer design, there are tradeoffs to be made in the implementation of performance analysis tools. For example, the degree to which desired characteristics can be satisfied for most of the tools depends heavily upon the cost one is willing to absorb. Similarly, a tradeoff typically exists between the passivity of a data collection/event detection tool and its accuracy. Perhaps the most important tradeoff, however, is integration versus complexity. The higher the degree of integration, the more complex the design and implementation of performance analysis tools becomes.

3.3. Application of the CEDAR Performance Analysis System

Methodology and tools are intimately related in the performance analysis strategy. Postulating some grand methodology in the absence of tools is useless if the objective is to optimize the performance of real codes on a real system. Likewise, using tools without some methodology leaves the success or failure of the valuable performance optimization activity to little more than pure chance. The interaction of the two, however, can lead to an efficient solution of the performance optimization problem. This section presents an overview of the application of the CEDAR performance analysis system to the two basic activities of the applications group outlined in the previous section.

The hierarchical nature of the CEDAR architecture tends to cause codes to be designed in a similar modular hierarchical fashion. The hierarchical nature of both complements that of the tools. These relationships result in similarly structured application performance analysis activities.

Recall that the first basic task of the applications group activity involved a bottom–up approach to application code implementation. In this approach, an encyclopedia of performance information is formed comprising qualitative models from low–level architecture/computation interaction such as: single–CE computational behavior (e.g. parallelism available at the arithmetic expression level via chaining or operation spreading), low–level memory behavior (e.g. the influence of spatial and temporal locality, and memory reference patterns), and the tightly–coupled parallelism within a single cluster (e.g. vectorization, concurrency, synchronization, and communication); through computationally intensive kernels such as matrix multiplication; up to constituent algorithms such as dense linear algebra computations. The models in the encyclopedia are in turn used to make qualitative performance statements about application codes as well as algorithms and kernels which are candidates for inclusion in the encyclopedia. Typically,

these statements are made using the tools and methodology above using a decoupling technique in the analysis phase.

An example of such an analysis, is the study performed at CSRD concerning the implementation on a single CEDAR cluster of the block methods for dense linear algebra computations in terms of matrix multiplication kernels [13]. Extensive use was made of analytical methods and the empirical experimentation tools designed for a single CEDAR cluster (see below). The results of this study showed that the influences on performance of the hierarchical memory system and the mapping of the algorithm to the computational resources, could be decoupled, analyzed separately in a simple fashion, and then reconciled to yield near–optimal algorithmic parameters, for both kernels and block algorithm, as functions of system characteristics such as cache size, number of processors, vector register lengths, and so on.

This analysis clearly demonstrated that valid qualitative conclusions can sometimes be reached concerning the behavior of phenomena of great complexity by assuming a superposition principle holds for the performance of an algorithm in terms of the performances of its constituent parts, i.e. the interaction between components can be safely ignored. This is, of course, true if the granularity of the parts is sufficiently large, but in some cases, where granularity is not necessarily large, the algorithmic transformations can be used to reduce the significance of the interaction to a level such that it can be ignored when choosing algorithmic parameters. Furthermore, the conclusions in this study were at a level of abstraction appropriate not only for a single CEDAR cluster, but for many multivector processors with a hierarchical memory system.

Finally, these results demonstrate the ability of the analysis activity of one group to yield benefits for another. In this case, the techniques can be generalized and applied as a compiler analysis technique to determine the best way to exploit a hierarchical memory system.

The second task of the applications group involves a top–down analysis and design of application codes. Whereas the bottom–up approach above, tended to make use of performance analysis tools monitoring low–level behavior of the cluster, this approach makes extensive use of the high–level program analysis tools described in a later section. Typical activities are: identifying major computational components, if any, via event–trace–based profiling tools, investigating the parallel structure inherent in the code and exploitable by restructuring techniques via data flow analysis by tools such as Parafrase, and investigating the influence of scheduling and synchronization on the dynamic behavior of the code via virtual execution flow analysis. Clearly this approach can benefit from the integration of these tools into an expert system–based supercomputer programming environment.

4. SIMULATIVE EXPERIMENTATION TOOLS FOR CEDAR

As described above, simulative experimentation tools support the performance analysis of supercomputer architecture, hardware, software, and applications without the need for measuring empirical data directly from the system. Simulative experimentation tools have been used extensively in the design of the CEDAR system. Investigations making use of such tools have included studies of compiler restructuring techniques [18], memory access combining [19], compile–time and run–time scheduling [20] [21], and prefetching for cache memories [22]. In this section, details of two important simulative experimentation tools are discussed. The implementation and application of these two tools are evolving with the CEDAR design.

4.1. Parafrase

During the feasibility study and early design phase of CEDAR, much effort was devoted to evaluating various aspects of system performance. Of particular concern, was the performance gains achievable through restructuring of programs, originally written for uniprocessors, for parallel execution on the evolving CEDAR architecture. The source–to–source parallelizing compiler Parafrase served as an invaluable simulative experimentation tool not only for exploring new restructuring techniques, but also for giving first–order estimates of the amount of parallelism exploitable by programs with respect to various machine architectures [11] [23].

Four basic architectures are used in Parafrase to represent target machines for the program transformations:

SES – Single Execution of Scalar instructions

SEA – Single Execution of Array instructions

MES – Multiple Execution of Scalar instructions

MEA – Multiple Execution of Array instructions

Minor variations from these four basic types include representations of special recurrence solving hardware and special synchronization instructions to handle DOACROSS loops.

Parafrase has been supplemented with timing passes that estimate the execution time of a program on a sequential machine, T_1, and the execution time on the target machine organization with p processors, T_p. In addition, the speedup of the program, $S_p = \frac{T_1}{T_p}$, and its parallel efficiency, $E_p = \frac{S_p}{p}$, are calculated. The user can specify conditional branch probabilities and loop bounds to aid Parafrase in determining these performance measures. Parafrase also generates operation counts and indicates the number of statements enclosed in each level of loop nesting.

Although the performance information obtained from Parafrase can be used to infer the performance of real machines similar to the general architectures, in particular CEDAR (an MES type machine), Parafrase provides only static first–order estimations of actual run–time performance [23]. Performance degradations due to scheduling delays, memory and interconnection network conflicts, and synchronization overheads are not taken into account.

4.2. CEDAR Performance Prediction Package

The CEDAR Performance Prediction Package (**CPPP**) is a hierarchy of simulative experimentation tools that provide the user several alternatives for predicting the performance of the CEDAR design with different tradeoffs in accuracy and cost [17]. The organization of the CPPP is shown in Figure 3. The CPPP can be used to tune the design of the CEDAR architecture, compilers, and operating system, as well as aid application development and algorithm design.

