PROCEEDINGS

OF THE

1985 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

August 20-23, 1985

Douglas Degroot
Editor

Co-Sponsored by

PART IV

PROCEEDINGS

OF THE

1985 INTERNATIONAL CONFERENCE

ON

PARALLEL PROCESSING

August 20-23, 1985

Douglas Degroot
Editor

Co-Sponsored by

Department of Electrical Engineering
PENN STATE UNIVERSITY
University Park, Pennsylvania

and the

IEEE Computer Society

In Cooperation with the

Association for Computing Machinery

PART IV

COMPUTER
SOCIETY
PRESS

IEEE COMPUTER SOCIETY

1884 · 1984 THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

A Graph-Based Computation Model for Real-Time Systems

Aloysius K. Mok[†]

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract

Real-time systems are unique in that their success depend on the capability to meet very stringent timing constraints involving multiple parallel activities. As a result, they are among the most difficult systems to build and to verify. In this paper, a methodology to automate the synthesis of code for time-critical applications is presented. This methodology is built on a graph-based model to capture the computational requirements of time-critical systems. In terms of this model, we can perform resource allocation and other analysis which are fundamental to the computer-aided design of time-critical systems.

Key words: real-time systems, design methodology, concurrent processes, parallel computation model, requirements specification, algorithms, scheduling, program synthesis

Introduction

An important application of computers has been to control physical processes such as regulating a power plant or guiding the flight of an airplane. These so-called embedded systems have two important requirements: (1) the capability to prevent the loss of essential functionalities in the face of partial failures; and (2) the capability to meet critical timing constraints. These two capabilities are closely related in the *hard-real-time* environment where the system must satisfy assertions not only about the logical integrity of the software, but also about the absolute timing of computational events.[1] The design and maintenance of these systems has been widely recognized as posing some of the most challenging problems in systems research. Some researchers e.g., [SCH & SCH 81], [RAND 75] have concentrated on techniques to maintain the logical integrity of these systems. In this paper, we shall describe a design methodology which is capable of satisfying critical timing constraints involving multiple parallel activities and thereby guaranteeing assertions about the absolute timing of computational events. More importantly, our methodology can be mechanized and is in fact a distillation of our experience with the successful implementation of a very high level language for real-time control applications at MIT [WARD 78].

Software Automation Strategy

The general strategy of our research involves the following steps:

(1) Capture the unique computational characteristics of the hard-real-time environment by formulating an appropriate formal model.

(2) For each problem instance, translate the design specifications into an instance of the formal model for resource allocation and other analysis.

(3) Perform resource allocation analysis to arrive at an implementation which meets the specified critical timing constraints.

The key step in our strategy is to identify an appropriate computational model so that we can translate user specifications into an instance of the model for resource analysis. The choice of an appropriate model is important because it must be sufficiently expressive so that computational requirements involving multiple activities and timing constraints can be naturally expressed in it, and yet the model must also be simple enough so that our design methodology can be mechanizable in practice. Past attempts in automatic program synthesis have met with limited success mostly because the optimization problems involved are often domain-specific and defy general problem-solving techniques. Our model is formulated by taking advantage of the domain-specific characteristics common to many hard-real-time systems. Specifically, (1) a precise way to characterize their workload is possible; and (2) the computational events performed by system components at the level of major functional elements are relatively simple: a computational event is either a functional transformation or transmission of data values subject to pipelining constraints.

In the following, we shall first briefly review some related work which uses process-based computation models. A formal definition of our graph-based model and its motivations will then be given. We shall give a summary of our results in resource allocation analysis in the context of our graph-based computation model.

[†] Supported by the Office of Naval Research under ONR contract N00014-85-K-0117. Part of this work was done under DARPA contract N00014-75-C-0661 while the author was at the Real Time Systems Group of the MIT Laboratory for Computer Science.

[1] As an example of a fault in absolute timing, the first flight of the Space Shuttle was delayed by a synchronization error which was traced to an improbable race condition when several processors disagreed on the *current value of time*. [GARM 81]

Process-Based Models

The work that is most closely related to our research are the design methodologies that have been proposed for writing real-time programs. Most of these methodologies are not directly relevant to hard-real-time applications in that they do not address the problem of verifying compliance to timing constraint specifications and most are not sufficiently precise for automation. Notable exceptions are [DERT 74], [LEIN 78] and [WEI et al 80]. In their work, critical timing constraints are specified by permitting a process to have a deadline and/or repetition period attribute. In [MOK 83], we have made a detailed study of process-based models for specifying hard-real-time systems and have devised algorithms to generate run-time schedulers tailored to meet performance specifications even when there are both synchronization and (under certain restrictions) mutual exclusion constraints.

Our research has also noted some significant shortcomings of process-based models for software automation purposes.[2] For example, by formalizing the computational requirements of an application in terms of a set of processes before analyzing the tradeoff between communication and computation, a designer may have unnecessarily ruled out certain system architectures for implementation. Process-based models are also awkward when the system architecture includes non-von Neumann type processors such as systolic arrays. The result is that design methodologies that are based on process-based models are limited in their usefulness.

A Graph-Based Model for the Hard-Real-Time Environment

The formal model that we are proposing results from the cumulation of our experience with the function block schemata which is the basis of CONSORT [WARD 78], an experimental very high level language which has been implemented at MIT. CONSORT has a graphics interface which allows a control engineer to define controller structures and specify *latency* constraints. Our model is significantly different from that of CONSORT. For motivation, an example will immediately follow the definition below.

A *graph-based model* M is an ordered pair (G,T) where G is a *communication graph* and T is a set of *timing constraints*. Specifically, G = (V,E,W_V) is a digraph where V and E are respectively the node and edge sets and W_V is a function which assigns a non-negative weight to each node in V. T is a finite set of timing constraints each of which is a tuple (C,p,d) where C is a *task graph*, p and d are respectively the non-negative integer *period* and *deadline* of the timing constraint. Intuitively, the nodes and the edges are meant to model the functional elements and their communication paths in an application.

A task graph C is an acyclic digraph which is compatible with the communication graph G. We say that the graph C is *compatible* with the graph G iff there is a mapping h such that: (1) if v is a node in C, then $h(v) \in V$; and (2) If e is an edge from a node u to another node v in C, then h(e) is an edge from h(u) to h(v) in E. (A task graph is meant to define the precedence relation of the computational events that must occur in order to satisfy a timing constraint. The nodes and edges of a task graph denote respectively the execution of the corresponding functional elements and transmission of data in the communication graph.) The computation time of a timing constraint (C,p,d) is the sum of all the weights of the nodes in C.

The set of timing constraints T is composed of two non-intersecting subsets: T_ρ and T_a. If a timing constraint $(C,p,d) \in T_p$, then it is invoked automatically every p time units, starting from time = 0 (we say that the timing constraint is *periodic*). If $(C,p,d) \in T_a$, then it can be invoked at any integral time instant t with the provision that two successive invocations of the same timing constraint must be at least p time units apart (we say that the timing constraint is *asynchronous*). If a timing constraint (C,p,d) is invoked at time = t, then the task graph C must be executed in the interval [t, t+d]. A task graph C is said to be *executed* in a time interval I if a subset S of the set of (instances of) functional elements that have been executed in I forms a partial order such that: (1) There is a bijective mapping between the functional elements in S and C; (2) Under this mapping, the partial order S is consistent with the acyclic graph C; (3) In the case where the functional elements are physically distributed and the graph C has an edge from a node u to another node v, then an execution of C must include the transmission of the latest output of the functional element u to the functional element v before the corresponding instance of v is executed in the time interval I. Furthermore, we require real-time computation to be *pipeline-ordered* in the sense that: (1) Two executions of a functional element must have distinct start-times and that the execution which has an earlier start-time must also finish earlier than the other; and (2) Two data transmissions from a functional element u to another functional element v must be sent at distinct instants at the site of u and the earlier transmission must also be received earlier at the site of v.

