Task-Oriented Architectures

ROBERTO BISIANI, MEMBER, IEEE, HORST MAUERSBERG, MEMBER, IEEE, AND RAJ REDDY, FELLOW, IEEE

Invited Paper

Abstract—**Recent advances in system design provide increasing opportunities for rapid experimentation with task-oriented architectures, i.e., architectures designed to perform extremely well for a given application. Problems encountered in one such experiment indicate that, contrary to initial expectations, only a small fraction of the effort is required for hardware development. Most of the effort was devoted to algorithm analysis and restructuring, architecture design, systems programming, software and hardware debugging, and performance analysis. This paper presents a case study in task-oriented architecture, using the Harpy connected speech recognition system as the problem domain. This study highlights the limitations of current methods and presents a number of tools and techniques that are useful for experimentation with task-oriented architectures.**

I. Introduction

THE FACTORY OF THE FUTURE is a concept of an essentially peopleless, paperless organization capable of producing a wide variety of parts on demand with minimal in-process inventory. The concept entails intelligent capabilities that exceed current computer systems. Such a system will require the capabilities for monitoring a large number (10^3-10^6) of sensors, distributed fail-soft operation, self-diagnosis and self-correction, dyanmic planning and scheduling, and self-simulation to answer "what-if" questions. Since each of these tasks is computation-intensive, together they can prove to be expensive and uneconomical. One way to reduce the excessive cost is to develop custom computer architectures that are engineered to perform specific task efficiently.

Speech recognition, speech synthesis, and image processing tasks are examples of sensor-interpretation problems that require the development of special-purpose processors. The typical Factory of the Future will have a large number of image and sound sensors requiring analysis and interpretation. Today's connected speech recognition systems use 10^7 to 10^9 operations to interpret a second of speech, depending on the vocabulary size, complexity of the language, and the nature of the algorithm. Image understanding systems with substantially higher data rates require 10^7 to 10^{11} operations to process and interpret images. The smaller numbers are typical of simple robotics applications, while a number of operations in the order of 10^{11} is needed for real-time analysis of multispectral satellite imagery.

As the possibility for rapid experimentation with new design increases, solutions based on improving the relation between algorithms and architectures (we call these solutions task-oriented architectures), will become attractive alternatives for sensor-intensive problems in robotics and for computation-intensive tasks in speech and image processing. However, task-oriented architectures raise a number of issues: How much computing power is required? How much memory is required? What is the optimal architecture and system organization? Is it fail-soft? What type of software is available? How is it programmed? How is it debugged? How is it evaluated? This paper presents a case study in task-oriented architecture (Harpy machine) using the Harpy connected speech recognition system as the problem domain and discusses tools, techniques, and methodologies that are useful for the development of custom computer architectures.

Any approach to task-oriented architectures must consider a number of external factors that might affect the overall system, such as cost, time, and architectural innovation. The following may be considered as guiding considerations for the development of effective task-oriented architectures.

Throwaway hardware: It is common practice to view hardware design as a one-time activity resulting in a semi-permanent product. In contrast, it is not uncommon in software design to discard a program which has taken several man-months of effort and start over again. Custom design is more like software design rather than hardware design. Therefore, rapid prototyping with a willingness to discard the resulting hardware is an important attribute for realization of a successful custom design.

Time and cost per iteration: If a proposed custom architecture project requires hundreds of man-years of effort as in the case of general-purpose systems, then the resulting product is likely to be uneconomical unless the volumes involved are large. For custom architectures, it is essential to plan, develop, and adapt existing tools so that the entire project takes no more than a few man-years of effort. Developing the tools may require much greater effort—especially tools for producing software such as operating systems, run-time environments, and diagnostic aids—and should be viewed as an independent activity.

Not a solution in search of a problem: Usually, in general-purpose parallel systems research one attempts to map a problem solution onto some predefined parallel architecture. For tailored architectures, it is essential to start with the problem and then attempt to discover what type of architecture is most appropriate.

Manuscript received January 15, 1983; revised March 19, 1983. This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD) and monitored by the Air Force Avionics Laboratory under Contract F336 15-78-C-1151.

R. Bisiani is with the Computer Science Department and The Robotics Institute, Carnegie-Mellon University, Pittsburgh, PA 15213.

H. Mauersberg was with the Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213. He is now with Siemens Corporation, Corporate Research and Technology, Princeton, NJ 08540.

D. R. Reddy is with The Robotics Institute and the Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213.

0018-9219/83/0700-0885 $01.00 © 1983 IEEE

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.

In the Harpy machine, all of these considerations affected the final solution, and the resulting architecture had many novel features not usually present in conventional systems.

II. Algorithm Analysis and Restructuring

The first step in the design of a task-oriented system is to evaluate the critical parameters of the algorithms that the system has to execute: *What is the total computational effort? How much memory is required?* Although the answers to such questions require a detailed evaluation of an algorithm, what is needed is not always the traditional analysis of algorithms "a la Knuth" [11], [9] or the usual measurement of the execution time of different parts of the algorithm [6], [7]. The former does not easily take into account the characteristics of the computer architecture that executes the algorithm. The latter generates information that is dependent on the architecture and implementation of the machine on which the algorithm is run. Since the process of task-oriented design implies changing and reorganizing the algorithm, a simple analysis and measurement of the original algorithm does not yield the desired attributes. What is needed is an invariant (implementation- and architecture-independent) characterization of the basic computation that is detailed enough to guide the evaluation of alternative architectures and implementations. From now on we will use the term "architecture" in a broad sense to indicate both the architecture and the implementation.

The problem with most evaluation techniques is that they do not clearly specify the interactions between an algorithm and its implementation on a given architecture. This happens either because the evaluation is too abstract, e.g., pencil and paper analysis, or because it relies too heavily on a given implementation. For example, a programmer who has implemented an algorithm on a given machine with a given language might have a distorted idea of how the algorithm would behave on another machine with another language: The programmer might not be aware of the relationship between algorithm, compiler and, say, number of registers available in the central processing unit. Knowing how many registers are needed to optimally execute a given algorithm might be exactly what is required, and a good analysis technique should make this information evident.

The designer of task-oriented architectures is not interested in general information (e.g., how many millions of floating-point operations per second can be executed), but in the interaction between a specific algorithm and a specific architecture. In this context, algorithm analysis can be described as a *guided search* of an *algorithm-behavior space*, by which we mean the set of all possible performance data points regarding the behavior of a given algorthm when executed by a family of architectures. Viewing the analysis process as a search in the behavior space helps to limit the number of experiments, because the information collected during analysis directs the designer to expand the search in certain directions rather than others.

The performance of a given algorithm must be considered in the context of the model used to compute it [16], which means that an algorithm that is optimal with a given model may not be optimal with a different model. When designing a task-oriented architecture, one can adopt the model-algorithm pair that best suits the problem and then build an architecture that closely resembles that model. Therefore, there is a greater degree of freedom than in conventional system design because both the algorithm and the architecture can be varied. Properly matching these two components is a difficult process in which the intuition of the designer becomes the only guide unless some proper methodology has been established. The search of the algorithm-behavior space by rewriting the algorithm using appropriate tools is one correct way to guide the designer in devising a good algorithm–architecture compatibility.

In this section we present a two-step analysis and restructuring methodology we have found to be useful. First, information is extracted from one of the available implementations of the algorithm and is used as input to an iterative process that explores the architectural and algorithmic alternatives. Second, the algorithm is restructured to evaluate its performance under different architectural solutions. We will demonstrate how this methodology can be applied by using the Harpy system as a case study.

In the domain of speech recognition, signal processing is often a significant part of the computation. However, as the size of the vocabulary and the complexity of the language increases, signal processing problems diminish and the key computational issue shifts to comparing the given utterance to the extremely large number of utterances permitted by the language. For example, the 1000-word vocabulary and highly restrictive syntax of the Harpy system [13] permitted over 10 billion alternative utterances. Thus the problem for connected speech recognition is not signal processing but intelligent search, i.e., how to eliminate unpromising utterances from consideration so that only a few alternatives need to be examined in detail.