The highest level of the CPPP hierarchy is Parafrase and its static performance estimates discussed above. Measurements made by Parafrase are the least accurate but cheapest to

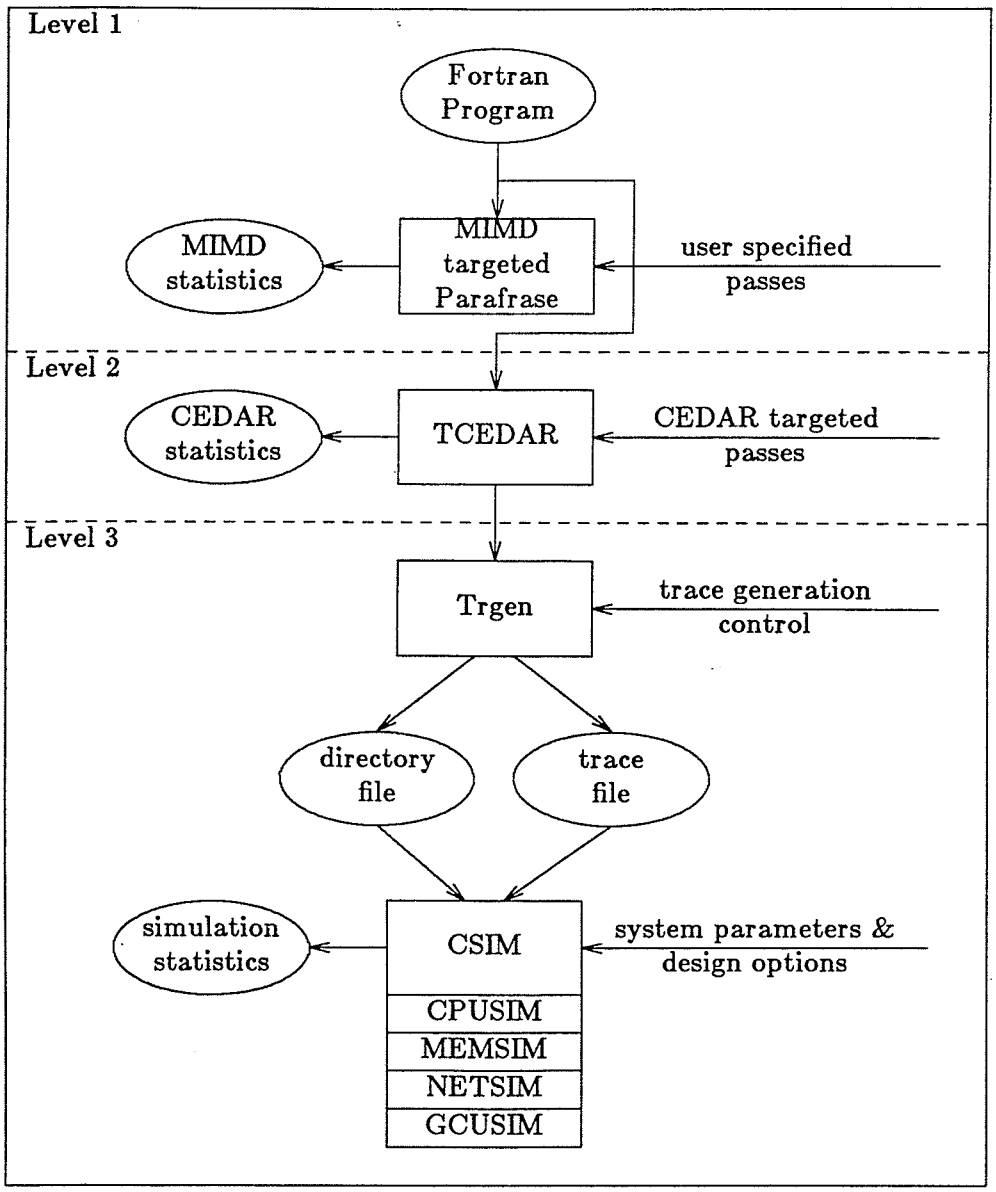

Figure 3. The Three Level CPPP Hierarchy

produce. **Tcedar** is the next level in the CPPP hierarchy [24]. It is implemented as an additional pass in Parafrase. Tcedar produces more accurate performance statistics, at a negligible increase in cost, by incorporating a simplified model of the CEDAR computational processors dis-

tinguishing between local and shared memory references. The performance statistics produced include execution rate in MFLOPS, the number of operations of different types, and the number of references to local and shared memories.

CSIM is a parallel trace driven simulator of the CEDAR machine and represents the lowest level in the CPPP hierarchy. Every component of the CEDAR system has a parameterized model in CSIM. A trace generator, **Trgen**, extracts a parallel trace from the restructured program produced by Tcedar which becomes input to CSIM. Four major module simulators comprise CSIM: the global control unit simulator (GCUSIM), the processor unit simulator (CPUSIM), the global memory simulator (MEMSIM), and the global interconnection network simulator (NETSIM). At the time CPPP was developed, these four modules map into the major components of the proposed CEDAR architecture.

CSIM produces a large number of statistics. Overall system performance statistics include total execution time, estimated one processor execution time, and estimated program speedup. Operation execution rates are computed in MOPs and MFLOPs, and arithmetic–logical, floating point, fixed point, and boolean operations are counted. Numerous statistics relevant to the performance of each functional unit in each processor are also generated. The network statistics include: number of requests serviced, network throughput, average network delay, maximum network delay, average utilization, average load, and average conflicts per request. For shared memory systems, the percentage of time each module is busy, the bandwidth used, and the percentages of references to local and shared memory are calculated. The average and maximum times to access global memory are computed as well.

CPPP was used to study 22 routines from the EISPACK and LINAPCK packages for a 32 processor CEDAR machine [17]. The performance degradation due to conflicts in the shared memory, the delay in the CEDAR interconnection network, and synchronization overhead were

measured. Generally, the results confirm that the architecture of CEDAR is balanced. The performance of the CEDAR network was shown to be close to that of a crossbar switch. The maximum performance degradation of almost half of the programs is due to shared memory conflicts. Synchronization is the major performance degradation factor for the other half of the programs.

The CPPP is evolving with the development of the components of the CEDAR system. It has been modified to handle synchronization instructions as well as parallel execution of multiprocessed constructs generated from macro data flow graphs. Coding of models for other hardware components of CEDAR is underway. The CPPP could also be used to generate numerous statistics relevant to the control and scheduling overheads in the design of the CEDAR operating system.

5. EMPIRICAL EXPERIMENTATION TOOLS FOR CEDAR

The CEDAR empirical experimentation toolset is a hierarchically organized group of integrated specification, instrumentation and data presentation tools. In this section, these tools are described in detail. First, the hardware monitoring system, the lowest–level set of tools, is presented. Although such a system does indeed provide powerful experimentation tools (on many systems rudimentary forms of this toolset is all that is provided), in a complex computing environment such as CEDAR this basic toolset is inadequate. The interaction between the basic hardware monitoring system and higher–level software experimentation tools needed to meet the required experimentation capabilities is discussed next. The tools for measurement specification, instrumentation and data collection, and data reduction and presentation phases of the empirical experimentation process, shown in Figure 4, are considered in turn. Finally, a brief overview of the development of the Faust supercomputing programming environment is given. The CEDAR performance analysis system presented in this paper will eventually be integrated into the Faust environment.