Example Design Requirements Specification

Figure 1 is the block diagram of an automatic control

[2] During our investigation, the author was at first reluctant to part with process-based models since the process abstraction has long been the basis for most software designs and more recently, theoretical models of distributed computation. The semantic gap between process-based models and the hard-real-time environment is, however, too serious to be ignored.

system. This control system has three inputs x, y, z and a single output u. There are five functional elements: f_X, f_Y, f_Z, f_S and f_K. The functional element f_S has two outputs one of which is fed back via f_K to itself so that u is a function of x', y', z' and its own previous value. The other output is to the external environment and has the same value. For brevity, we use the same name, u to denote the two outputs. The computation times of the five functions are assumed to be bounded and their maximum values are respectively c_X, c_Y, c_Z, c_S and c_K.

The design objectives of this system can be stated informally in terms of the computation required by two periodic and one asynchronous timing constraints as follows. The input x is to be sampled at the regular rate of $1/p_x$ cycles per second. (Sampling rates are determined by the dynamics of the physical process under control.) The output u must be recomputed by executing the function f_S with the new value of x' and recent values of y', z' and v (to be determined by their individual update rates). The internal state v must then be updated by executing f_K with the new value of u. The input y is to be sampled at a rate of $1/p_y$ cycles per second and the variables u and v must be likewise recomputed. The input z is a boolean signal, i.e., $z \in \{0,1\}$, and can change state asynchronously. When a state transition occurs, the new value of z must be detected and a new z' computed by executing f_Z. The output signal u must also be recomputed by f_S within d_z time units. The input z is assumed to change state very infrequently compared with p_x and p_y.

A physical interpretation of this block diagram is to regard f_X and f_Y as the preprocessors of signals from two sensors measuring the physical quantities x and y one of which changes much more slowly than the other, hence the different sampling rates. The signal z can be regarded as the output from a toggle switch and u is the control signal to an actuator. The signal u is also used to compute an internal state to be used in subsequent calculations, e.g., f_K may be a state estimator in a compensator. f_S is used to determine the output from the inputs x and y and the internal state. The variable z' may be a parameter which selects a different mapping for f_S depending on the operating regime selected by a human operator via the toggle switch z.

Figure 2 shows an instance of our graph-based model which defines the design requirements of the example control system. It should be emphasized that the requirements specification language employed by the end user is of only secondary importance in so far as it permits a precise translation of user requirements into an instance of our graph-based model. Otherwise, it should be as natural as possible to the application domain, e.g., an avionics engineer may want to use a specification language such as the one devised by Parnas et al [HENI 80], and a robotics engineer may want to use an entirely different language to express the motion of a robot arm. The domain-specific nature of our model should make the translation from end user requirements to an instance of our model relatively straightforward.

Synthesis Techniques

A straightforward way to implement an instance of our graph-based model is to map each periodic/asynchronous timing constraint (C,p,d) into a periodic/asynchronous (i.e., demand driven) process T' where the body of T' consists of a straight-line program which is any topological sort of the operations in the task graph C. The computation time c of the process T' is then the computation time of C. In order to enforce pipeline ordering, we create a monitor [HOAR 74] for each functional element that occurs in two or more timing constraints. To improve efficiency, we can reduce the size of critical sections by software pipelining, i.e., decomposing a functional element into a chain of sub-functions each of which has the same computation time. (We now see one of the virtues of the graph-based model: all the data dependencies are made explicit and hence software pipelining can be easily automated.) The scheduling results for process-based models, e.g., [MOK 83] can now be applied to implement the resulting set of processes. However, this approach is inefficient since it does not take advantage of operations that are common to two or more timing constraints. For example, if p_x is equal to p_y in the example control system, then there is no reason why f_S should be executed twice per period. In the process model, there are two distinct calls to f_S and so the redundant work cannot be avoided. We introduce the *latency scheduling* technique for meeting asynchronous timing constraints which can take advantage of operations common to two or more task graphs. The latency scheduling technique is formulated in terms of the graph-based model as follows.

Let M = (G,T) be a graph-based model. An execution trace of a processor is a mapping F from the non-negative integers to $V \cup \{\phi\}$ where V is the set of functional elements in G and the symbol ϕ denotes an idle interval. F(i) = u if the functional element u is being executed on the processor in the time interval [i,i+1]; F(i) = ϕ if the processor idles in [i,i+1]. An execution trace F is said to have a latency of k time units with respect to a timing constraint (C,p,d) iff F contains an execution of C in any time interval of length $\geq$ k.

A *static schedule* is a finite string of symbols in $V \cup \{\phi\}$. A static schedule L is said to have a latency of k time units with respect to the timing constraint (C,p,d) iff the execution trace which a round-robin scheduler generates by repeating L *ad infinitum* has a latency of k time units with respect to (C,p,d). A static schedule L is said to be feasible with respect to a set of asynchronous timing constraints T_a iff L has a latency of d time units with respect to every timing constraint (C,p,d) $\in T_a$. Obviously, if we can compute a feasible static schedule, we can construct a run-time scheduler guaranteed to meet the given performance requirements.

The following are the key results for computing a feasible static schedule for a graph-based model (G,T) where $T_p = \{ \}$, i.e., all the timing constraints are asynchronous. The same results hold with minor modifications if $T_p \neq \{ \}$.

Theorem

If there is an execution trace F which has latency d with respect to every asynchronous timing constraint (C,p,d) in a graph-based model (G,T), then there must be a feasible static schedule (finite by definition) with respect to T.

This result shows that feasible static schedules can always be computed in finite time. The proof is by means of an appropriately constructed finite simulation game.

Theorem

The problem of determining whether a feasible static schedule exists for a graph-based model (G,T) is NP-hard in the strong sense for the following two restricted cases:
(i) All the functional elements in G have unit computation time and all the task graphs in T are chains of length 1 or 3.
(ii) Every task graph in T consists of a single operation; all but one of the deadlines are the same and the functional elements cannot be pipelined into chains of sub-functions.

Proofs are respectively reduction from 3-partition and cyclic ordering [GAR & JOH 79].

In general, we can employ a good heuristic algorithm which first computes a static schedule to satisfy the periodic timing constraints and then incorporates additional operations to satisfy the asynchronous timing constraints. Good heuristics are available, e.g., we can provide a lower bound on the performance of heuristic algorithms.

Theorem

Let w_i, d_i be the computation time and deadline of the i^{th} timing constraint. If (i) $\Sigma\, w_i/d_i \leq 1/2$; and (ii) $\lfloor d_i/2 \rfloor \geq w_i$; and (iii) all the functional elements can be pipelined, then a feasible static schedule always exists.

Even though optimal static schedules are hard to compute in general, it should be emphasized that the run-time scheduler is very efficient once a feasible static schedule has been found off-line.