The Harpy system relies on a graph search to establish an optimal path through the candidate data. This procedure is usually referred to as *beam* search. The graph embodies the syntactic and vocabulary constraints of the language. Given an unknown utterance, Harpy segments the signal and assigns probabilistic labels. The sequence of labeled segments is compared against each of the alternative paths in the graph that represent acceptable phoneme sequences in the target language. At each stage of the comparative process, the Harpy algorithm examines each promising (i.e., remaining) candidate, accesses potential successor information from the syntactic and vocabulary constraints, compares each successor to the unknown signal, rank orders the resulting candidates, eliminates the least promising, and repeats the process until a match is achieved. The Harpy algorithm is discussed in detail by Lowerre and Reddy [13]. We shall present a fragment of this algorithm later in this section.

A. The Algorithm Evaluation: Data Collection

The first step of the analysis procedure is to collect information regarding the dynamic behavior of the algorithm. This step needs to be performed only once. The method of *frequency counts*, as suggested by Knuth in [12], is a convenient way to extract information concerning the behavior of a program. This method entails *instrumenting* the program by inserting *counters* at appropriate program locations to determine the number of times each statement is executed. For example, *checkpoints* or *software probes* can be inserted at all conditional branches in a program. The effect of checkpoints is to increment a counter by one every time the program takes a certain path. Software measurement tools like SMF [15], SIPE [4], UNIX profile, Informer [6], or SMT[8] can instrument a program with little user effort.[1] We did not have a software measurement tool as sophisticated as Informer,

[1] See [7, example 2.5, p. 54].

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.

```
struct{ char next;              /* pointer into "next nodes list" */
        char maxdur;            /* max duration */
        char mindur;            /* min duration */
        char phnof;             /* associated phoneme */
        char transcnt;          /* transition count to the final node */
} graphnode[16620];

if (segment > 1)                /* for every segment but the first */
{   duration = graphnode[index].mindur - fromlength;
                                /* minduration is stored in the item mindur
                                   of the element identified by index */
    if (duration > 0)           /* we have already been in this node */
        nodeprob = nodeprob + tprob * duration; /* compute likelihood */
}
```

(a)

```
if (segment > 1)
{       block6 = block6 + 1;
        duration = graphnode[index].mindur - fromlength;
        if (duration > 0)
        {       block6= block6 + 1;
                nodeprob = nodeprob + (tprob * duration);
        }
}
```

(b)

```
if (segment > 1)
{ <block6>
    duration = netfromnode - fromlength;
    if (duration > 0)
    { <block6>
        nodeprob = nodeprob + tprob * duration;
    } <block6>
} <block6>
```

(c)

```
if (segment > 1)
{ <block6>
    duration =  contentof(graphnode #+ (index #* graphnodesize) #+ cmindur)
                                                        - fromlength;
    if (duration > 0)
    { <block6>
        nodeprob = nodeprob + tprob * duration;
    } <block6>
} <block6>
```

(d)

Fig. 1. Examples of program rewriting. (a) Original program. (b) Instrumented program to generate frequency counts. (c) Modified nonexecutable program, the implementation of graph access is hidden by rewriting graphnode [index].mindur as netfromnode. (d) Modified nonexecutable program, the operation of calculating the address of data in the graph when compared with other operations is emphasized; graphnode[index].mindur is rewritten using new operators (e.g., #+ that has the meaning of sum of addresses).

so it took a little longer to modify a UNIX utility program[2] into a system that would insert statements, such as *blocknumber = bocknumber + 1*, at every branch of the program. Different runs of the instrumented program show if the program behaves differently with different input data and specify the parts that are more influenced. The frequency counts so obtained are independent of the architecture and implementation. However, frequency counts are not independent of the algorithm used and can be used to evaluate architectural alternatives for a given algorithm.

Fig. 1(a) shows the implementation of a small part of the Harpy algorithm written in C language. This fragment belongs to the search inner loop and computes the likelihood that the correct path goes through a given node of the Harpy network. Each node in the network is represented by one element of the structure called *graphnode*. Each element contains five items that define the characteristic of the node (the data structures that connect the nodes to form a network are not shown). For example, following the C language syntax, *graphnode[index].mindur* represents the minimum expected duration of the speech signal represented by graphnode *index*.

Counters are added at the beginning of each block, and each block is assigned a label. The two blocks in the example are called *block5* and *block6*. The modification of the original program is shown in Fig. 1(b). The execution of the instrumented program generates a set of numbers that indicate how many times a block has been executed (frequency counts). Frequency counts are computed for different cases to check their dependency on input data.

B. The Algorithm Evaluation: Algorithm Rewriting

The task of the designer is to investigate the changes that could be made to the algorithm to improve its performance on some new architecture. This task requires a good understanding of the drawbacks of the existing implementation and the capability to evaluate the impact that different reorganizations of control and data would have on performance. For example, one might want to evaluate the impact of implementing cer-

[2] This utility program, called *Indent*, neatly formats C language programs according to the language's syntax.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.

tain operations on a given data structure by means of specialized hardware. This means that certain operations would have to be taken out of the code (because the operation has migrated into the specialized hardware), others would have to be modified (because the data structure is accessed in a different way), and others would have to be added (because the specialized hardware and the processor must exchange data). This modification entails much more than weighing the data collected from the execution of the program, since the effect of the modification cannot be estimated independently of all the rest of the code.

The technique that we used lets the designer automatically evaluate the impact of algorithm implementation. Initially, all the code that does not strictly represent the behavior of the algorithm, such as debugging code and declarations, is deleted. The goal is to obtain a description that is as much as possible representative of the algorithm's behavior, although still language and implementation dependent. Such a description contains operators and data structures that are typical of the language that is being used and an organization that is convenient for the machine for which the algorithm was implemented. This code can be analyzed as it is to extract the information relative to the current implementation, or modified to evaluate alternative implementations.

The analysis consists of, first, inspecting the program to count, for each block, the number of times operations are performed and data accessed and, second, multiplying this information by the frequency counts. The analysis of the Harpy algorithm was performed automatically by a program (Handy) that computes the number of times each operation is executed and each variable is accessed and reports these values in tabular form. Handy is a simple program and does little more than relieve the designer of an error-prone process of normal data tabulation.

Fig. 1(c) and (d) are two examples of how the same program fragment can be rewritten. The modifications are underlined. The program in Fig. 1(c) is used to distinguish between memory accesses to the graph (rewritten as *netfromnode*...) and memory accesses to other variables. Knowing the number of times the variable *netfromnode* is accessed is useful for evaluating the impact of implementing the graph access by means of specialized hardware rather than by software.

The modified program in Fig. 1(d) is an attempt to understand the effect of the operation of calculating the address of data in the graph when compared with other operations. To this purpose, the operations required to get an item of the structure *graphnode* are explicitly indicated by using the newly defined operators: #+ means sum of addresses, #* means product of addresses, and *contentof*, whose meaning is obvious. The constants *graphnodesize* and *cmindur* are, respectively, the size of an element and the offset of the item *mindur* in an element.

New arbitrarily complex operators can be envisioned or the implementation of a data type can be completely changed. It is the designer's responsibility to emphasize the interesting operations and to hide what is not relevant. Thus the designer can get a coarse evaluation of the impact that certain architectural features will have on the performance.

C. The Evaluation of Alternatives: Algorithm Restructuring

The process of algorithm restructuring was used for the Harpy system and Table I shows some of the results of the evaluation of the search algorithm. From Table I we can derive that:

TABLE I
SOME RESULTS OF THE EVALUATION OF THE HARPY SEARCH

number of arithmetic operations:	$.68 * 10^6$ (5% of total)
number of comparisons:	$1.3 * 10^6$ (10% of total)
number of accesses to data other than network:	$4.0 * 10^6$ (43%)
number of accesses to the network data structure:	$5.2 * 10^6$ (57%)
divided in accesses for address calculation:	$3.8 * 10^6$ (73% of 57%)
and accesses for information retrieval:	$1.4 * 10^6$ (27% of 57%)

Arithmetic operations, especially multiplications, constitute a small part of all operations (5 percent). No special arithmetic capability appears to be essential.

The number of comparisons (e.g., if statements) is small compared to the number of other operations (10 percent).