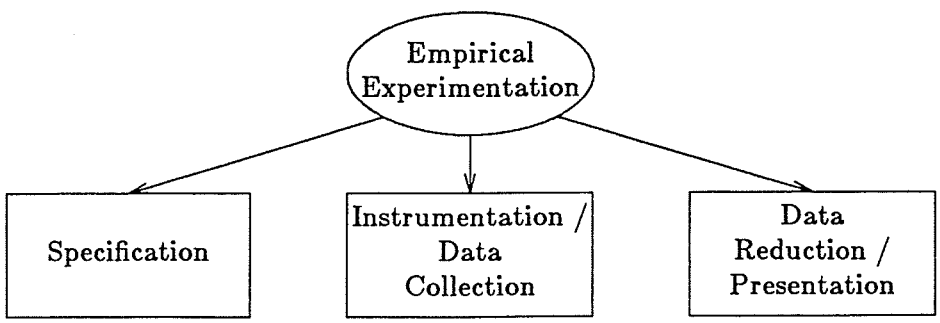

Figure 4. Empirical Experimentation Process

5.1. The CEDAR Hardware Monitoring System

A hardware monitor is a measurement device that uses high–impedance electrical probes connected directly to hardware devices on a system. The sensing of pre–defined signal patterns is used to trigger data collection on some subset of the probes. Because hardware monitors are separated from the measured system, they do not interfere with the system or alter its states.

Hardware monitors, however, can be difficult to use in some situations. There are several reasons for this. First, signals that are of interest to a user may be scattered about the system, and very often are not well documented. Since most users do not have detailed hardware information of a system, simply specifying what is to be monitored can be nearly impossible. Even if the signals and their location are known, signals internal to a chip cannot be probed. Therefore, it is a prerequisite that all of the signals that are important to performance measurement be accessible and well documented. (Hence, the pessimism concerning future capabilities of retrofitting hardware monitors on supercomputers mentioned earlier). Second, due to physical constraints of cabinets, wires, board space, and the bulkiness of hardware probes, it can be very difficult to attach a significant number of probes to a system. It is necessary, therefore, that the system be packaged so as to be instrumentable, i.e. performance measurement signals must brought out to the edge of a board and concentrated in an accessible area, preferably on a connector. Third, hardware monitors usually do not have access to software related information. For example, it is almost impossible for a hardware monitor to tell which process caused an event. This makes post–collection analysis very difficult unless the system environment is strictly controlled. Provisions must be made to allow software control of hardware monitors so software information and hardware signals can be correlated.

From the above considerations, hardware monitors can only be effective when necessary steps are taken during system design and implementation. Performance measurement must be

viewed as part of the system requirements, and those requirements must be met in both hardware and software design. Recognizing the importance of performance measurement during the hardware and software design phase of the CEDAR system has resulted in the design of the integral hardware monitoring system shown in Figure 5. The key elements of this system are buffered on–board test points, a data acquisition system, and software configuration and control.

The CEDAR system is observable by way of a series of performance measurement test points provided on each circuit board. Typically, there are from 50 to 120 test points on each board. They include information such as microcode instruction and address on the CE boards, control states and important counters on the network interface boards and other signals specific to the global network boards and the global memory boards. These signals provide the most

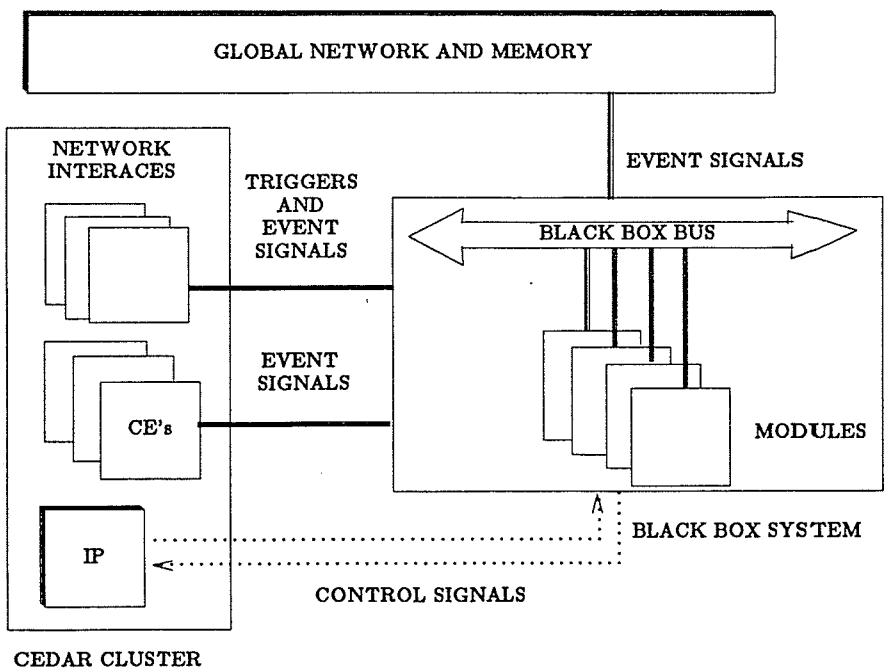

Figure 5. CEDAR Hardware Monitoring System

vital and indicative information on the activities of each board. For each of these on–board signals, a buffered copy of the signal is brought out to an accessible edge.

These signals are monitored by a general purpose high resolution data acquisition system, called the black box system. A black box card cage can hold up to 32 modules. Each module is a 5–inch by 6–inch printed circuit card and is individually addressable on the black box bus.

There are several types of modules including signal conditioning, counting, timing, data–logging. For signal conditioning, currently available modules include combinatorial and programmable logic for creating composite conditions from the measurable signals. These modules act as filters to reject data that need not be stored so the amount of data collected can be kept to a minimum. They can also be used to generate triggering signals for other modules.

For counting, timing and data–logging, there are modules available with interval timers and event counters. Each of these modules has a 32–bit timer/counter clocked at a rate of 40ns. A local memory of 32k 32–bit words is provided in each module. Each timer/counter can be started or stopped from either a user program or by the triggering signals from a conditioning module. The content of the timer/counter can be stored into the local memory (the address of the local memory will be incremented automatically) and then be reset for new events. For example, one module can be configured as an interval timer and another as a counter to count the number of floating–point operations. Using this configuration, Mflops (million floating–point operation per second) can be easily obtained for an entire program or for some program segments.

Modules also exist for program tracing. Each tracing module has a large buffered memory for storing 8–bit wide traces. The memory size is 128k 8–bit words. This memory size can be extended by cascading several tracing modules together, using the overflow signal of one module as a triggering signal for the next. Data width can also be increased by using several modules in

parallel. Other useful modules which will be implemented in the future include histogrammers, interrupt generators, ranging comparators, and sequencers (for detecting certain special sequential events).