We have also taken care in formulating the graph-based model such that for a multiprocessor architecture, the synthesis problem can be decomposed into a set of single processor synthesis problems and a similar-looking problem for scheduling the communication network. We shall report this work in another paper.

Conclusion

In spite of the rapid advance in hardware technology, processor power is still at a premium and will probably remain so for many real-time applications with the result that software for these applications needs to be highly optimized. We cannot expect to see substantial improvement in software productivity as long as we rely on manual efforts to fine-tune a system. In this paper, we have presented a concrete design methodology which can substantially automate the design and maintenance of real-time systems to meet critical performance constraints. The practicality of our methodology is partially demonstrated by the successful completion of the very high level language CONSORT. Learning from our experience with CONSORT, we are currently developing a more advanced design system for hard-real-time systems.

We have just embarked on a wide area of research. Our graph-based model provides a formal framework to investigate issues in real-time system design which often involve many parallel activities together with stringent timing constraints. There are also other interesting problems that can be formulated in terms of our model. For example, we can pose the problems of maintaining the logical integrity of real-time systems in terms of relations on the data values that are being passed along the edges of the communication graph of our model and devise more domain-specific fault-tolerance techniques. Another direction of research is to synthesize complete hardware-software systems from specifications based on our model by taking advantage of vlsi technology, such as along the line of the *system compiler* project of [DAS et al 83].

Bibliography

[DAS et al 83] B. Dasarathy, D.S. Prerau, J.H. Vellenga, "The System Compiler", *Proceedings of the IEEE 1983 International Symposium on Circuits and Systems, pp. 530-533.*

[DERT 74] M. Dertouzos, "Control Robotics: the procedural control of physical processes", *Proceedings of the IFIP Congress, 1974, pp. 807-813.*

[GAR & JOH 79] M. Garey and D. Johnson, *Computers and Intractibility: a Guide to the Theory of NP-Completeness, W.H. Freeman, San Francisco, California, 1979.*

[GARM 81] J.R. Garman, "The Bug Heard 'Round the World", *Software Engineering Notes, vol. 6, no. 5, Oct. 1981, pp. 3-10*

[HENI 80] K.L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and their Applications", *IEEE Transactions on Software Engineering, vol. SE-6, no. 1, Jan. 1980, pp. 2-13.*

[HOAR 74] C.A.R. Hoare, "Monitors: an Operating System Structuring Concept", *CACM, vol. 17, no. 10, Oct. 1974, pp. 549-557.*

[LEIN 78] D. Leinbaugh, "Guaranteed Response Times in a Hard-Real-Time Environment", *Proceedings of the 7th Texas Conference on Computing Systems, Houston, Oct. 1978, pp. 3.24-3.36*

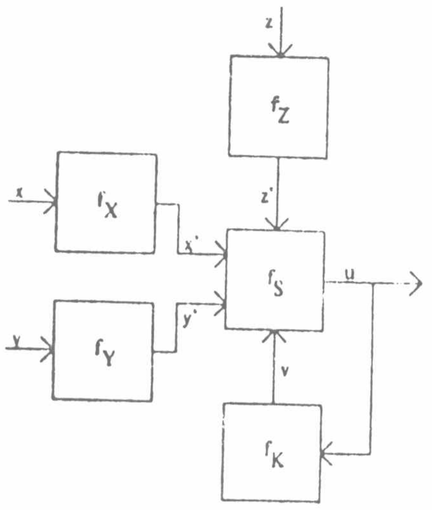
[MOK 83] A.K. Mok, "Fundamental Design Problems of Distributed Systems in the Hard-Real-Time Environment", *Doctoral Dissertation, Massachusetts Institute of Technology, May 1983*

[RAND 75] B. Randell, "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering, vol. SE-1, no. 2, June 1975, pp. 220-232.*

[SCH & SCH 81] F. Schneider, R. Schlichting "Towards Fault-Tolerant Process Control Software", *Proceedings of the 11th Annual International Symposium on Fault-Tolerant Computing, Maine, 1981, pp. 48-55*

[WARD 78] S. Ward, "An Approach to Real-Time Computation", *Proceedings of the Seventh Texas Conference on Computing Systems, Houston, Texas, Oct. 1978, pp. 5.26-5.34.*

[WEI et al 80] A.Y. Wei, K. Hiraishi, R. Cheng and R.H. Campbell, "Application of the Fault-Tolerant Deadline Mechanism to a Satellite On-Board Computer System", *The 10th International Symposium on Fault-Tolerant Computing, Kyoto, Japan, Oct. 1980, pp. 107-109.*



Example control system function block diagram
Figure 1

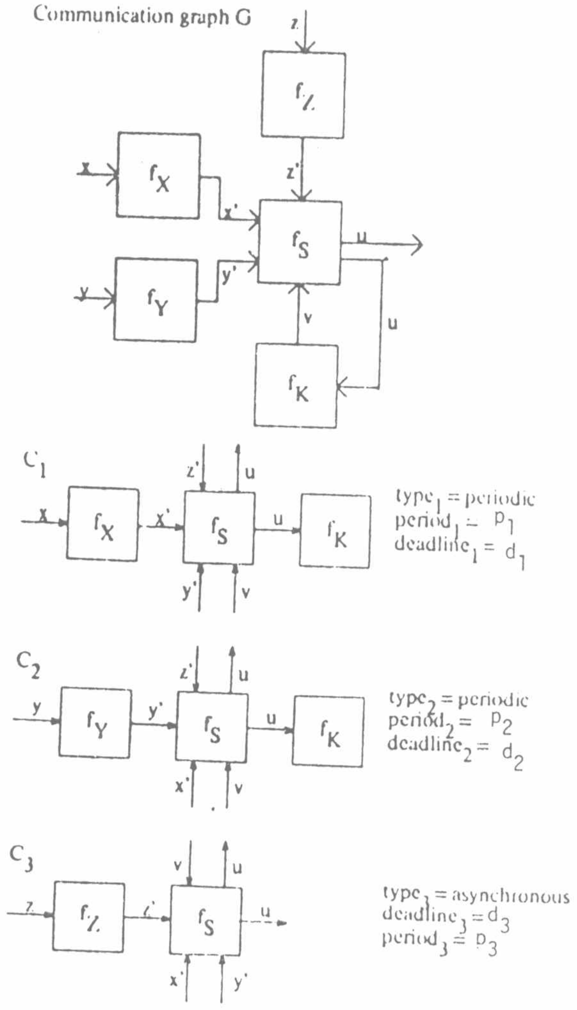


Figure 2
Specification of Design Example in Graph-based Model

Formulation and Programming of Parallel Computations: A Unified Approach

J. C. Browne
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

ABSTRACT

A graph model of parallel computations is sketched. The critical properties of the graph model are clean separation of computations from dependency relations and effective hierarchical structuring. The former property leads to a high degree of architectural independence for computations formulated in this model. The latter property unifies and integrates design and programming. The representation capability of this basis is shown by illustration to cover most popular models of parallel computation. Extension of the computation graph model across levels of abstraction from applications to chip level hardware is proposed.