Most of the time is spent accessing large and complex data structures. 57 percent of the memory accesses store or retrieve information in the data structures that describe the network, i.e., *5.2/(5.2 + 4.0)*. Of all memory accesses to the network, 73 percent are address calculations, i.e., *3.8/5.2*.

From these observations we can derive the following.

The time spent to pack and unpack the information contained in the network data structure is significant. It is necessary to either avoid packing or to speed up the operation by means of some hardware implementation.

Some of the data structures created during the search are managed using a hashing algorithm that saves space but requires considerable computation. These data structures can be changed to use a simpler access algorithm requiring less computation.

Both these characteristics resulted mainly because the first few implementations were constrained by the limited addressing capability of the machines used. In rearranging the network access algorithm, we observed that the same access pattern to the network could be obtained with a simpler algorithm. Although the algorithm had been investigated in depth earlier, this characteristic had gone unnoticed. The rearranged algorithm was retrofitted to the original system and improved its performance considerably. This is an additional benefit of exploring task-oriented architectures for a given application: Even if one decides not to build a task-oriented architecture, the original system itself may emerge considerably improved because of a deeper understanding of the computational complexity of the algorithm.

D. Summary

The two-step process for the iterative examination of the algorithm-behavior space can be used in the evaluation of different algorithm structures. There are four characteristics that make this approach appropriate for task-oriented architectures:

1) *The granularity and the kind of information* contained in the results *can be strictly controlled* by the designer. It is possible to gather exactly the information required and to implement increasingly refined experiments until all the interesting aspects of the algorithm-behavior space have been investigated.

2) Unlike conventional analysis of the structure of the algorithm, this procedure *takes into account the dynamic behavior*.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.

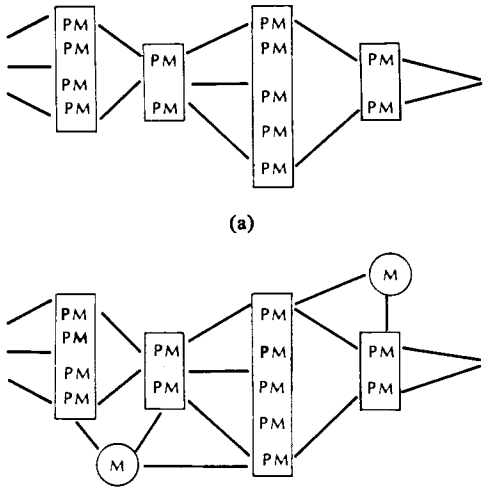

Fig. 2. Architectural alternatives for the Harpy system. (a) Parallel pipeline. (b) Parallel pipeline with global memories.

3) This technique is *quick to use*, because program rewriting is a mechanical job, once the designer has decided which modifications are necessary. The effort required to implement the analysis program is small, and the computation time is negligible. If more time were necessary, the designer would be discouraged from using the analysis tools and tempted to use intuition instead.

4) As a side effect, rewriting the algorithm for implementing an experiment *forces the designer to examine each part of the algorithm* and to understand it fully. In this case, the algorithm evaluation process led to the improvement of the original system.

III. Architecture Design

In this section we consider the problem of algorithm-to-architecture mapping. Given the results of algorithm analysis, there are any number of ways of designing the architecture. The global parameters are known to the first order of approximation: number of operations per second, amount of data to be stored, and approximate program size. If the task can be performed on a single general-purpose microprocessor, then there is no need for a tailored architecture. If it can be implemented on a single custom processor, then there is no need to consider parallel architectures. In our case, both a custom processor and a parallel architecture were required.

A. The Task

The analysis of the previous implementations of Harpy shows that the system can be modeled by a pipeline with a strong imbalance of computation requirements at each stage of the pipe and the need to share a large amount of data between stages. The analysis of separate stages of the pipe shows that the amount of computation required varies widely from stage to stage and that the computation performed at various stages is of a different kind. For example, the signal processing part involves mostly arithmetic operations and indexing, while the search part relies on complex data access operations not readily available in conventional architectures. Pipe stages can be modeled by clusters of heterogeneous processor–memory (PM) pairs operating in parallel (Fig. 2(a)).

The nature of some of the subtasks implemented by each stage of the pipe calls for an efficient sharing of data within clusters, while the overall structure calls for data sharing between nonadjacent stages of the pipe. Therefore, a parallel-pipeline structure with global memories (Fig. 2(b)) was used. This is still a rather general scheme that encompasses many different structures. For example, the data within a pipe stage or between stages could be shared in many different ways. The decision about how to implement such a sharing has to be based on the evaluation of each stage of the pipe, and can be different for each stage. In this section, we will present a design and implementation of one of the pipe stages.

B. The Design of the Search Stage

The following items summarize what could be extracted from the evaluation of the algorithm-behavior space for the search stage of the pipe:

- The behavior of the algorithm is dependent on input data. For example, because of the particular search technique implemented (beam search), the number of transitions that are generated by expanding a given state can vary widely from state to state and from input utterance to input utterance. Moreover, a large amount of data has to be accessed in a way that is not predictable in advance.
- The search process can be performed in parallel on many nodes since the evaluation of a node is only dependent on the result of the evaluation of nodes directly proceeding it.
- Some complex data access operations must be indivisible. The algorithm requires operations that temporarily invalidate the correctness of the data structure operated upon. Thus if more than one processor is used, the architecture must provide an efficient mechanism to enforce the correct sequence of accesses.
- The data structures are rather complicated, and their associated access functions are time consuming.
- If more than one processor is used, the synchronization overhead and delay become crucial and the load must be shared dynamically to achieve task and input independence.

Since the search task exhibits some "natural" parallelism, we investigated the ways in which the algorithm could be partitioned among multiple processors. The following factors were found to be of primary importance:

synchronization overhead should be kept to a minimum;
synchronization delays should be avoided as much as possible;
the load should be *dynamically partitioned* between processors;
the solution should be *independent of the language and vocabulary* used in the recognition.

The importance of minimizing the synchronization overhead, i.e., the time the system components spend to synchronize a given operation, was made evident by the performance evaluation of two previous implementations of the search algorithm on two multiprocessors. C.mmp and Cm* [14], [17], [10]. In both cases, the network had to be partitioned among different memory modules because of architectural constraints. For C.mmp, the synchronization overhead necessary to reshuffle the processor–memory interconnections to accommodate algorithm fluctuations was one factor that limited performance. For Cm*, the time penalty in accessing nonlocal sections of the network was the bottleneck. In fact, the implementation of Harpy had the worst ratio between local

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.

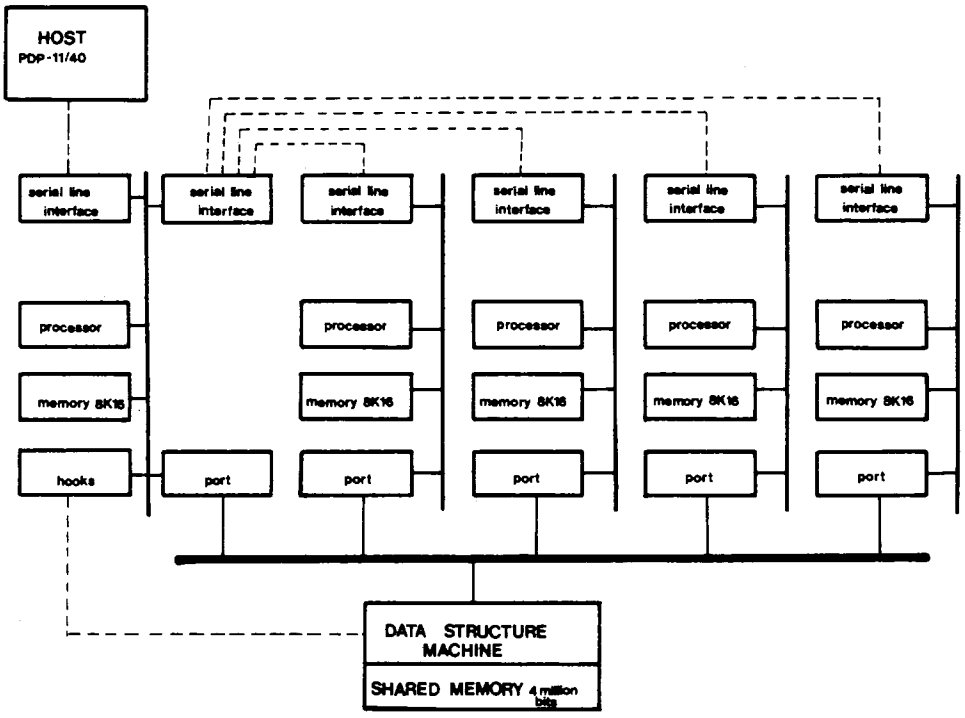

Fig. 3. Architecture of the Harpy machine.

and nonlocal memory references among all the benchmark algorithms tested during the initial evaluation of Cm* [17], [10]. The algorithm-behavior analysis confirmed the C.mmp and Cm* measurements. It was, therefore, necessary to design the architecture such that the synchronization overhead would be minimized.