In addition to using signal conditioning modules to trigger modules, triggers can also be obtained directly from each CE by issuing special trigger instructions. These trigger instructions can be embedded in a user's program or in the operating system so precise measurement can be started and stopped by a user, or by the operating system. These trigger instructions are actually store instructions to the globally shared memory. Each particular trigger instruction is assigned one special memory address in the global memory address space. These memory addresses are specifically reserved for these trigger instructions and can not be used for storing data. The network interface for each CE will intercept store operations to these locations and generate triggering signals accordingly. Using this technique, no special trigger instructions are needed in the CE's instruction set.

The black boxes are connected to a controller card resident in one of the Multibus backplanes used by the Interactive Processors (IP's) in a CEDAR cluster (see Figure 5). Interactive Processors are responsible for performing most of the functions for the CEDAR operating system including interfacing to peripherals. Each controller can handle as many as 16 black box systems. Multiple controllers can be installed in one Multibus (for centralized control by one IP), or in multiple Multibuses (for distributed control by several IP's, either within one cluster, or spanning multiple clusters in the CEDAR system).

Each black box is mapped onto the Multibus address space, as is each module in a black box. A controller can control and collect performance data from a module by simply moving data between different parts of the Multibus address space using simple read and write instructions. The IP software for communicating with the black box controller residing on the IP's

Multibus is a collection of operational primitives. The primitives allow an IP to directly control each black box, and to read the contents of the black box registers and memory and thus obtain the collected data. This software can be readily manipulated to handle any desired black box configuration. Since an IP is also responsible for providing disk I/O to the cluster, it is possible for run–time performance data to be logged to disk as they are collected. Real–time analysis by the IP critically depends on the volume of data received and the amount of processing to be performed. If remote monitoring is being performed (i.e. one otherwise uninvolved cluster is monitoring the actions of one or more other clusters), then the full power of a cluster can be directed toward real–time analysis.

One advantage of using such a hardware monitoring system is that it is highly modular. Each module in a black box system has an identical standard interface which makes the system easily expandable. Special types of performance modules can be designed as needed, and as long as their interface stays the same, all of these modules can be controlled by the same mechanism. This allows different types of modules to work together collecting different kinds of data simultaneously. It is often necessary to correlate data collected in different parts of a system within the same time frame. The ability to mix different types of modules together to collect different types of data simultaneously makes the post–collection analysis much easier and more accurate.

5.2. Empirical Experimentation Software Tools

While the CEDAR hardware monitoring system is a fairly powerful and versatile tool, it is unable to fulfill the CEDAR empirical experimentation requirements. The major difficulty is that its capability to detect software events, even with special triggering instructions, is far too limited to capture the extremely complex software events which are crucial to performance analysis on the CEDAR system. At the very least, software instrumentation is required to translate the high–level software events defined in the specification phase of the empirical

experimentation process into a set of measurement events. In addition, it is clear that the data reduction and presentation phase is purely a software task which transforms raw data, possibly collected by the hardware monitoring system, into meaningful statements concerning the goals of the experiment defined in the specification phase.

5.2.1. Measurement Specification

A standard supercomputer program measurement practice is to time a program's execution from beginning to end and compute MFLOPS. More adventurous measurement strategies for parallel processors time sequential and concurrent execution and calculate speedup. Because of CEDAR's multiprocessor architecture, multitasking operating system and multiprogrammed run–time environment, simple program timings are inadequate to characterize program execution. The special problem of program performance measurement in the CEDAR system is the need to detect and record data relating to the asynchronous parallel processing of multiple program tasks and their interactions with the architecture, operating system and other programs.

Table 1 lists examples of events considered significant in the makeup of program performance in CEDAR [25]. It should be emphasized that the events most important to performance are program–specific and can change depending on the level of program analysis and the granularity of program optimization. For instance, the timing of individual routines or tasks is useful for isolating time–consuming components of the program at a high–level but provides little insight into dynamic program behavior. Observing events related to loop–level parallelism, multitasking, and synchronization can uncover performance limiting concurrency and dependency problems. Performance problems can occur at still finer levels of detail, however. Because hierarchical memory usage is clearly a performance factor in CEDAR, memory reference and contention events must be monitored to determine their impact on program performance. Also, the run–time environment can cause performance to fluctuate, requiring events of this type to be

EVENTS			
Program	*Operating System*	*Architecture / Hardware*	*Run–Time Environment*
routines	task creation	processor concurrency	system load
doall parallelism	task execution	cache utilization	scheduling
doacross parallelism	task deletion	memory referencing	resource contention
vectorization	task suspension	memory contention	resource queuing
synchronization	task synchronization	bandwidth utilization	context switching
instructions	VM management	communication	I/O

Table 1. Program Performance Measurement Specification

observed and their impact assessed. The purpose of performance measurement specification is to define a hierarchical organization of program execution events which allows a programmer to select the appropriate context in which to monitor the areas of interest for a particular application.

5.2.2. Instrumentation and Data Collection Tools

Program event measurement is composed of two parts: event detection and event data collection. Detection is accomplished through hardware or software instrumentation depending on event accessibility. Hardware event detection is preferred to minimize impact on program behavior. Standard event data collection is generally based on two metrics: i) the number of times an event X occurs ,$n(X)$, and ii) the total amount of time X lasts, $t(X)$. The common measurement practice is to detect when an event occurs, increment the associated event counter, and update the running total event time by the interval of time the event is present. However, event counts and times only summarize program performance and are unable to describe time–dependent program execution. To fully analyze program performance in the CEDAR environment, the ability to observe the time–ordered sequence of asynchronous concurrent program

execution events is required. Therefore, the standard event counting and timing measurements are complemented in CEDAR by the recording of event traces [26]. This enables program performance analysis tools based on traced event data to determine both the level of performance obtained as well as investigate reasons for the observed performance.

Tracing in CEDAR is based on the simple operation of saving the *ID* of an event, plus a high–resolution time–stamp, in a sequential buffer whenever the event occurs during program execution. Let *X(i)* denote the *i*th occurrence of event *X* and *T(X(i))* the time–stamp of *X(i)*. The trace of event *X* is denoted as $\tau(X)$. Given τ, we can compute *n(X)* by adding up occurrences of event *X* found in the trace; $n(X)=\mid \tau(X) \mid$ Computing *t(X(i))* from $\tau(X)$, however, requires that we know when *X(i)* begins and when it ends. Each event is defined by two sub–events: $X_B(i)$ is the sub–event "*X(i) begins*", and $X_E(i)$ is the sub–event "*X(i) ends*". *t(X(i))* can then be computed as $T_B(X(i))-T(X_E(i))$.