INTRODUCTION

This paper presents a framework for formulation and representation of parallel computations. A parallel computation is defined as a sequence of programs for abstract machines which realizes with increasing resolution programmable representations of a model of parallel computation. A program is a directed graph where the nodes represent the binding of operations to data and the arcs represent dependency relations between schedulable units of computation. A critical property of the graph model of computations proposed here is a clear separation between the actual computations and the dependency relations between them which define the structure of a computation expressed in a given set of data types and operations upon instances of the types. This clean separation allows the computations and the dependencies to be separately bound to a target architecture. It also gives a degree of independence from the implementation of dependency mechanisms in hardware. Finally, it provides, through appropriate choice of dependency protocols, a unified approach to parallel computations. This representation basis can define programs for abstract machines over a range of resolution of detail from very abstract to realized hardware. This representation basis gives a unified framework for mapping from abstract specifications to hardware realized machines down to the level of physical device layout.

This basis can, at an appropriate level of abstraction, represent most well-known methodologies for formulation of parallel computations. Examples are given in Section 4 Programs prepared in the representation basis will be shown to have a high degree of architectural independence. Representation of computations as directed graph models has been an important thread in the development of the theory of parallel computation. A partial list of significant papers includes [KAR66, KAR69, ADA68, EST63, MAR69, ROD69, PET62]. The use here of graphs is operational rather than theoretical or analytical. The purpose of the graph model defined and described herein is to support formulation and programming. This is, again, not entirely novel. Keller and Davis [KEL81, DAV81] have proposed graphical programming languages for data flow programs. Dependency graphs are the basis for compiler optimization [KUC81, COC72] and recognition of parallelism in programs [KUC77]. There have recently been more general studies of dependency graphs as a basis for general program transformations [FER83]. Visual programming, sometimes employing a graphical basis, is now a popular research subject [REI84, RUB85, WOR83]. The paradigm for program development described herein also owes debts to the literature of automatic programming [BAR79, KAN81, WAL82].

The next section defines and describes models of parallel computation. The section "Abstract Machines and Their Programs" defines realizations of models of computation and hierarchical sequences of abstract machines. "Computation Graphs" describes the graphical representation of computation structures (programs for abstract machines) which we propose. The section "Example Representations" illustrates representation of several well-known computation methodologies in abstract terms in the graphical representation. The section "Algorithms to Architecture" sketches the use of the computation graph model as a framework for design and specification of architectures or elements of architecture. "Computation Graphs as a Programming System" gives advantages of application of the computation graph model as a programming system.

The results reported will make it clear that the development of these concepts is at an early stage. This early report gives only a brief introduction to the properties of the model. We have already found practical results in the assistance these concepts offered to the coordinated implementation of two programming languages for parallel computations which utilize task level schedulable units of computation, one of which uses an explicit procedural specification of parallelism and the other of which uses an implicit specification of parallelism.

MODELS OF PARALLEL COMPUTATION

A model of computation is a conceptual specification of the following elements:

1. Primitive data types and operations on these data types

2. Rules for composing schedulable computations from primitive operators and data types and definition of the state of a schedulable unit of computation.

0190-3918/85/0000/0624$01.00 © 1985 IEEE

3. Procedures for constructing the name spaces in which the schedulable computation structures execute and for the binding of values to names

4. Representations for dependency relations between schedulable units of computation. The representations may include:

   a. Mode and granularity of synchronization between schedulable units of computation

   b. Constraint relations for the execution of schedulable units of computation

   c. Topology, mode and type specification for the units of information which are the basis of communication between schedulable units of computation

   d. Functional relationships between the set of dependency relations which complete the definition of the initial state of a schedulable unit of computation.

Bindings of operators to type instances are the elementary units from which a computation is constructed. Sets of operator/type instance bindings may be composed into units for which state is maintained. A schedulable unit of computation has a set of initial states, a set of final states and a sequence of active states. A name space is the set of objects reachable by a schedulable unit of computation. Binding of values to names defines the state of a schedulable computation. A schedulable unit of computation can be executed whenever it reaches an initial state. Dependency relations are defined between schedulable computations.

Schedulable units of computation must, in general, be executed in some constrained or specific sequence in order to implement a meaningful computation. Synchronization and sequencing dependencies specify ordering relationships between the schedulable units of computation. Synchronization mechanisms are defined by mode and granularity. Constraint relationships such as mutual exclusion for access to a named object by schedulable units of computation are commonly required.

Communication mechanisms are used to implement data dependencies. Communication mechanisms move information between name spaces associated with schedulable units of computation. A communication mechanism is characterized by a topology of connection, a protocol for initiation and termination of a binding mode (circuit or packet) for the resolution of names and a granularity of data transfer.

The selection of conceptual elements for a model of parallel computation determines the capabilities of the abstract machine which realizes the elements of the model of computation in programmable form.

ABSTRACT MACHINES AND THEIR PROGRAMS

An abstract machine is a realization of a model of parallel computation as a programmable machine. A program is a set of operator/type instance pair bindings composed into schedulable units which are ordered by dependency relations.

An abstract machine may realize different elements of a model of parallel computation at different levels of abstraction. For example, data types and operators may be resolved to a fine granularity and have precise specification of the semantics of each operator on its type instances while leaving dependency relation definitions at a high level of abstraction.

The initial formulation of a computation will begin by selection of a model of computation, definition of a high level abstract machine and preparation of a program for this high level abstract machine. The elements of the model of computation and their realization in the abstract machine may be application oriented. There may or may not be a realized machine which can execute programs in this first abstract machine. If not, then the program must be mapped by one or more transformations to an abstract machine for which there exists a realized machine which can execute programs. A program expressing the computation in the initial abstract machine will, for execution, be successively mapped between pairs of abstract machines until a program for a realized machine is obtained. It may be convenient to define several abstract machines which realize different degrees of resolution between an initial high level abstract machine and a realized machine which has desired execution properties. A given abstract machine may be realized by simulation on another machine, or by hardware, or by any mixture of realization modes. Coherence of definition of the sequence of abstract machines through use of compatible models of computation should ease the tasks of mapping between programs for the sequence of abstract machines.

The preceding discussion is a (rather pretentious!) description of the process of programming, compilation, linking and loading which is the mode by which informal or mathematical specification of computations is turned into executable programs on hardware realized machines. Definition of a sequence of abstract machines and mappings which yield efficient programs on hardware realized machines is difficult even for sequential computations. It has not yet been widely attained for vector architectures and will be even more difficult for parallel architectures due to the complexity added by inclusion of general dependency relations in parallel models of computation. It is even more important for parallel computations to ease the task of generating effective mappings by use of a coherent framework for the definition of the sequence of abstract machines leading from specification level programs on unrealizable machines to executable programs on hardware realized machines. The next section describes how computation graph models of programs may be an effective framework for resolution from specification to execution.

COMPUTATION GRAPHS

A computation graph is a program for some conceptual computation in terms of the operations of an abstract machine. It is a directed graph where the nodes represent schedulable units of computation and arcs represent dependency relationships between source and sink pairs. Execution of the computation is attained by traversal of the directed graph along the paths defined by the dependency relations associated with the arcs. It is only the binding of operators to type instances which is always associated with a node. The program defining the schedulable unit of computation and the type instances may either or both arrive as values on arcs or else be permanently bound to a node. A node is therefore an abstraction for execution on a processor and memory. An arc is an abstract representation of a communication channel which may include memory or execution of a synchronization protocol. The initial abstract machine will normally have a processor for every schedulable unit of computation and sufficient channels and control lines coupling the processors so that each dependency relation can be assigned its own set of resources.