Moreover, a simple simulation experiment and data by Oleinick [14] showed that even with a negligible synchronization overhead the utilization of the processors would be much less than 100 percent unless the synchronization delay, i.e., the time spent waiting for other processors to terminate a given step, was minimized. Minimizing synchronization delays often results in increasing the synchronization overhead. The simplest way to minimize synchronization delay is to reduce the granularity, i.e., to divide the algorithm into many small steps, which usually implies greater synchronization overhead. In the Harpy machine, unlike most general-purpose multiprocessors, the synchronization overhead can be kept as low as the time it takes to access one memory location. Thus the synchronization does not offset the benefit of small granularity.

The third requirement, i.e., that *the load be dynamically partitioned*, resulted from the input-dependent behavior of the search algorithm. Input-dependent behavior means one cannot predict from a single run of the program how the algorithm will behave in another run. For example, in the case of beam search in which nodes are expanded and pruned, it is not possible to statically assign a set of nodes to a processor and let it handle all the successors. The pruning procedure might later decide to discard all the nodes assigned to a given processor, leaving it idle while the other processors have to deal with an increasing number of nodes. One solution is to divide the algorithm into small steps and let the first available processor pick up the first available task in the "producer-consumer" queues. This solution is particularly suitable for beam search, but it might not be as effective when applied to other problems. In particular, this approach will not work if the task requires substantial transfer of data between atomic computations. Beam search behaves well with this kind of restructuring, because each path can be expanded in parallel with minimum information sharing.

Finally, the desire for the solution to be independent of the size of the task (e.g., number of words, complexity of the grammar), fail-soft, and independent of the number of processors available at any given time required a partition of the algorithm to take advantage of all available processors. This led to a structure in which the producer–consumer queues are in global memory so that a processor can perform a new atomic computation as soon as it completes the previous one. This makes the partition task independent and eliminates the need to prepartition the task. However, such a partition can potentially introduce a new bottleneck, memory contention, and it is viable only if the architecture makes it very inexpensive for processors to access global memory.

A possible solution that satisfies these needs is shown in Fig. 3: a set of microprocessors (five in the implementation) shares a fast, intelligent global memory. Processor requests are directed to the memory processor, called Data Structure Machine (DSM), that translates them into real addresses in the shared memory. The central position of the DSM in the cluster allows it to perform other critical tasks such as synchronization and load sharing. The DSM is not only necessary to achieve an efficient parallel implementation of the search, it also substantially improves the performance of each search processor. As shown in Table I, 57 percent of the total memory accesses deal with the network data structures. The DSM can be designed to perform these accesses in a fraction of the time required on a general purpose processor. An implementation of the DSM is described in Subsection IV-B.

C. Summary

A systematic approach to algorithm-to-architecture mapping is essential for a successful implementation of a task-oriented

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.

architecture. Since a general-purpose system is designed to run a wide variety of programs, a wrong decision in selecting or omitting an architectural feature does not drastically impair the performance of the system. This is not true in task-oriented systems. In the case of Harpy machine, the computational requirements led to a parallel-pipeline architecture with intelligent global memories.

IV. Hardware Development

Unlike the development of general-purpose systems hardware, task-oriented computers provide a unique advantage: The application software is already implemented, debugged, and running on some general-purpose system, albeit slowly and possibly at great cost. If one can modularize and fit this software to the structure of the target system, then the hardware/software development process can be greatly simplified.

A. The Strategy of Interchangeable Hardware–Software Modules

The Harpy system, as it has been implemented on a general-purpose computer, is a complete system that accepts analog data from a microphone and generates an answer on a display. It is crucial that the whole system be implemented, rather than only parts of it; this is true for both the initial concept demonstration and for subsequent implementations. On the other hand, there might be parts of the system that do not warrant the effort of building them in hardware in a prototype because the solution is straightforward and the performance predictable. In the Harpy system, this was the case with the signal processing part that required fast arithmetic that has already been extensively investigated.

Before proceeding with the hardware implementation, the behavior of the hardware components was described at a high level and the interfaces programmed in detail so that the data flowing between modules and their format remained the same whether the module was executed in software or in hardware. Since the basic algorithms were already correct, only a short time was needed to debug and test the modular decomposition. The system remained operational through the development stage: As they became ready, hardware modules were substituted for software modules. This was in contrast to conventional approaches used for small systems that attempt to debug most of the components at the same time. Although the system was substantially slower when some of the hardware modules were emulated in software, its degraded behavior did not affect the debugging process.

This approach was also useful to ease the microprogramming of the DSM: The behavior of the machine was first described and *tested* as a software process written in C language, then translated into microcode, effectively reducing the required effort.

Such strategies are important for the success of task-oriented computer architectures. When both new hardware and new software are tested at the same time, substantial effort is wasted to locate an error. The hardware/software migration strategy allows an almost complete debugging of the software on general-purpose hardware, and then the debugging of the hardware in the final environment with continuously working software.

B. The Hardware Development of the Harpy Machine

Fig. 3 shows the architecture of the Harpy machine system—the dashed lines represent connections and modules used only for debugging or maintenance. Algorithm analysis—of the kind described in Section II—indicated that five processors can run the current Harpy tasks in real time. The Digital Equipment Corporation LSI-11 was chosen as the search processor because of earlier experience and the availability of software. Each LSI-11 processor has a local memory of 8K 16-bit words and a custom module called port that interconnects it to the DSM. The port checks the addresses generated by the processor against two programmable address limits. If an address is located in the "window" identified by the two limits, the port stores the address (and the data in case of a write operation) generated by the processor, freezes the processor in the middle of the cycle, and notifies the DSM. The DSM accesses the port, retrieves the address, performs the operation associated with that address, and releases the processor after returning a value if the processor made a read operation. (Port requests are served by the DSM with a priority/round robin policy, which prevents starvation.) The DSM should not be confused with conventional memory mapping hardware. Although the DSM operations are triggered by reading or writing at a predefined location in the processor address space, a DSM operation can be arbitrarily complicated and involve more than one memory cell or, possibly, none.

Fig. 4 shows the data paths of the DSM. The DSM was implemented using bit slice bipolar technology (AMD 2903/2910) along with standard LSI-11 memory cards. It is possible in one microcycle (150/200 ns) to read a word from the input port, operate on it, and write it into the shared memory input register. In addition to the 16 internal registers of the AMD 2903, the DSM has 1K by 16 bit of high-speed register memory, which is used to store some of the information necessary for addressing (e.g., pointers, counters). The DSM is controlled by a long microword (80 bits) that allows the use of almost all the data paths in parallel.

Debugging and maintenance are performed by using the communication paths indicated by the dotted lines in Fig. 3. Serial lines connect one of the processors to a general-purpose host processor (PDP-11/40) and to the other four processors. The rationale for these interconnections will be explained in Section V. In addition, a special interface (hooks) connects one of the processors with the DSM and allows the reading and writing of most of the registers, memory, and control memory. Moreover, the "hooks" interface controls starting, stopping, single step, and many other debugging modes. The "hooks" implementation was patterned on a similar device designed for Cm* [17].