Software support for tracing primarily exists as instrumentation in the user program and the operating system to signal software events such as routine entry/exit, synchronization operations, different states of task execution, and other complex events. Software trace data buffering is also provided with a caution about its effect on program performance behavior. The hardware performance monitor organization easily supports the implementation of trace buffering and time–stamping facilities for each processor in the CEDAR system, thus allowing tracing of complex software events to be implemented in a reasonably passive manner. As with the hardware monitoring facility, enabling of software instrumentation is integrated with the CEDAR programming environment and run–time system. The effectiveness of event tracing comes from the ability to measure and analyze a program at several levels of execution. Here, as with the entire experimentation activity, programmer interaction is important. In addition to defining the granularity of the events to be traced, the programmer can perform selective tracing to evaluate

certain periods of the program's execution.

High–resolution, globally–synchronized time measurements are critical in CEDAR for establishing the exact dynamic flow and timing of concurrent events. Individually–maintained processor traces must be attributed time–stamps based on a central real–time reference of sufficient resolution to accurately merge the traces into a single time–ordered stream of events. In order to establish virtual program execution time measurements in CEDAR's multiprogrammed run–time environment, it is necessary to annotate the program execution event trace by task state events such that non–program execution time periods can be removed. Software tools are being developed for CEDAR to provide high–resolution timing information and to track task execution.

5.2.3. Data Reduction and Presentation Tools

The ease with which a programmer can understand and deal with the diverse issues affecting program performance depends not only on the ability of the instrumentation and data collection facility to supply the necessary performance information, but also on powerful performance data reduction and presentation tools devoted to the problem of characterizing program execution in ways that the programmer can understand and apply to optimizing program performance.

As with specification and instrumentation, data reduction and presentation in CEDAR is hierarchically organized. The highest level of presentation generalization encapsulates program performance data into summary statistics such as counts, relative frequencies, fractions of elapsed time and total elapsed time. Surprisingly, the majority of supercomputer performance analysis is based entirely on this level of presentation. More sophisticated presentation illustrates the dynamic flow of events and their interactions at various levels of detail. At the lowest level, the raw performance data gives an exhaustive insight into the intricate complexities and nuances

of program performance.

Clearly, performance analysis at each level is highly user–interactive. However, it should be emphasize that because of the heterogeneous factors affecting performance and the range of performance characterization and optimization goals, the performance analysis environment must be highly experimental, customizable and investigatory. For this reason and the fact that program event specification and measurement are continually being refined and improved, an all–inclusive list of program performance analysis tools is impossible. Instead, the following describes several tools being developed for CEDAR showing different levels of analysis and performance investigation.

Profiling is a performance analysis technique based on counting and timing routine execution to identify CPU–intensive portions of a program. Originally developed for single–processor machines, profiling is commonly implemented as interrupt–based periodic program counter sampling to determine relative frequency of routine execution, augmented by program instrumentation for routine call counting. There are two reasons why this standard profiling implementation is unacceptable for supercomputers in general and CEDAR in particular. Basing profiling on program interrupts can lead to the absurd practice of attempting to retrieve accurate program profiles on machines with cycle times on the order of ten to one hundred nanoseconds using sample intervals of one to ten milliseconds. It is amazing that such an interrupt–based profiling approach has even been considered for use in supercomputer systems. Only by measuring the time between routine entry and exit can accurate profiling statistics be achieved. The second criticism of standard profiling is its current dependence on a single thread of program execution. Obviously, information about concurrent execution is necessary to properly guide parallel program optimization efforts.

The program profile tools being developed for CEDAR are data presentation tools which generate statistical information from the program traces, thus providing accurate timing analysis and the ability to profile concurrent execution. Using only routine and task event data, all common profiling measurements can be produced including routine call counts, descendent routine call counts, and direct and cumulative execution time. In addition to these results, concurrency statistics can also be generated. These statistics include:

sequential and concurrent routine call counts
sequential and concurrent routine execution time
number of tasks created
average task execution time
execution time histogram of task concurrency
average task concurrency

Data from the other events only increases the database from which execution profile statistics can be drawn. Of particular interest is the execution time of parallel loops and synchronization operations such as:

CDOACROSS and CDOALL execution times
SDOACROSS and SDOALL execution times
loop–level synchronization counts and times
task wait synchronization counts and times
event wait synchronization counts and times
lock wait synchronization counts and times

Clearly, some profiling statistics are simpler than others to produce from the program event traces. To provide an execution profile analysis tool capable of completely summarizing program behavior is unreasonable. Moreover, complex profiling summaries can be difficult for the user to understand and apply. The CEDAR program profiling tools concentrate instead on a standard set of events, including the ones shown above, from which statistics are straightforward to generate and meaningful program characterization is possible. Other program execution flow analysis tools allow the programmer to look in detail at the program's execution behavior by analyzing the flow of events directly.

The ability to observe the program events in a time–ordered sequence of occurrence is the key difference between execution flow analysis and profiling tools. Statistical summaries give a global picture of program execution but lack the historical perspectives. Execution flow analysis provides the programmer with a window into the program traces at various levels of detail. The concept of *replaying* the program's execution with respect to the traced events forms the basis of the tool.

In general, execution flow analysis is used as a means to explore the program's execution for evidence of good, bad or interesting behavior. Sometimes the programmer just wants to see general characteristics, such as the sequence of routine execution. At other times, the programmer will apply execution flow analysis to highly specialized event traces for "search and destroy" missions to pinpoint some anomalous behavior or dissect a poorly performing section of code. Therefore, execution flow analysis in CEDAR must be a highly–interactive environment for the programmer to intelligently search and analyze the program execution trace database.

The function of moving around in the program trace and displaying events is called *event trace browsing.* The event trace browser provides various statistical and graphical presentation capabilities for displaying events that occur in different regions of the trace. One interesting graphical representation is the *dynamic call graph* which shows only the active calling arcs of the static *subroutine interconnection graph* except with the nodes being generated dynamically as the routines are encountered in the program trace. Figure 6 gives an example of the state of a task's execution on an 8–processor CEDAR cluster in the form of dynamic call graphs. The routine execution thread of each processor is shown, with active routines depicted by □'s, plus the global dynamic call graph of the task as a merge of the individual processor threads. A higher–level presentation could show the dynamic execution of tasks over time in the form of program execution graphs or time lines.