One critical property of the computation graph representation of programs for abstract machines is its natural implementation of hierarchical resolution. Any computation can, at a sufficiently high level of abstraction, be defined as a single schedulable unit of computation with inputs and outputs. Resolution of the computation to greater detail is obtained by replacing a node by a subgraph. Each element in the graph can be defined with greater or lesser detail.

A second critical property of the graph model is its clean separation of computations and dependency relations. This clean separation allows each to be separately resolved and mapped. It enhances portability by allowing the portion of a code most likely to vary between parallel architectures to be retained in an abstract form as long as possible.

The third critical property is the declarative form of expression for dependency relations. This declarative form is also a critical element in portability since only the logical properties of the dependencies are specified, not how they are to be implemented. This graphical programming form can be envisioned as a vehicle for functional programming with much more flexible provision for specification of dependency relations.

There have been several studies of the equivalence of different forms of dependency relations. Allen, *et al* [ALL83] have discussed the equivalence and conversion of control dependency to data dependence. Pingali and Arvind [PIN84] have analyzed equivalence between data and demand driven data flow modes of computation.

The next paragraphs sketch in abstract terms a computation graph basis which covers and integrates most of the common formulations for parallel computations. The definitions are given only with detail enough to coverage of and differentiation among the parallel programming methodologies discussed in Section 4. A detailed specification of the syntax and semantics of the proposed computation graph language with examples is given separately [BRO85]. Space limitations preclude full definition here. The model as specified generates hypergraphs including cycles [BER62].

Each schedulable unit of computation has a unique name which may include indexes to facilitate compactness of representation. Each schedulable unit of computation has at least one initial state, a sequence of active states and at least one final state. An initial state is defined by the persistent internal state (if any) of the schedulable unit and by the binding of values from one or more input arcs to the uninitialized names of the schedulable unit of computation. The final states of the computation must include the assignment of values (possibly null) to the output arcs. A dependency relationship is either a synchronization relationship between schedulable units of computation or a producer/consumer relationship. There are two types of synchronization relationships, those which specify an ordering between execution of schedulable units of execution and those which specify only that some set of schedulable units of computation cannot execute simultaneously. This latter constraint condition is represented by a coupled arc loop. Only one of a set of nodes connected by such arcs can be simultaneously active. Arc loops defining dependencies to be implemented by mutual exclusion protocols must have the data name as an attribute. Sets of arcs entering or leaving a given node may be joined by either "and" or "or" conditions. There are special nodes called source nodes which are external to the computation and generate the type occurrence values which determine the initial state for schedulable units of computation bindings that do not have predecessors. Arcs from source nodes are input arcs. There is no assumption in this model that the objects which traverse arcs are data objects. They may be data objects, operator objects, composite operator/data objects or control signals. Each arc has a unique name (which may be indexed) and carries a set of attributes. The attributes include specification of the type of the dependency (an example might be data driven data flow or procedure call) and, if data flows on the arc, the unique name of each data item. Attributes may be assigned globally (or to subgraphs). An example is a specification that all dependencies in a given subgraph are packet-type messages. Arcs without sink nodes carry the result of the computation.

A computation graph need not be static. A possible result of an operator/data binding at a node is the creation of a new arc, a new node or an entire subgraph. A node can always be replaced by a subgraph, provided that the dependency relations map properly. Mapping of dependency relations is by name matching.

A program is conceptually executed and a computation is evaluated by satisfaction of the input dependency relations and traversal of the paths through the graph which originate in source nodes and terminate on the output arcs. Actual evaluation requires mapping of the computation graph to an executable machine (which may require the addition of further arcs to the graph) and traversal of the graph in its mapped form.

An example may help in clarifying application of this representation to actual programming. A computation may, for example, be specified in terms of schedulable units of computation which can be realized as procedures in a higher level programming language such as Pascal or Ada and dependency relations between these schedulable units. The dependency relations may be of several types including some specific order for the reading of external inputs such as sensors or data dependencies such as the outputs of one procedure being inputs of other procedures. The schedulable units of computation are resolved by being written as Pascal procedures with appropriate encapsulation to implement interaction with the protocols which implement dependency relations. The dependency relations may be resolved to message or shared memory synchronization operators depending on which is provided at the next lower abstract machine. If the final target machine is Pascal and UNIX, then the procedures implementing the schedulable units of computation would be encapsulated as UNIX processes and the dependencies programmed as PIPES and/or SIGNAL/WAIT commands. The computation could straightforwardly be ported to another execution environment provided the protocols for implementing dependency relations have been implemented in the environment.

Programs utilizing hardware, operating system and language run-time system realized target machines are representable in this graph model. Schedulable units of computation are mapped to processors and memories. Dependencies are mapped to logical or physical channels or to synchronization operator/type instance pairs. The difficulty in this final step of mapping from abstract machine to executable machine is that we now lose freedom to specify arbitrary numbers of processors, etc. It will commonly be the case that the abstract computation graph will have many more nodes than the graph of the realized architecture since the nodes of the architecture are processors. It is generally necessary (and desirable) to map many nodes of the abstract computation graph to a processor node of the abstract machine graph. An optimal mapping will minimize total execution time of the computation. This requires simultaneous minimization of time spent satisfying dependency relations with maximum parallelism. These optimizations are difficult to formulate and optimal solutions are frequently of exponential complexity. There are, however, many heuristics from graph theory which have promise. It is important to note that optimizations can be applied to programs for high level machines where the nodes may have problem-oriented semantics which can assist the optimization algorithm.

EXAMPLE REPRESENTATIONS

This section describes informally how several well-known methodologies for parallel computation are represented in the computation graph basis.

Data Flow

Data flow models of computation [DEN72, KEL82] have always used a graphical representation. The additional levels of abstraction in the computation graphs defined preceding allow for integration of the sub-cases of data flow models of computation. The principal parameters in data flow models of computation are granularity of unit of computation, data/demand control, token/stream representations of data and static or dynamic graph structure.

Demand-driven or data-driven data flow programs are both defined by selecting the data types and operators for the abstract machine and an appropriate protocol for data flow control. An instruction level data flow computation is defined by a program in an abstract machine where the operators and types are instructions at the level of "floating point multiply on pairs of real numbers," etc. The communication mechanisms are packet movement operators which transfer data from source nodes to sink nodes.

Specific models of data flow computation are obtained by definition of the protocols which implement the dependency relations. The sequence of operations in the demand-driven model of computation is that the sink node requests data from the source node and the source node responds. A data-driven methodology employs a communication protocol where source nodes send results to sink nodes and perhaps await acknowledgement of result arrival or use. In each case the functions are embedded in the nodes and data objects are carried on the arcs. The computation graph model integrates demand - and data-driven control since the protocol for each arc can be separately specified. Note that a computation graph node can have some arcs using a data-driven protocol and some arcs using a demand-driven protocol provided the schedulable unit of computation is defined with the appropriate set of initial and final states. The Eazyflow [JAG84] and Rediflow [KEL84] data flow models of computation both incorporate this capability.

Functional Programming Languages

A functional programming language such as SASL or KPC [TUR81] can be viewed as a set of equations where the dependency relations are determined by the occurrence of variables on the left and right hand sides of function definitions. These languages define static computation graphs where the nodes are the evaluation of the right hand side of the equations and arcs are producer-consumer relationships carrying results of function evaluations. Functional programming languages typically do not specify modes of data movement so that the specification of these operators can be deferred until the graph is mapped to a realized machine for execution.