C. Implementation of the Search on the Architecture

Each LSI-11 processor executes the same algorithm on data that the DSM sends to it and returns the results to the DSM. The following is a simplified description of the algorithm executed by each processor:

1) retrieve a node pair and all the necessary information from the Data Structure Machine;
2) compute the score;
3) compare it with the previous best score;
4) if the new score is better, update the "best score" variable;
5) compute the "threshold";
6) if the score is not within the threshold then go to 1), else return the pair and the associated score to the Data Structure Machine;
7) go to 1).

The DSM assigns the first available pair of nodes to the first processor that requests it. In this way all processors are kept

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.

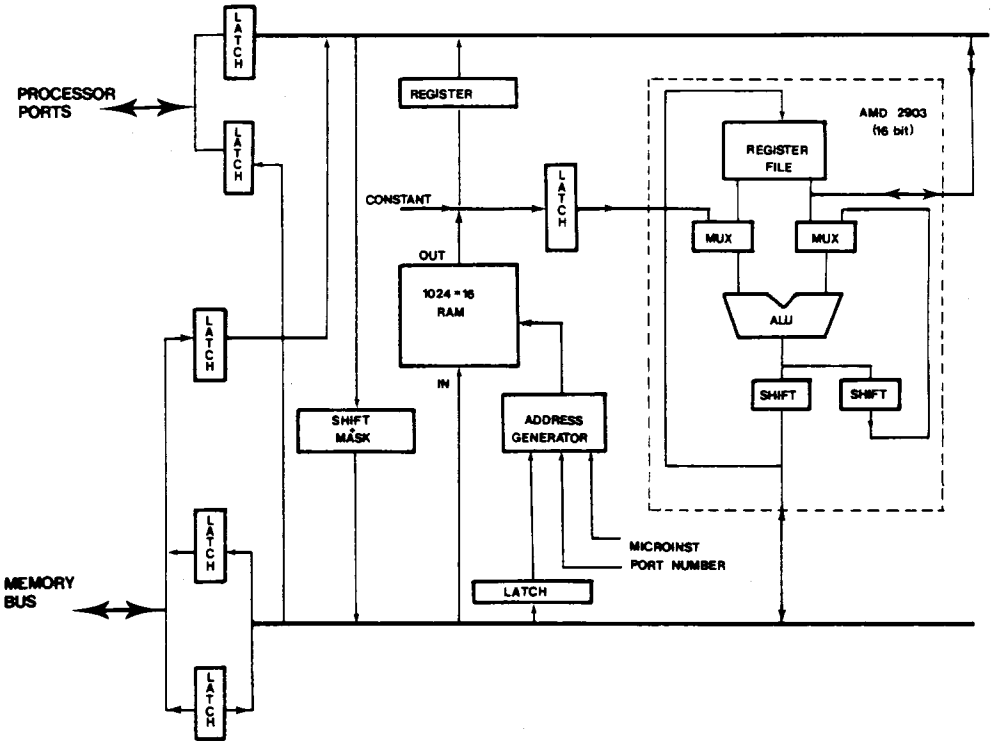

Fig. 4. The data paths of the DSM.

Function Description:

Find the next pair of nodes for which a score has to be computed. Associate it with the processor that requested the execution of the function and return to the processor the name of the parent node of the pair.

Access Algorithm:

```
Get the first un-marked transition out of the "current"
parent node.

Mark it and assign it to the "current" parent node.

Return the name of the "current" parent node to the
requesting processor.

If this is the last un-marked transition out of the
"current" parent node then:

        Get the next parent node from the table containing
        the nodes to be expanded.

        Make the new parent node "current".
```

Data: (in the shared memory)

- network;
- table containing the nodes to be expanded;

(in the local 1K by 16 bit memory)

- pointer to the current parent node;
- pointer to the first un-marked child node;
- pointer to the child node associated with the requesting processor.

Timing:

- 24 data structure machine instructions if there is already a transition out of the current parent node (3.5 μsec);
- 20 to 40 data structure machine instructions if a new parent node has to be retrieved (3 μsec to 7 μsec).

Fig. 5. Profile of a DSM operation.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.

busy. An interesting characteristic of this algorithm partition is that it can be executed, without any change in the software, on a variable number of processors; and it is independent of the size of the network, which is useful for both task independence and performance evaluation.

The implementation of the Harpy search requires 18 DSM operations that use about 500 words of nonoptimized microcode. Fig. 5 shows the implementation of one of the more complex of such operations, i.e., the node expansion operation. This is the first operation to be executed by a processor each time it starts processing a node pair. Retrieving a node pair and its associated information is a logically indivisible operation that requires many physical accesses to global memory. Since the DSM creates the necessary information in its local memory to assign the pair to the processor, an indivisible operation such as this can be performed by a processor as a sequence of DSM accesses without constraining the other processors' accesses in any way. In other words, the DSM can manage any interleaving of requests from different processors without having to "lock" the network. Fig. 5 also describes the data used in this operation and where they are allocated (the large shared memory or the internal 1K by 16-bit DSM memory). Timing is in microcycles, where each microcycle takes 150 or 200 ns, depending on the kind of instruction executed.

D. Summary

The strategy of gradual migration of components from a simulator on the host to the target system is useful for hardware debugging in that it allows one component to be introduced into the prototype at a time. In the initial architecture definition, often one tends to omit many of the hardware features needed for control and debugging. Without careful attention to these details, subsequent debugging can become more difficult than necessary. The DSM is novel in that a microprogrammable engine is used to implement an intelligent memory incorporating complex accessing and synchronization functions.

V. Software Development

The availability of proper hardware building blocks plays an important role in the realization of task-oriented computer architectures. Apparently, the concept of hardware modularization has no equivalence in the software area. We cannot easily assembly operating systems for multiprocessor systems out of commonly available software modules, nor customize the behavior of a specific software function by collecting off-the-shelf software modules. This section describes the problems of software development for task-oriented architectures, gives an overview of the approach taken for the Harpy machine project, and outlines some possible improvements for the future.

A. Programming Environment

The role of a programming environment is to support a full range of software development tasks, including language support, operating system support, device control, debugging, and application program development. The design and implementation of all these tools can often take hundreds of man-years of effort for a general-purpose system. In the case of task-oriented architectures, it is often impossible to devote such an effort to the development of tools. What is needed is an approach that will make it unnecessary to develop all these tools from scratch.

The issue of a proper programming environment has been considered only in the context of relatively large projects. Small projects are usually based on the premise that a few simple tools (e.g., a cross assembler) give complete control and adequate support in the development of a system. Unfortunately, a programming environment is required not as a function of the size of a project but of its complexity.

A suitable programming environment is essential for the development of task-oriented architectures. One reason is the tight hardware/software dependency. When implementing an application package on a general-purpose machine, the problems are entirely software related. On a task-oriented architecture, the situation is more complex. The goal of tailoring the hardware and software structure to perform a specific task leads to strong interdependencies between hardware structure and software design. Any improvement through algorithm analysis is directly reflected in the hardware structure and, conversely, hardware modifications, such as a different interconnection structure, may change the decomposition of the application software.

Another cause of complexity is the requirement for low-level system software. Since the software has to run on bare hardware, a tailored operating system has to be built to provide run-time support. This operating system need only be rudimentary (i.e., a kernel operating system) because of the specialized application. However, the design of a kernel operating system is not straightforward. Tailoring the architecture to a specific task means that the resulting architecture will be rather unique, and it will be hard to find existing operating systems that can be adapted easily to suit the requirements of the system.

The multiprocessor environment imposes a further degree of complexity on the software development process. The application software has to be distributed, and the operating sytem kernel has to provide for interprocessor communication. An additional problem arises when debugging the system. A heterogeneous system with a number of specialized functional units requires a diverse set of nonconventional debugging functions.

The components of a programming environment include:

Language support. Different languages have to be supported. Most of the software is written in a high-level language. Also, special language features are needed for the specialized hardware. In the case of the Harpy machine, the language support included C language and assembly code for the LSI-11 microprocessor and a customized microassembler for the data structure machine. In addition, facilities for separate compilation and linking/loading are needed.

Operating system kernel for task-oriented architectures must include protocols for inter-processor communications, device drivers and interrupt handlers, and a run-time system for accessing peripherals and for handling run-time exceptions.