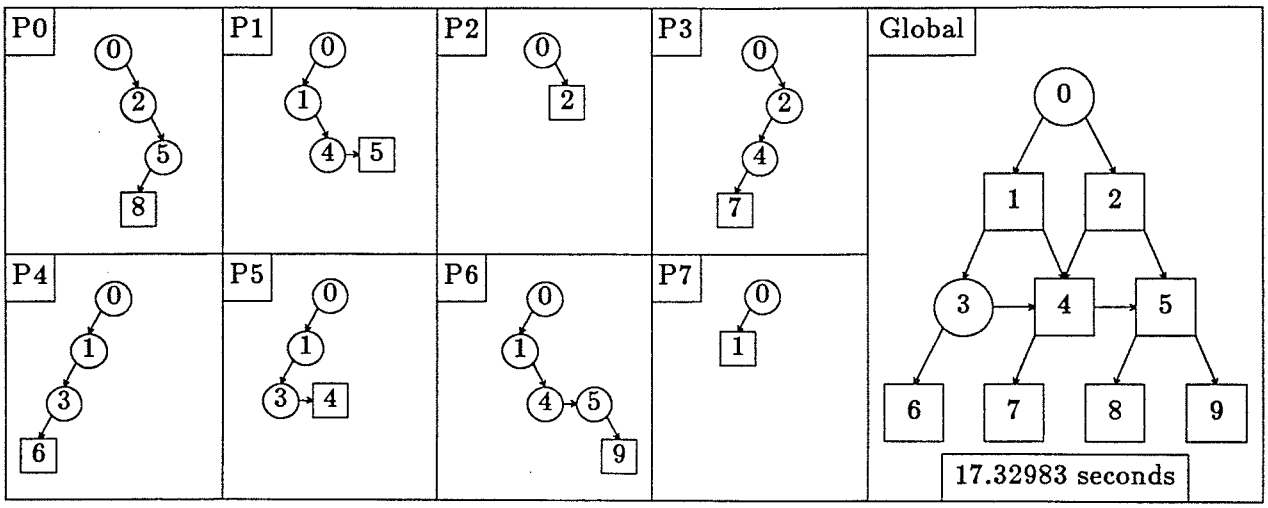

Figure 6. Multi–Thread Dynamic Call Graph

The key feature of the event trace browser is that it is interactive. The event trace browser defines the lowest level of event trace flow analysis. It takes the event specification and format information, and provides a front–end for general inquiries about program execution. Basic searching and event presentation are handled by the browser.

Execution flow generalization provides the programmer with a way of observing higher–level execution behavior not necessarily represented directly by some event. Although the browser provides minimal functionality in this area, the execution flow generalizer works with high–level events defined from combinations of the traced events. The resulting execution flow generalizations are presented in various ways to the user. For instance, processor concurrency can be analyzed from the relative occurrence of processor activity events in the program trace. Figure 7 shows a graph of processor activity that might correspond to the task of the dynamic call graph above. The programmer can quickly identify periods of reduced parallelism from the graph and then return to the browser to look at the specific events occurring at those times. Although this

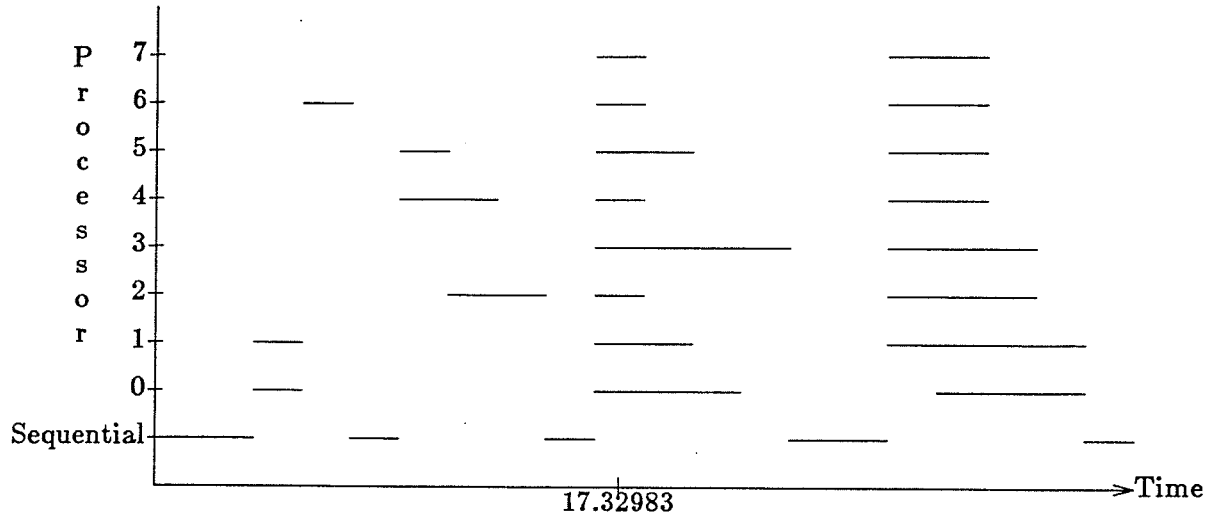

Figure 7. Processor Activity Graph

is a simple example of execution flow generalization, it illustrates the basic ideas. It would be easy, for instance, to apply the same methods to produce task concurrency graphs.

If CEDAR was a monoprogrammed system, the real–time time–stamped program event trace would be, to a first approximation, a time–accurate representation of program execution. However, CEDAR is multiprogrammed, and, unfortunately, this causes problems for the program performance analysis. Essentially, what is required is the ability to quantify and remove the effects of multiprogramming from the collected performance data. Standard approaches keep separate measurements for the different jobs, updating the values only when a job is executing. This might be acceptable for sequential programs, but the problems are exacerbated for programs composed of multiple tasks. In this case, the multiprogrammed run–time environment can actually alter the dynamic program execution flow requiring significantly more imaginative measurement and analysis techniques to assess "virtual" program performance.

Virtual execution flow analysis is an experimental performance tool being developed in CEDAR for multi–tasked programs. Its purpose is to remove the task scheduling effects of CEDAR multiprogramming from the observed program execution as represented by the program trace. The tool is based on the tracing of run–time environment events, such as task context switches and scheduling events, and operating system events dealing with different task execution states and task synchronization. The analysis attempts to identify periods of dependent and independent task execution subject to the inter–task synchronization activities present in the trace. If all task synchronization is done using primitives that are traced, virtual execution flow analysis will be able to maximally "compress" the original trace into a virtual execution trace where all inter–task dependencies remain ordered in time. Non–traced inter–task synchronization, however, can limit the ability of virtual execution flow analysis to remove multiprogramming overhead.

An example of the benefit of virtual execution flow analysis can be seen in the before and after program execution graphs of Figure 8. It is hard to discern true effective parallelism from the original program trace data displayed in the upper graph as task activity over time. Parallel execution is observed, but so are periods of complete program inactivity (represented by □'s); for this example, it is assumed that the tasks are not waiting for I/O during this period. Shorter total execution time plus more parallelism might be expected if the program were not subject to multiprogramming. Using the inter–task synchronization information from the program trace (indicated by the arrows), the not running, non–waiting periods can be removed from the task traces producing the virtual execution graph shown below. Now the "effective" parallelism is more obvious. Although this graph only approximates program execution in a non–multiprogrammed environment, it does help to estimate the degree of performance degradation and pertubation due to system load.