Object-Oriented and/or Distributed Programming

Consider a computation graph where each function is implemented by a type manager which also stores the occurrences of the objects it creates and implements concurrency control on access to the instances of its type. In this object-oriented style both state data and functions are permanently stored at nodes. The objects which traverse the arc are control signals and parameters for (and results from) the functions implemented by the type managers. This representation is one commonly discussed in models of distributed computation.

Procedural Programming

To represent procedural programming in this graphical framework we separate actual computations from sequencing and communication operators. The control statements in a procedural programming language can be seen to specify a traversal of the computation graph where the nodes are the actual computations and the sequencing and communication operations of the procedural language program define and implement the arcs of the graph. The Computation Structures Programming Language (CSL) [BRO82] defined for the Texas Reconfigurable Array Computer (TRAC) [SEJ80] is an example of a parallel language which procedurally implements this graphical representation of parallel computations.

Process/Transaction Model of Programming

The concept of processes originated in structuring of operating systems for concurrent execution of the internal computations necessary to implement multiprogramming on a single CPU. The transaction concept of dividing the actions of a process into units which do not interact with other processes was implicit in the mutual exclusion mechanisms of early operating systems theory and has been extended and formalized in the context of both operating systems and data bases. One popular approach to structuring of parallel computations which derives from this origin is to construct a set of processes composed of a sequence of transactions (straight-line graphs) and to complete the definition of the computation by introducing synchronization and/or communication actions between the processes. State is maintained for processes and transactions. A transaction is the smallest schedulable unit of computation. Dependency relations within a process are sequential. Dependencies between processes are implemented as interprocess interactions. The introduction of the interprocess interactions can be seen to create a computation graph where the arcs are of two types, sequencing within processes and interprocess interactions. (Each straight-line segment may have its own internal state extending across its nodes if the computations are not entirely enclosed in transactions). The equivalent graph will be obtained in the computation graph representation by choosing the transactions as schedulable units of computations (incorporating the code surrounding transactions as appropriate) and partitioning the resulting graph into straight line segments. Formulating the computation in the graph representation may call attention to partitionings into processes which would not be straightforwardly encountered by direct composition. The graph representation offers a basis for analysis not easily extracted from the interacting processes basis. It seems to us that the computation graph approach is the more fundamental and, indeed, more likely to lead to an efficient computation structure.

Distributed Data Bases

The convention that data remains at nodes and that functions move along the arcs, perhaps with parameters, is a model which has been proposed for distributed data base systems.

FROM ALGORITHMS TO ARCHITECTURES

The importance of special purpose algorithmically specific architectures will grow as their development is eased by "silicon compilers". The computation graph model is a natural front-end for silicon compilers.

The computation graph model can be used to define desirable abstract machines for the execution of given computations as well as to ease mapping from problem-oriented abstract machines to realized architectures. It is possible, in principle, to realize an abstract machine which directly implements the abstract machine defined by the schedulable units of computation and dependency relations of a given computation graph. There are a substantial class of computations where near direct realizations of computation graphs are possible. These cases include matrix operations, convolutions, etc. Chen [CHE84] defines a desirable two-dimensional systolic array architecture for all operations expressible as linear recurrences from a representation closely related to the computation graph defined herein. The processors of target abstract machines need not be homogeneous. The computation graph formulation may suggest syntheses of architectures as a collection of special purpose computation units coupled by a problem-dependent topology.

PROGRAMMING WITH COMPUTATION GRAPHS

Computation graphs are expressible in an abstract data type format. The specification follows.

1. Nodes, arcs, and uninterpreted graphs are parameterized primitive types. The properties of these types are described by a set of generic functions such as:

   n:= create-node (name) .. create node n with name "name"

   g:= replace (g',n) .. replace node "n" with graph "g"

2. There are constructed types which result from specification of a context for the execution of the generic functions and specification of type

dependent functions. A context is a specification of a model of parallel computation and a mapping of names to schedulable units of computation, subgraphs or protocols for realization of dependency relations. Constructed types define a class of programs.

3. Each primitive type has an attribute template. The attribute template for a primitive type may vary with the context.

4. It will usually be the case that the target language of the mapping in one context is the source language for the mapping in another context with an increased level of resolution of detail.

   The example in Section 4 of task level data flow formulated in UNIX would have arcs of type data with the attribute "pipes" specifying the protocol for flow of data between the schedulable units of computation.

5. A given graph is an occurrence of some constructed type. Each occurrence is a multi-version object in the definition of Reed [REE83].

Programming with computation graphs (preparation of an occurrence of a constructed type) proceeds as follows:

1. Nodes are created for each schedulable unit of computation at the initial level of abstract machine.

2. The structure of the graph is created by specification of the dependency relations between the schedulable units of computation.

3. The definition of each node and arc is completed by specification of its attribute set.

4. The replacements and type-specific transformation which create the next level of resolution of detail are executed.

The eventual result of this sequence of transformations through constructed types is a program which can be compiled for execution on a realized machine.

ASSESSMENT OF POTENTIAL ADVANTAGES

The abstract machine computation graph representation provides a basis for parallel programming which integrates many parallel programming methodologies and is, at high levels of abstraction, independent of the characteristics of the eventual target architecture. Integration is based on the recognition that the differences in parallel programming methodologies can be isolated to specification of dependency relations between schedulable units of computation. Architectural independence is again based on the clean separation of computations and dependency relations upon its breadth of representation of dependency relations. The differences between parallel architectures are primarily in their realization of dependency relations. Dependency relations are declaratively specified which facilitates separate resolution and compilation. Incremental resolutions of high level designs to executable forms is aided by the fact that each node and each arc in each graph has a specific type of semantics which can be used to define automated transformations to assist in the resolution from abstract to realized machines. These semantics also provide a natural interface to a library of schedulable units of computation. It is a framework for the evaluation of existing programs toward parallel structuring since modules from existing programs can be encapsulated to obtain a first level of schedulable units of execution. The graph basis is a natural interface for visual programming. There are many advantages in this paradigm for parallel programming.

1. It is intrinsically visual and should aid in conceptual formulations.

2. It is a natural structure for hierarchical resolution of computations and thus gives the programmer powerful concepts for managing complexity.

3. It gives a framework for default specification of common properties of objects.

4. It is a framework for definition of meta-operators which act upon the graph to transform it to executable forms.

5. It admits of use of arbitrary levels of granularity of data structures and operators and also different levels of granularity within a given graph.

6. Programs constructed in such a methodology would be extremely portable between parallel architectures since resolution of communication and synchronization operators can be deferred until a final target architecture is selected.

7. The clear identification of computations and the interactions between computations will enhance re-useability of computation units and assist in program validation and maintenance.

Acknowledgements

This research was sponsored in part by the Department of Energy under grant number DE-AS05-81ER 10987, by the Air Force Office of Scientific Research under grant number F49620-83-C-0049 and by the National Science Foundation under grant number DCR-8116099.

References

[ADA68] Adams, D.A., "A Model for Parallel Compilations," in Parallel Processor Systems, Technologies and Applications, pp. 311-334 (Spartan/MacMillan, NY, 1968).