Control. The run-time environment and the operating system must include features for loading and starting single components (e.g., processors/memories) or the whole system. In addition, one has to be able to start and stop selected parts of the system during debugging.

Debugging. It is hard to imagine a more difficult situation for debugging than a dedicated and tailored multiprocessor architecture. Such a system would require simultaneous debugging of hardware and software on different machines using different languages.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.

Development of the application programs. It should be possible to write and test the application program separately from all or part of the nearly developed hardware.

B. A Programming Environment for Task-Oriented Systems

Time-sharing systems usually offer a great variety of useful tools. Although such tools are not directly usable (since they are tailored to the particular machines they are implemented on), it is possible to exploit the existence of these tools by adapting them to the needs of the task-oriented system under construction. In the past, general-purpose hosts have been used for software development.[3] In most cases, however, they are only used for code production and low-level hardware debugging. After the code is produced, and the end product transfered to the target system, the user is faced with a very poor environment in which to deal with hardware and software errors. Starting the execution and controlling the behavior of the program can only be performed within the target machine without assistance from the host.

Even if it were possible to build a sophisticated programming environment on the target system, the fact that some hardware components have to be built would prevent the debugging of software on the target system until later stages. This, in turn, would make hardware debugging more difficult because of the lack of proper software support. Software debugging is also more difficult because of the lack of reliable hardware on which to perform the test. This is a well-known problem for all systems. For example, in large mainframes, the savings of being able to start software testing early in the development of a new system is large enough to offset the expense of developing complete emulators, e.g., see the Yorktown Simulation Engine [5]. Although a tailored system is not replicated so many times to offset the expense of building a complete emulator, the problem cannot be ignored. One solution is to build functional emulators in software on the host for components that are novel and interface them with standard components (e.g., off-the-shelf microprocessors) to form a hybrid hardware–software system capable of running the application program. In this way, the complete system can be made to work (albeit with degraded performance) much earlier without waiting for all the hardware components to be ready. When hardware components become available, the respective function can be migrated from the host to the target system without altering the other parts of the system.

The requirements on the programming environment change during the development of the target system. Not only are different functions needed for different phases of the development, but also the different system components may be visible at different levels of abstraction. For example, in the phase of bootstrapping, a system designer might find it convenient to view the system as a collection of memory locations and registers. In a later phase, it might be more useful to look at the system as a collection of processes that exchange messages. Moreover, the most convenient level of abstraction might not always be the same across the target system. For example, one might be interested in having some processors visible at the application language level while the control of a different processor might be at the microcode level.

C. The Harpy Machine Programming Environment

The approach taken for the Harpy machine was 1) to use the host on-line, i.e., to couple it tightly with the target system, 2) to build a consistent programming environment with a well-defined interface, and 3) to implement new and adapt existing tools so that they make use of this interface. In the Harpy Machine Project we tried to make the host the center of control for each stage of software development so as to initiate and supervise all activities of the target machine entirely from the host. In fact, there was no provision to communicate with the target system other than through the host (no terminal connection and no magnetic tape facility). The UNIX system on a PDP-11/40 was chosen as the host because it provides a comfortable working environment and allows for connecting systems without major system modifications. To achieve consistency and flexibility in software development support, we decided to provide an entire programming environment rather than a collection of separate support tools. By using the support system kernel as a common interface, it is easier to adapt existing support tools and to make new tools portable to other systems. The programming environment kernel consists of three subparts: a hardware interface, a message system, and a primitive command language.

The hardware interface allows the host to access the target system. The anticipated architecture of the Harpy machine did not provide an easy way to connect its components to the host. Furthermore, the communication links available were designed to give the individual LSI-11 processors access to the DSM, but not between the processors. In Fig. 3, the dashed lines indicate the interconnection links that were added for system development. These links are of two different types. The links connecting the LSI-11 processors are simple serial lines while the link to the DSM is a special parallel port (called *hooks*). The LSI-11 requires only global control while specialized hardware like the DSM requires fine-grained control.

As Fig. 3 shows, the communication for system operation (application program access) is separate from the communication used by the programming environment. The method of having a connection scheme solely for system development has several benefits. The communication system can be simple because it does not have to include complex communication protocols of the application. As a consequence, the communication system can be brought up rather quickly and can be used in early phases of the system development process. Programming environment functions can be active during normal operations (e.g., tools for monitoring system behavior). At any time desired, the host can access the system operation without interfering with the application.

The message system reflects the simplicity of the hardware connection scheme. Messages flow only from the host to the target system processor and vice versa, not between LSI-11 processors (except for the one LSI-11 processor that, with regard to the programming environment, merely serves as a concentrator and distributor for the message streams). Messages are typed and of arbitrary length (limited only by internal buffer sizes). Reliability of communication is achieved by attaching and verifying a checksum with each transmission. The message flow is regulated by a "quota" technique. Each sender maintains a quota that is initialized to the buffer spaces involved in the transmission. The receiver sends a message back which increments the quota. To avoid blocking the message system, messages that contain quota information are treated differently than regular messages. Similarly, there is a special mechanism for transmitting exceptional messages (e.g., transmission error, special interrupt conditions, traps, and process stop).

The command language on top of the message system implements access mechanisms of various degrees of sophistication.

[3] These systems are often called "development systems."

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.

There are basic mechanisms to access all relevant components of the target system (e.g., read and write memory locations and registers, manipulation of the DSM hooks). In additon, there are more complex functions that have some built-in knowledge of the system structure (e.g., loading and initializing individual processors and the whole system) and of the run-time environment (e.g., starting, stopping, and proceeding a C-language process by proper handling of the run-time stack and of the internal machine state.

The support tools made extensive use of programming environment kernel. Existing tools (e.g., the UNIX "sdb" debugger) could be easily converted to this common interface, and new tools were written as an extension of the command language (e.g., the DSM microcode debugger). All the tools accessed the target system through the programming environment kernel. The following is a list of the key features of the Harpy machine programming environment:

Compiler and linker/loader: The PDP-11 UNIX C-compiler was used. The C-library supplied all functions that did not require special support through the UNIX run-time system. Except for the small run-time environment written in LSI-11 assembly code (for accessing registers and starting interrupt handlers) the entire software for the Harpy machine was written in C. An "absolute location" linker/loader that could relocate object files to absolute locations in the LSI-11 memory was written.

Simulator: A simulator of the complete system was developed on the host. Since no simulation facility was available, the simulator was implemented as a collection of UNIX processes. Different components were simulated at different levels. For example, the search processors were simulated at the C language level while the DSM was simulated at the "user function" level. The communication between modules was simulated in detail. The simulator could execute the same tasks as the target system. Different components could be moved out of the simulator and out of the host into the target system simply by changing the interface between modules. The simulator is described in [1].

Microassembler: The microassembler was "borrowed" from a previous project, and modified to include the special features of the DSM.

System loader/message system/contol support: This software was initially built on the host, with some additional code inside the LSI-11 and the DSM. As the hardware and software of the machine improved, many of the software support functions were moved to the target system.

Debugger/diagnostics: The C language debugger was obtained by modifying the UNIX "sdb" debugger and interfacing it to the target system by means of the message system/control software. The DSM debugger took more effort because it had to be designed and implemented from scratch. The programming environment also made it possible to switch between the two debuggers without altering the state of the target system, thereby obtaining a "variable level" of abstraction in the system. The programming environment could automatically control the execution of diagnostics on the target system and also facilitated the collection of reliability data.

D. Summary

Unlike general-purpose systems where each new design is expected to come with its own complement of operating systems, compilers, editors, and other system utilities, it is not necessary for a custom architecture to have a full complement of software running on the target system. A closely coupled host that has all the support facilities can provide both the needed programming environment and a close control over the debugging and execution of the application program.

VI. Performance Evaluation

Measuring the performance of task-oriented systems to evaluate the effectiveness of the architecture and of the implementation requires the measurement of a number of parameters of different components of the machine, e.g., the execution time of a software module, the utilization of a hardware functional unit, or the traffic on a communication link.