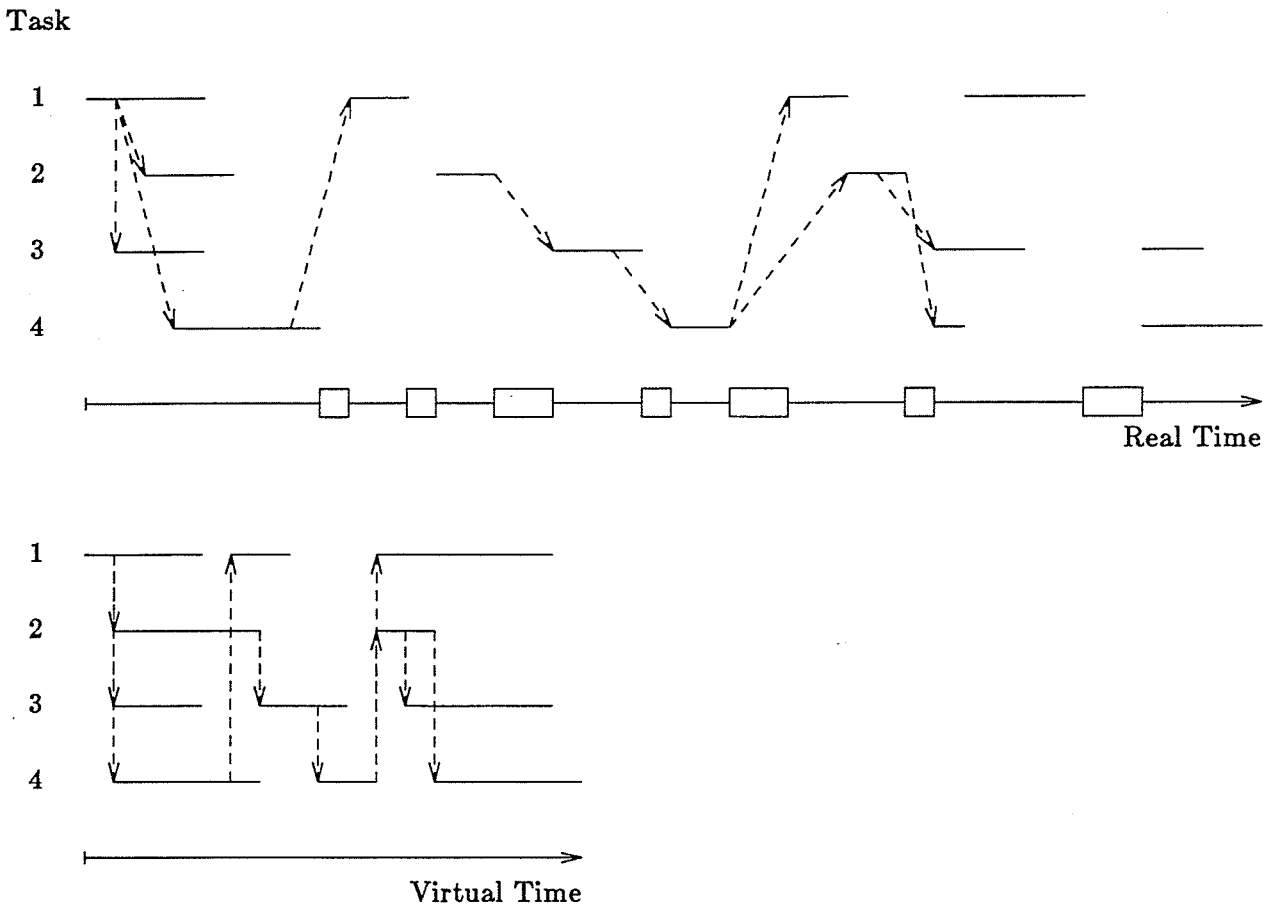

Figure 8. Virtual Execution Flow Analysis

5.3. Supercomputing Programming Environments

It is essential for CEDAR that performance analysis tools be made available to users through an integrated environment supporting the full range of program development facilities from optimizing and restructuring compilers to parallel debuggers. In keeping with the performance tools themselves, the supercomputer programming environment must be general and adaptive, providing the appropriate level of assistance for users of varying degrees of sophistication [27].

The *Faust* software environment being developed for CEDAR is shown in Figure 9. The fundamental goal of *Faust* is to make the CEDAR system easier to use. Although performance assessment is shown as only one component of the overall organization, in fact, performance analysis pervades the entire programming environment. Knowledge gained by the performance analysts today will be incorporated in tomorrow's compilers. Algorithm and subroutine libraries are the product of results from actual performance experiments. Dynamic and adaptive optimization requires the ability to analyze performance factors of the run–time environment. Pre–compilation and pre–execution performance analysis tools will be based on estimates of the run–time performance of an application derived from past execution statistics of the library kernels and the extent of their usage in the code. Effective integration of performance analysis tools into the *Faust* environment will result in a level of supercomputer usability heretofore unavailable.

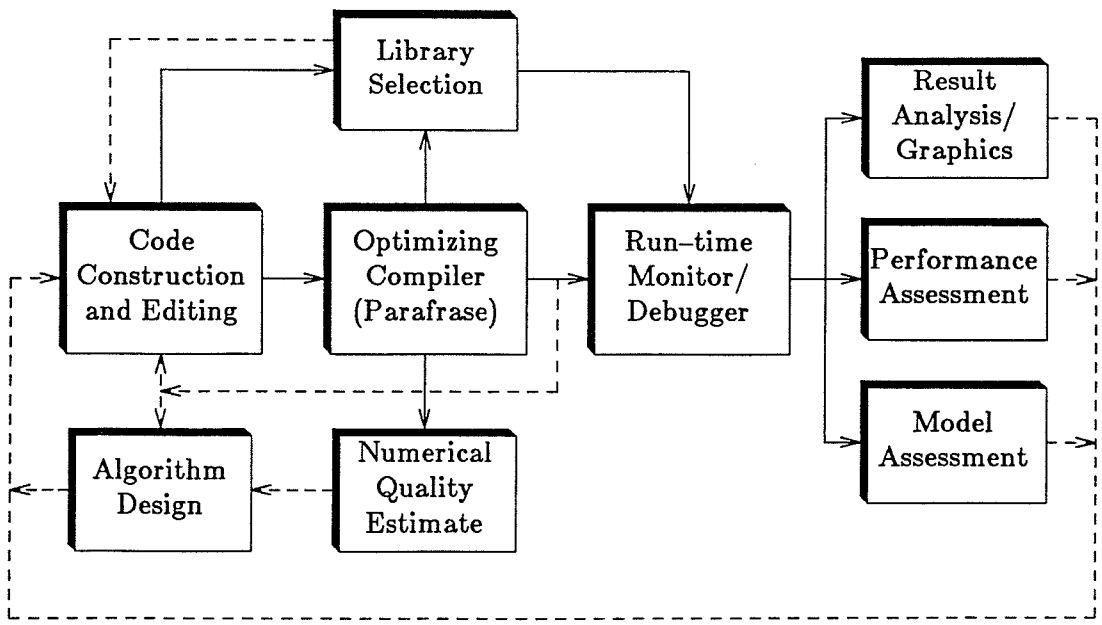

Figure 9. The *Faust* Supercomputer Software Environment

6. CONCLUSION

As supercomputer technology advances, it becomes increasingly clear that an integrated performance analysis system must exist as a central component of the overall supercomputer design. The performance analysis system should be hierarchical to support performance investigations at various levels of detail, adaptable to accomodate the different performance viewpoints of its users, and extensible to allow the inclusion of new analysis tools. Furthermore, the performance analysis system should be conceived as an eventual part of a general supercomputing programming environment able to apply performance knowledge and experience together with measured performance data in an expert system–based performance optimization tool.