[ALL83] Allen, J.R., Kennedy, K., Porterfield, C. and Warren, J., "A Conversion of Control Dependencies to Data Dependence," Proc. 10th Annual ACM Symp. on Princ. of Prog. Lang., pp. 177-189, Jan. 1983.

[BAR79] Barstow, D.R., "An Experiment in Knowledge-Based Automatic Programming," Artificial Intelligence 12, No. 2, pp. 73-120, (Aug. 1979).

[BER62] Berge, C., Theory of Graphs, (Barnes and Noble, New York, 1962).

[BRO82] Browne, J.C., Tripathi, A., Fedak, S., Adiga, A. and Kapur, R., "A Language for Specification and Programming of Reconfigurable Parallel Computation Structures," Proc. 1982 Into. Conf. on Parallel Processing.

[BRO85] Browne, J.C., "Specification for a Hypergraph Model of Parallel Computation," (preprint -- available from the author).

[CHE84] Chen, M.C., "A Synthesis Method for Systolic Design," Report YALEU/DCS/RR-334, Yale University (Jan. 1984).

[COC72] Cocke, J. and Allen, F.E., "A Catalogue of Optimizing Transformations," in Design and Optimization of Compilers, (R. Rustin, Ed.), pp. 1-30, (Prentice Hall 1972).

[DAV81] Davis, A.L. and Lowder, S.A., "A Sample Management Application Program in a Graphical Data-Driven Programming Language," Digest of Papers, Compcon Spring 81, pp. 162-167 (Feb. 1981).

[DEN72] Dennis, J.B., Fosseen, J.B. and Linderman, J.P., "Dataflow Schemas," in Theoretical Programming, Springer-Verlag, Berlin, pp. 187-216 (1972).

[DEN80] Dennis, J.B., "Data Flow Supercomputers," Computer, Vol. 13, No. 11, pp. 48-56 (Nov. 1980).

[EST63] Estrin, G. and Turn, R., "Automatic Assignment of Computations in a Variable Structure Computer System," IEEE Transactions on Electronic Computers, Vol. EC-12, No. 5, pp. 755-773 (Dec. 1963).

[FER83] Ferrante, J., Ottenstein, K., "A Program Form Based on Data Dependency in Predicate Regions," Proc. 10th Ann. ACM Symp. on Princ. of Prog. Lang., pp. 217-231 (1983).

[GRA81[Gray, J., "The Transaction Concept: Virtues and Limitations," Proc. 7th Int. Conf. on VLDB, pp. 144-154, Sept. 1981.

[JAG84] Jagannathan, R. and Ashcroft, E.A., "Eazyflow: A Hybrid Model for Parallel Processing," Proc. 1984 ICPP, pp. 514-523, August 1984.

[KAN81] Kant, E. and Barstow, D.R., "The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis," IEEE Transactions on Software Engineering, SE-7, No. 5, pp. 458-471 (Sept. 1981).

[KAR66] Karp, R.M. and Miller, R.E., "Properties of a Model for Parallel Computations: Determinacy, Terminations, Queueing," SIAM J. Appl. Math, Vol. 14, No. 6, pp 1390-1411 (Nov. 1966).

[KAR69] Karp, R.M. and Miller, R.E., "Parallel Program Schemata," Journal of Computer and System Sciences, Vol. 3, No. 4, pp. 167-195 (May 1969).

[KEL81] Keller, R.M. and Yen, W-C.J., "A Graphical Approach to Software Development Using Function Graphs," Digest of Papers, Compcon Spring 81, pp. 156-161 (Feb. 1981).

[KEL82] Keller, R.M. and David, A.L., "Data Flow Program Graphs," Computer 13, pp. 26-41 (1982).

[KUC77] Kuck, D.I., "A Survey of Parallel Machine Organization and Programming," Computing Surveys 9, pp. 29-60 (1977).

[KUC81] Kuck, D.I., Kuhn, R.H., Padua, D.A., Leasure, B. and Wolf, M., "Dependency Graphs and Compiler Optimizations," Proc. 8th Ann. ACM Symp. on Princ. of Prog. Lang., pp. 207-218 (1981).

[KEL84] Keller, R.M., Lin F.C.H. and Tanaka, J., "Rediflow Multiprocessing," Proc. IEEE Compcon, pp. 410-417, Feb. 1984.

[MAR66] Martin, D.F., "The Automatic Assignment and Sequencing of Computations on Parallel Processor Systems," Report No. 66-4, Department of Engineering, University of California, Los Angeles (Jan. 1966).

[MAR67] Martin, D.F. and Estrin, G., "Models of Computational Systems - Cyclic to Acyclic Graph Transformations," IEEE Transactions on Computers, Vol. EC-16, pp. 70-79 (Feb.. 1967).

[PET62] Petri, C.A., "Fundamentals of a Theory of Asynchronous Informations Flow," Information Processing, (North Holland, Amsterdam, 1962), pp. 386-391.

[PIN84] Pingali, K. and Arvind, "Efficient Demand-Driven Evaluation (1)," MIT Technical Report MIT/LCS/TM-242.

[REE83] Reed, D.P., "Implementing Atomic Actions on Decentralized D .ta," ACM TOCS 1, pp. 3-23 (1983).

[REI84] Reiss, S.P., "PECAN: Program Development Systems that Support Multiple Views," Proc. of 7th Int. Conf. on Software Engineering (1984).

[ROD69] Rodriguez, J.D., "A Graph Model for Parallel Computation," Technical Report TR-64, Project MAC, M.I.T., Cambridge, MA (1969).

[RUB85] Rubin, R.V., Galin, E.J. and Russ, S.P., "Thinkpal - A Graphical System for Programming by Demonstration - Summary Paper," Proc. 18th HICSS, pp. 115-120 (1985).

[SEJ80] Sejnowski, M.C., Upchurch, E.T., Kapur, R., Charlu, D.P.S. and Lipovski, G.J., "An Overview of the Texas Reconfigurable Array Computer," AFIPS Conf. Proc., pp. 631-641 (May 1980).

[TUR81] Turner, D.A., "The Semantic Elegance of Application Languages," Proc. Conference on Functional Programming Languages, Portsmouth, N.H., Oct., 1981, pp. 85-92.

[WAL82] Walters, R.C., "The Programmers Apprentice: Knowledge-Based Program Editing," IEEETSE 8, pp. 1-11 (1982).

[WOR83] Workman, D.A., "GRASP: A Software Development System using D-Charts," Software-Practice and Experiences 13, pp. 17-32 (1983).

Synthesizing Distributed and Parallel Programs through Optimistic Transformations

Rob Strom and Shaula Yemini

IBM T J Watson Research Center
P O Box 218
Yorktown Heights, NY, 10598

Abstract

We propose a programming methodology for synthesizing efficient distributed and parallel implementations of serial programs by applying correctness preserving program transformations.

We introduce one particular family of such program transformations called *optimistic transformations*. Optimistic transformations allow a logically serial sequence of computations C1; C2 to be executed in parallel whenever C1's effect on C2 can be guessed in advance with high probability. If the guess is wrong, C2 will have to be undone, but if the probability of a correct guess is sufficiently high, the losses due to undoing computations will be compensated by performance gains due to increased parallelism.

We give three examples of "guesses" which lead to optimistic transformations of practical value: (a) the guess that multiple iterations of a loop will not conflict, (b) the guess that exceptional program conditions will not occur, and (c) the guess that machine failures will not occur.