A. Measuring Task-Oriented Systems: What is Different?

One of the most common problems with performance evaluation of general-purpose architectures is *the lack of proper tools*. Few projects have achieved the goal of designing and implementing the necessary performance evaluation tools together with the system itself. One reason for this is that many of the measurements necessary in a general-purpose system environment require tools powerful enough to deal with parameters whose importance is not known when the tools are designed. This is especially true for software tools that must deal with user programs. On the other hand, in task-oriented architectures the important parameters are usually well defined from the beginning.

Another problem for general-purpose measurement tools is that *general-purpose systems are often shared*: the multiprocess and multiuser environments of general-purpose systems require the capability to access information that is usually hidden at low levels in the system while at the same time maintaining the necessary protection and security qualities of the environment. Since a task-oriented system is usually a collection of process cooperating to solve the same problem, security is not an issue and protection can be, in part, trusted to the application software.

The *interference between the measurement tool and the system being measured* is another major problem. Fortunately, when measuring computer systems, interference can often be evaluated and factored out of the results. For example, some measurement tools like SMT [8] explicitly compute the overhead of calling and executing a measurement routine and subtract it from the time returned by the real-time clock. Although this technique is sufficiently accurate when measuring the behavior of a program for its own sake, the perturbation caused by the measurement software can substantially change the behavior of a parallel computer whose architecture has been optimized on the basis of a particular behavior of the processors. For example, in the Harpy machine, adding a procedure call in the search processors increases the time of execution, thereby changing the frequency of requests to the DSM which, in turn, drastically alters the performance of the entire system.

Finally, *the quantity of measurement data* can raise a number of problems. During measurement of a program in a time-shared system, data can be sorted and analyzed on the same system that is being measured. Special-purpose systems present real problems since measurement data have to be transferred to a host on special communication paths.

B. A Measurement Tool for Task-Oriented Systems

Since a measurement tool is a rather large subsystem, it is usually impractical to build an ad-hoc tool for each task-oriented system that one implements. On the other hand, each system might have completely different requirements. It is convenient to be able to distinguish between the operation of detecting *events* that happen in the system and the compu-

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.

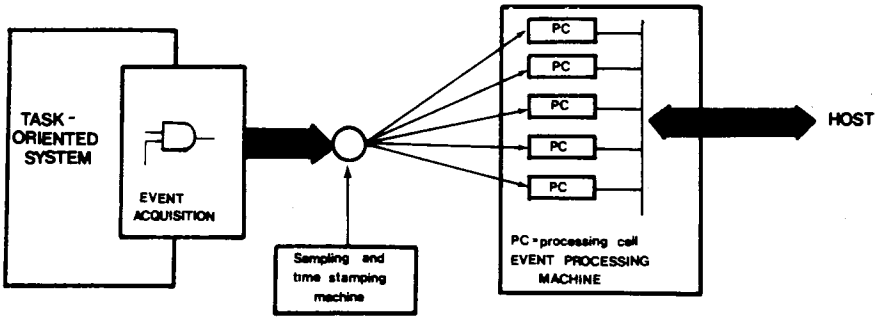

Fig. 6. The performance evaluation machine.

tation necessary to extract the information from a *sequence of events*. Let us examine some of the requirements for a measurement tool that can count and time events.

What is an event? Events are changes of state in the machine that mark a salient change in what the machine or part of it is doing. For example, if there is a communication link between two components over which messages are transmitted, each message might be an interesting event. Another possible event could be a procedure call in a processor or accessing a given area of memory. It is possible to collect events by means of simple hardware, like a comparator or a gate; but sometimes it can be convenient to use a combination of hardware and software, e.g., an instruction that sets a bit in a register that is accessible from the outside. If events are qualified with a known timing source, e.g., sampled by a clock and stamped with their time of occurrence, they can be used to extract information relative to the number of occurrences of events and to the time between events. Because events are specific to a given system, they have to be acquired more or less in an ad-hoc fashion and embedded in the structure of the system.

Fig. 6 presents the structure of a performance evaluation machine suitable for task-oriented systems. It consists of a system-dependent "event acquisition" module, a sampling and time stamping clock, and an event processing machine. The event processing machine consists logically of a collection of submodules, each dedicated to a different measurement (e.g., count the number of events of type A occuring between events B and C). All these modules are connected to the host machine to collect measurement data. The event processing machine cannot be a general-purpose processor because, although the logical operations it has to perform when events arrive are simple (e.g., increment a counter), many of these events may have to be measured every, say, 50 ns, which would be too fast for most conventional microprocessors. It is, however, possible to design simple structures that can deal efficiently with many events happening in parallel. For example, we designed (but did not implement) a pipelined architecture that can measure a number of properties on a stream of events and at the same time check that events follow each other in a way that is consistent with the correct behavior of the system. In this structure, events are clocked in parallel into a pipe of equal cells and the clock frequency is the same as the sampling frequency; therefore, cells are able to keep track of event times relative to each other by counting the number of pipeline clocks between them. Cells can be programmed to perform functions such as recognizing two different events and accumulating the elasped time between them. The same structure can also be used for the debugging of VLSI chips, and a realistic implementation of such a machine on a single VLSI chip is discussed by Bisiani *et al.* [3].

C. Measuring the Performance of the Harpy Machine

In the Harpy machine case study, we could not afford the time to completely build the performance evaluation tool that we have described. We only implemented the hardware and software necessary to recognize events. For example, in the Harpy machine the port behavior was one of the most important variables to measure. Hence, special hardware was provided in the port itself. Analogously, the software was designed such that it would be easy to measure the time spent in different parts of the code.

The sequence of events was then used as input to an ad-hoc collection of counters manually set up to perform various measurements. As a result, we had to run the complete test set through the machine many times to compute all the information we were interested in. We had to repeat measurements occassionally because the equipment was not set up correctly and spent a considerable amount of time in the whole process. Although building a performance evaluation machine like the one described in the previous subsection would have taken more time, its use in at least one more system would have made the extra time spent worthwhile.

The rest of this section describes the measurements performed on the case study. The amount of computation required for a recognition varies widely depending on the task used. When measuring the performance of the Harpy machine, we used two tasks: a 1000-word "Artificial Intelligence Retrieval" task (AIX05) and a 40-word "Desk Calculator" task (DESCAL). A typical sentence of AIX05 would be: "When was the last paper by Holland published?"; a typical sentence of DESCAL would be: "What is seven nine times epsilon?" The performance evaluation was done using prerecorded utterances. Since the same tasks and utterances had been used to evaluate other implementations of Harpy, they could be used for a comparison. One of the five processors received the data from the previous stages of the pipe (which were computed on the host) and sent them to the other four processors through the DSM. The same processor collected the resulting data and sent it to the host to be displayed on a terminal.

The events used were:

- *checkpoints in the software* were collected by means of an instruction inserted in the LSI-11 code. This instruction would set a bit in a register that was accessible from outside the machine (eight different events of this kind were used);
- the states of the processors' ports, i.e., *running, waiting to be serviced* by the data structure machine, and *being serviced* by the data structure machine. The events were collected by "tapping" the appropriate signals in the port (fifteen events of this kind were used, three for each of the five processors);

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.

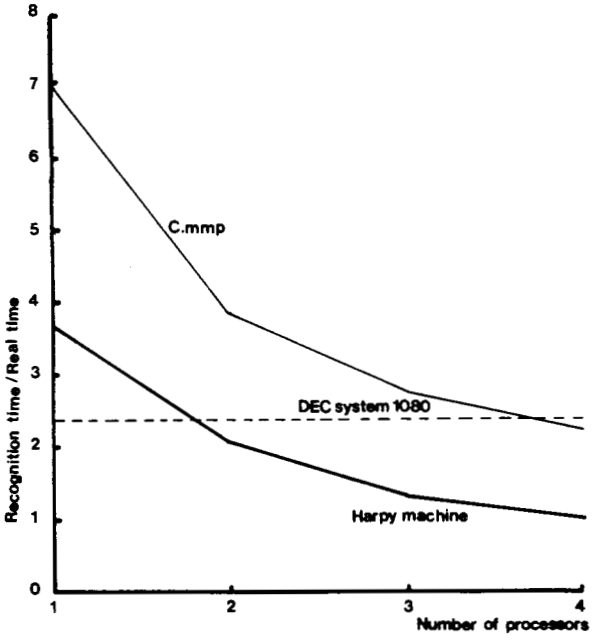

Fig. 7. Speed of the Harpy machine for the 1000-word artificial intelligence retrieval task.