The performance analysis system discussed in this paper is an attempt at achieving these goals in the context of the design and development of the CEDAR multiprocessor supercomputer. Many parts of the basic system design have been implemented. Interestingly, as the rudimentary performance measurement components of the system have become available, the importance of its presence has increased as has the list of desired additional performance analysis tools. At the same time, the need to assimilate the performance analysis system into the Faust programming environment has become more obvious.

ACKNOWLEDGEMENTS

The authors would like to thank Dan Lavery for his contribution to the discussion of the CEDAR hardware monitoring system.

REFERENCES

[1] D.J. Kuck, E.S. Davidson, D.H. Lawrie, and A.H. Sameh. *Parallel Supercomputing Today and the Cedar Approach.* **Science**, Vol. 231, Feb. 28, 1986, pp. 967–974.

[2] S. Fuller, R. Swan and W. Wulf. *The Instrumentation of C.mmp, A Multi-(Mini) Processor.* **IEEE Compcon**, 1973, pp. 173–176.

[3] Z. Segall, A. Singh, R. Snodgrass, A. Jones and D. Siewiorek. *An Integrated Instrumentation Environment for Multiprocessors.* **IEEE Trans. on Computers**, Vol. C–32, No. 1, January, 1983.

[4] H. Fromm, U. Hercksen, U. Herzog, K–H. John, R Klar, and W. Kleinoder. *Experiences with Performance Measurement and Modeling of a Processor Array.* **IEEE Trans. Computers**, Vol. C–32, No. 1, Jan. 1983, pp. 15–31.

[5] D.J. Kuck and A.H. Sameh. *A Supercomputing Performance Evaluation Plan.* **Proc. 1987 Supercomputing Conf.**, Greece, June, 1987.

[6] P. Emrath. *An Operating System for the Cedar Multiprocessor.* **IEEE Software**, Vol. 2, No. 4, 1985, pp. 30–37.

[7] R.E. McGrath and P. Emrath. *Using Memory in the Cedar System.* to appear in **1987 Int. Conf. on Supercomputing**, 1987.

[8] T. McDaniel. *Xas Reference Manual.* (unpublished), Univ. of Illiniois at Urbana–Champaign, 1986.

[9] M. Guzzi. *Cedar Fortran Reference Manual.* CSRD Report No. 601, Univ. of Illiniois at Urbana–Champaign, 1987.

[10] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. *Dependence Graphs and Compiler Optimizations.* **Proc. ACM Symp. on Principles of Programming Languages**, Jan. 1981, pp. 207–218.

[11] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* Ph.D. Thesis, DCS Rep. No. UIUCDCS–R–82–1105, University of Illinois at Urbana–Champaign, 1982.

[12] M. Berry, K. Gallivan, W. Harrod, W. Jalby, S. Lo, U. Meier, B. Phillipe and A. Sameh. *Parallel Numerical Algorithms on the CEDAR System.* CSRD Report No. 581, University of Illinois at Urbana–Champaign, 1986.

[13] K. Gallivan, W. Jalby, U. Meier and A. Sameh. *The Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design.* CSRD Report No. 625, University of Illinois at Urbana–Champaign, 1986.

[14] D. Kuck, E. Davidson, D. Lawrie and A. Sameh. *Parallel Supercomputing Today and the CEDAR Approach.* **Science**, Vol. 231, Feb. 28, 1986, pp. 967–974.

[15] D. Gannon and J. Van Rosendale. *On the Impact of Communication Complexity on the Design of Parallel Numerical Algorithms.* **IEEE Trans. on Computers**, Vol. C–33, No. 12, Dec. 1984, pp. 1180–1195.

[16] R. W. Hockney and C. R. Jesshope. **Parallel Computers**, Adam Hilger Ltd., Bristol, 1981.

[17] W. Abu–Sufah and A. Kwok. *Performance Prediction Tools for Cedar: A Multiprocessor Supercomputer.* **12th Int. Symp. on Comp. Arch.**, June 1985, pp. 406–413.

[18] A. Veidenbaum. *Compiler Optimizations and Architecture Design Issues for Multiprocessors.* CSRD Report No. 520, Univ. of Illiniois at Urbana–Champaign, May, 1985.

[19] R. Lee. *The Effectiveness of Caches and Data Prefetch Buffers in Large–Scale Shared Memory Multiprocessors.* CSRD Report No. 670, Univ. of Illiniois at Urbana–Champaign, May, 1987.

[20] C. Polychronopoulos. *On Program Restructuring, Scheduling and Communication for Parallel Processor Systems.* CSRD Report No. 595, Univ. of Illiniois at Urbana–Champaign, August, 1986.

[21] A. Miller. *Non–Preemptive Run–Time Scheduling Issues On a Multitasked, Multiprogrammed Multiprocessor with Dependencies, Bi–dimensional Tasks, Folding, and Dynamic Graphs.* CSRD Report No. 656, Univ. of Illiniois at Urbana–Champaign, May 1987.

[22] G. Brent. *Using Program Structure to Achieve Prefetching for Cache Memories.* CSRD Report No. 647, Univ. of Illiniois at Urbana–Champaign, Jan. 1987.

[23] D.J. Kuck, et.al.. *The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance.* **Proc. of 1984 Int. Conf. on Paralel Processing,** Aug. 21–24, pp. 129–138.

[24] A. Kwok and W. Abu–Sufah. *Tcedar: A Performance Evaluation Tool for Cedar.* CSRD Report No. 34, Univ. of Illiniois at Urbana–Champaign, March 9, 1984.

[25] A.D. Malony. *Cedar Performance Measurements.* CSRD Report No. 579, Univ. of Illinois at Urbana–Champaign, June 1986.

[26] A.D. Malony. *Program Tracing in Cedar.* CSRD Report No. 660, Univ. of Illinois at Urbana–Champaign, April 1987.

[27] D.A. Padua, V.A. Guarna, Jr., and D.H. Lawrie. *Supercomputer Programming Environments.* to appear in **Proc. Symp. on Parallel Computations and Their Impact on Mechanics**, to be held ASME Winter Annual meeting in Boston, Dec. 13–18, 1987.