We demonstrate the practicality of our approach by synthesizing a distributed version of a transaction processing program from a serial program to which we apply first a data distribution transformation, and then three optimistic transformations based upon the three guesses illustrated above. The distributed transaction processing program synthesized in this way is shown to be an improved response-time version of the classical distributed two-phase commit protocol.

1.0 Introduction

A programming language presents programmers with a certain computation model. The simpler and more abstract the computation model, the easier it is for programmers to develop correct algorithms and demonstrate that they meet their specifications.

Distributed data, concurrent computations on shared data, and the possibility of hardware failures are all factors which complicate the computation model. As a result, to invent a distributed algorithm and prove it correct is still a difficult art.

We suggest that distributed programs can be designed more effectively by using a more abstract model of computation which presents a "single-system-image". In such a model, programs are composed of modules, each of which is a serial process with only local data. Details of the underlying architecture, such as the number of physical processors, the means of communication between them, the degree of physical concurrency, and the existence of failures, are all hidden.

In order to realize such an abstract model, it is necessary to first *map* the abstract model onto the available physical hardware, and then to *optimize* the resulting implementation by transforming it into a behaviorally equivalent one which takes better advantage of the hardware.

Some typical mappings include allocating logical processes to physical processors, implementing inter-process communication over shared memory or over physical links, masking processor failures by checkpointing onto stable storage, etc.

Some typical optimizations for distributed systems include: replicating data to reduce communication delays, retaining copies of stable storage data in main memory, performing subactions of a serial program in parallel, etc.

Individual mappings and optimizations can be tailored to particular hardware, particular performance objectives, and particular usage patterns. These mappings can be analyzed for both correctness and performance properties independent of programs to which they may eventually be applied. Such mappings could be maintained in a library which would be available to compilers for the high-level language.

Using this approach, programmers specify the logic of their program in the high-level language without incorporating into these programs any decisions about implementation strategies. Programmers then improve efficiency by selecting appropriate mappings and optimization transformations from the library to be applied to the original program. For example, the designer of a distributed operating system would not design a distributed file system, a distributed mail server, a recoverable transaction-system, etc., Instead, he would design a file server, a mail server and a transaction system, assuming a fully reliable "single-system-image". These programs would then be converted to their distributed realizations by the appropriate set of mappings and optimizations.

Our conjecture is that it will be easier to design distributed algorithms by beginning with correct algorithms written in the single-systems-image model and then applying reusable implementation mappings and optimizations, than to complicate the computation model and require programmers to construct distributed algorithms *de novo*.

0190-3918/85/0000/0632$01.00 © 1985 IEEE

In earlier papers, we suggested particular strategies for embedding concurrency control [STR 83] and recovery [STR 84b] below the level of the source programming language, which would then not need to directly control shared data, locking, aborting, or processor assignment. Here, we introduce a *general class* of semantics-preserving program transformations called *optimistic transformations* that increase the degree of concurrency within a distributed system by reducing synchronization.

We first describe the general principles underlying optimistic transformations. We then give examples of several optimistic transformations and apply these transformations to a particular serial program – a database "transaction-processing" program. The original program has only local data (the data of the database), no communication, and is completely serial, while the transformed program, implemented on a network of computers, has both shared and distributed data, internal parallelism, and a communication protocol between sites very similar to the classical "two-phase commit" protocol, but with improved response time.

The example will demonstrate that program transformations, and in particular optimistic transformations, can be used to synthesize known implementation strategies, as well as provide a systematic approach to the development of new distributed systems.

2.0 Optimistic Program Transformations

2.1 Computation model

Our starting point will be a high-level programming language such as NIL [STR 83], which composes large systems from *modules*. Each module executes a serial program, has only local (private) data, and communicates with other modules only by message passing. There is no data sharing. Module boundaries are determined solely on the basis of software engineering principles such as low inter-module coupling, abstraction and information hiding. The language does not allow specifying any performance-related or implementation-dependent decisions since these decisions typically restrict the ability to apply program transformations. Our results apply equally well to other high-level programming languages which forbid data sharing and do not support specifying implementation-dependent and performance-related decisions.

The low-level language into which we map our programs reflects a computation model which includes multiple processor sites, multi-programming at each site, physical links connecting between sites, and stable storage to backup the volatile storage at each site.

To implement a program written in the high-level language, it is first translated into the lower level language and then *optimized* by applying *program transformations*.

Optimistic program transformations convert a set of serially scheduled computations into an equivalent computation in which computations are scheduled in parallel.

2.2 Optimistic schedules

Given a sequence of computations, *an optimistic schedule* allows several logically serial computations to execute *in parallel*. The increased parallelism is obtained by scheduling a computation C_n even before computations C_k, $(k < n)$ *preceding* it in the serial sequence have terminated, whenever it is possible to "guess" the effect of the preceding computations (e.g. their results or the values that they write) with a high probability of correctness.

Provided we maintain the ability to undo the effects of optimistically scheduled computations whenever the corresponding guesses prove incorrect, C_n can be executed in parallel with the preceding computations, by assuming they will have the guessed effect, and undoing when guesses prove incorrect. The optimistic schedule thus preserves the serial semantics, but allows computations which normally could not be scheduled until later to be executed earlier, by eliminating the synchronization involved in waiting for the earlier computations to terminate before scheduling the later computations.

Whether or not optimistic scheduling is an improvement depends upon the probability of successful guessing, the savings when the guess is successful, the costs of undoing computations when the guess is unsuccessful, and the fixed costs of making computations undoable.

2.3 Examples of useful guesses

The following are examples of situations in which we can guess the effect of computations with a high probability of correctness, and therefore optimistic scheduling is likely to improve performance.

1. most likely value: consider the following statement:

   ```
   if p(x) then f(y) end if;
   ```

   If we know that p(x) is almost always true, we may start to execute f(y) even before the computation of p(x) has completed.

2. no-conflict guess: consider the following two statement sequences:

   ```
   Sequence A:          Sequence B:
   A1: a[i] := f(x);    B1: a[i] := f(x);
   A2: a[j] := g(y)     B2: z := f(a[j])
   ```

   Such statement sequences are said to *conflict* whenever concurrent execution (e.g., execution of A1 concurrently with A2) will have different results from execution in the prescribed order. However, when i does not equal j, either of these sequences could be executed concurrently without conflict. Therefore, if we know that the probability of i being equal to j is low, we may choose to execute the two statements of a sequence concurrently.

3. no failures: consider the following statement:

   ```
   y := f(x);
   ```

   If x is in volatile storage (storage lost upon a crash) and y is in stable storage (storage that survives crashes), a crash would cause y to become invalid since the value of x from which it was computed would be lost. One way to make sure this does not happen is to log the value of x before computing y. However, if crashes are very infrequent, we may guess that the value of x will be written to stable storage before the next crash, and thus use the volatile value to compute y, rather than waiting for x to be made stable before computing y.

2.4 Implementing optimistic schedules

Optimistic schedules involve associating predicates called *commit guards* with computations executed on the basis of guesses. A predicate P is called a *commit guard* of an optimistically scheduled computation C if whenever P is true, C will produce the same effect as it would had it been scheduled in its proper sequence. Intuitively, P implies that the guess was successful. Commit guards must be monotonic, i.e., once a commit guard becomes true it