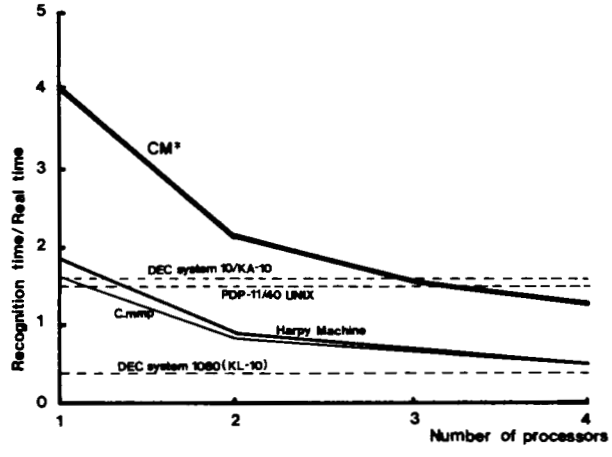

Fig. 8. Speed of the Harpy machine for the 40-word desk calculator task.

checkpoints in the data structure machine microcode; a bit in the microcode instruction was reserved to generate events under microprogram control (one event of this kind used).

The events generated by the software were used to measure the execution time (e.g., the data in Fig. 7) and the impact of different software modules within the processors. The "port" events were used to evaluate the utilization of the processors and of the DSM. The microcode-generated event was used to measure the performance of alternative microprograms in the DSM.

The speed of the Harpy machine is presented in Figs.7 (AIX05) and 8 (DESCAL) as the ratio between the recognition time and the length of the utterance (e.g., 1 means real-time recognition). The Harpy machine is faster than any of the previously implemented systems when running AIX05. Moreover, the system reaches real-time recognition with four processors and, when two processors are utilized, it is faster than a DECSYSTEM 1080. Note that even with one single processor the system is twice as fast as a single processor C.mmp (C.mmp uses 16 PDP-11/40's, each of which is roughly three times faster than an LSI-11). When running, DESCAL, the Harpy machine is less efficient: It barely equals C.mmp speed and never reaches the performance of the DECSYSTEM 1080. The main difference between AIX05 and DESCAL, as far as computational needs, is that DESCAL expands few nodes for each search step. Thus the amount of time it takes to set up a search step becomes comparable to the time it takes to process it. Since the DECSYSTEM 1080 is equally efficient in both computations and is much faster (as far as technology is concerned) than the LSI-11, it performs better. On the other hand, the Harpy machine is optimized for processing the search step on the assumption that in most cases this will take more than 90 percent of the computation. When the search step is small, the architecture is no longer tuned to the task, and the performance drops. More detailed performance data are discussed by Bisiani in [2].

D. Summary

A suitable performance evaluation tool is necessary to tune task-oriented architectures. By distinguishing between event processing and event collection it is possible to implement a tool that can be used for the measurement of many different task-oriented architectures.

It appears that, for the class of algorithms considered, efficiently dividing the algorithm into small steps leads to an increase in performance. However, there is a tradeoff between the improvement obtained by increasing the frequency of interaction and the performance loss due to the overhead generated by each interaction. Since the Harpy machine architecture has been designed to minimize interaction overhead for a certain class of algorithms, it can take advantage of a restructuring that divides the algorithms into small computational steps. It is particularly interesting to notice how the performance drops when the system is used with data (DESCAL) that have not been used in the algorithm behavior analysis. This clearly shows how important it is to discover the main characteristics of the algorithm and to design the architecture accordingly.

VII. Conclusions

In this paper we have discussed a number of issues of importance to a designer of task-oriented systems. These include: algorithm analysis and restructuring, architecture design, hardware development, software development, and performance evaluation. The main contribution of this research is the formulation of a technique for custom architecture development based on an integrated view of hardware, software, and system requirements of the application. In the case we have examined, the problem of hardware design and engineering tends to be just one part of the problem. Other aspects of the problem of equal importance are algorithm analysis, algorithm-to-architecture mapping, operating system, debugging, and performance analysis. These have received relatively little attention in the past.

This paper presents a number of concepts for the design and development of task-oriented architectures:

The concept of semi-automatic analysis of algorithms independently of implementation and architecture.

The concept of a hybrid parallel-pipeline architecture with intelligent global memories.

The concept of interchangeable hardware/software modules and gradual migration of software modules (running in a

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.

general-purpose machine) to hardware modules (running in the target machine).

The concept of a closely coupled host that is not only used for software development but also for control, debugging, and measurement of the target system.

The concept of a performance analysis engine capable of contemporaneously measuring many events that belong to different abstraction levels.

With proper tools it is possible to design, develop, and evaluate new architectures with only a few man-years of effort, as in the case of the Harpy machine. It is our belief that ideas and techniques presented here are useful in a wide variety of problems requiring task-oriented systems.

ACKNOWLEDGMENT

The authors would like to thank J. Gasbarro and J. Ousterhout for their contribution during the design and implementation of the Harpy machine. They would also like to thank A. Newell, D. Siewiorek, A. Fisher, M. Foster, K. Oflazer, and P. Jackson for their comments on the manuscript.

REFERENCES

[1] R. Bisiani, "The role of simulation in the development of task-oriented architectures," in *2nd Annu. Symp. on Small Systems*, ACM, Dallas, TX, Oct. 1979.
[2] ——, "The Harpy machine: A data structure architecture," in *5th Workshop on Computer Architecture for Non-Numeric Processing*. ACM SIGARCH, SIGIR and SIGMOD, 1980.
[3] R. Bisiani, M. Annaratone, and M. J. Foster, "An architecture for real time debugging of custom VLSI chips," in *1983 Int. Symp. on VLSI Technology, Systems and Applications*, Mar. 1983.
[4] W. R. Deniston, "SIPE: A TSS/360 software measurement technique," in *Proc. ACM Nat. Conf.*, ACM, 1969.
[5] M. Denneau, "The Yorktown Simulation Engine," in *Proc. 19th Design Automation Conf.*, IEEE, New York, June 1982.
[6] P. Deutch, "A flexible measurement tool for software systems," in *Information Processing 71* (Proc. IFIP Congr.). Amsterdam: North-Holland, 1971.
[7] D. Ferrari, *Computer Systems Performance Evaluation*. Englewood Cliffs, NJ: Prentice-Hall, 1978.
[8] D. Ferrari and M. Liu, "A general purpose software measurement tool," *Software-Practice and Experience*, vol. 5, no. 2, 1975.
[9] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. Computer Sci. Press, 1978.
[10] A. K. Jones *et al.*, "Programming issues raised by a multiprocessor," *Proc. IEEE*, vol. 66, no. 2, pp. 229-237, Feb. 1978.
[11] D. E. Knuth, *The Arts of Computer Programming*. Reading, MA: Addison-Wesley, 1969.
[12] ——, "An empirical study of Fortran programs," Tech. Rep. CS-186, Comput. Sci. Dept. Standord Univ., Stanford, CA, 1970.
[13] B. T. Lowerre and R. Reddy, "The Harpy speech understanding system," in Wayne A. Lea, Ed., *Trends in Speech Recognition*. Englewood Cliffs, NJ: Prentice-Hall, 1980.
[14] P. N. Oleinick, "The implementation and evaluation of parallel algorithms on C.mmp," Ph.D. dissertation, Comput. Sci. Dep., Carnegie-Mellon Univ., Pittsburgh, PA, 1978.
[15] J. W. Pomery, "A guide to programming tools and techniques," *IBM Syst. J.*, vol. 11, no. 3, 1972.
[16] I. E. Sutherland and C. A. Mead, "Microelectronics and computer science," *Scientific American*, vol. 237, no. 3, Sept. 1977.
[17] R. J. Swan, "The switching structure addressing architecture of an extensible multiprocessor: Cm*," Ph.D. dissertation, Comput. Sci. Dep., Carnegie-Mellon Univ., Pittsburgh, PA, Aug. 1978.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 21,2020 at 20:57:04 UTC from IEEE Xplore. Restrictions apply.