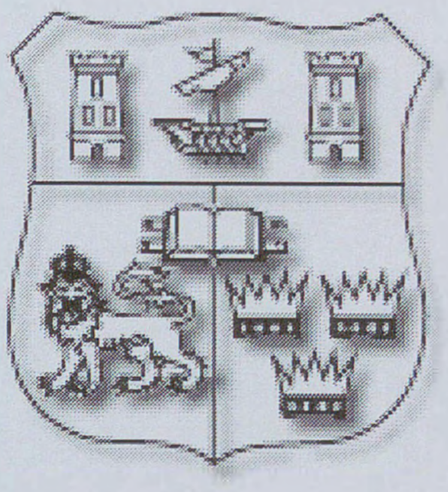
DP 2004 KENN



Design and Implementation of an N-Tier Metacomputer with Decentralised Fault Tolerance

by

James J. Kennedy, M. Sc (John)

THESIS

Presented to the Faculty of Science
National University of Ireland, Cork
for the Degree of

Doctor of Philosophy

May 2004

Abstract

The provision of fault tolerance and fault survival in a large scale dynamic system is complex and heretofore has been the responsibility of the application developer. The exponential increase in complexity associated with present day applications means that this traditional approach is not scalable. In addition, modern dynamic computing platforms, such as the Grid, make it impossible for application programmers to make any static assumptions on resource availability.

The WebCom approach is designed to alleviate the application programmer from this, and other implementation level constraints, by providing an abstract machine behind which these implementation details are hidden. This thesis presents the design and implementation of such a machine and shows how fault tolerance can be handled automatically by the WebCom metacomputer, independently of the application.

Acknowledgements

I am delighted to thank all those who have helped and supported me through the process of writing this thesis. First and foremost, my family - my Mum and Dad for all the support and belief over the years (got there eventually!), my brother Graham for being *such* a good influence on me, and Laura and Elaine for helping me with such vital things as shopping when I needed diversion, and for nodding and smiling when I talked shop for hours on end.

I must also mention some wonderful friends who have been of great help to me - Shane and Joanne, for being great pals and, more importantly, providing hot meals. Dr. Alex Ferguson, for explaining what all those big words meant, and (soon to be Dr.) Sarah Hastings, for the objective criticism and morale boosts when needed. Without all of these people, writing the thing would have been much more of a chore and would have taken *even longer* to finish!

Last, but certainly not least, I must use this space to thank my colleagues in the Centre for Unified Computing. In particular, my supervisor and mentor Dr. John Morrison, whose model made all of our work possible and whose unsurpassed editorial skills made this thesis possible. Also, Dr. David Power, whose career in academia has run parallel with mine for some time now. Thank you to all the great guys in our group who have helped to shape our whole project and continue to drive it forward into uncharted and exciting territory.

Contents

1 Introduction **3**
- 1.1 A Closer Look at Network Computing 4
- 1.2 Fault Tolerance Techniques for Distributed Systems 6
- 1.3 A Model for Modern Computing 9
- 1.4 Motivation and Contribution 14
- 1.5 Overview of Thesis 15

2 WebCom **16**
- 2.1 Minimal Configuration 16
  - 2.1.1 Creating a Module 17
  - 2.1.2 Interfacing with WebCom 17
- 2.2 Abstract Machine 21
  - 2.2.1 Implementing an Engine Module 25
- 2.3 Networking WebCom 27
  - 2.3.1 Distributing Computations 29
  - 2.3.2 Message Passing within WebCom 30
  - 2.3.3 Relaying Messages 32
- 2.4 Efficient Collaboration 32
  - 2.4.1 Redirection Mechanism 33
  - 2.4.2 Topology Management 35
- 2.5 WebCom Means Business 35
  - 2.5.1 Trust Management 38
  - 2.5.2 Economics of Computations 38
  - 2.5.3 Secure Naming 38
- 2.6 Summary 39

3 WebCom Applications **40**
- 3.1 WebCom with Condensed Graphs 40
  - 3.1.1 The Condensed Graphs Engine 40
- 3.2 The CGChess Application 45
- 3.3 Traditional Chess Algorithms 45
  - 3.3.1 Minimax 45
  - 3.3.2 AlphaBeta 48

3.3.3 Aspiration Search 49
3.3.4 Jamboree Algorithm 49
3.4 *CG* Chess 51
3.4.1 *PickMove* 51
3.4.2 *TopJamboree* 51
3.4.3 *Jamboree* 52
3.4.4 *TestMoves* 52
3.4.5 *SerialSearch* 56
3.5 New *CG* Constructs 59
3.5.1 Pass-If-False 59
3.5.2 ForAll 61
3.5.3 Synch Node 61
3.6 Parallelisation Study 64
3.7 Optimisations 66
3.7.1 When things go wrong 67

4 Managing the Unexpected **68**
4.1 Inherent Mechanisms 69
4.1.1 Cost of Survival 73
4.2 Fault Recovery Procedure 74
4.2.1 Unit Replacement 74
4.2.2 Replacing Failed Units 74
4.2.3 Reincorporating Units 79
4.2.4 Incorporating Traditional Checkpointing and Rollback 79
4.3 Adaptive Lineage Management 80
4.3.1 Cost/Benefit Tradeoff 83
4.3.2 Peer to Peer like Topology 84

5 Experiments **86**
5.1 Baseline Case 86
5.2 Faults Tolerance Results 89
5.2.1 Housekeeping Costs 89
5.2.2 Fault Masking 90
5.2.3 Fault Tolerance 90

6 Conclusions and Future Work **96**
6.1 Future Work 99
6.1.1 WebCom for Grid Computing 100

Chapter 1

Introduction

From the first implementation of the Von Neumann computer architecture to modern day global enterprise systems, computing technology has developed in leaps and bounds. Computer processors and peripherals have become increasingly powerful over time as computers have become essential to almost every aspect of day to day life. The PC became an essential business item in the 1980s as the use of business management tools, such as accounts applications and spreadsheets, grew. Simultaneously, the demand for more powerful computers was pushed by the computer games industry as game development progressed towards an ideal of interactive movies. Towards the early 1990s many businesses had begun to use *Local Area Networks*(LANS) and *Wide Area Networks*(WANS) to share data internally, such as between departments and already the benefits of distributed resources had become apparent.

By this time, the global Internet had grown from a means to share information between a relatively small number of mainly educational institutes. The *Hyper Text Transfer Protocol*(HTTP) and the World Wide Web grew in popularity and many businesses thrived on providing access to the internet to home users and allowing them to place their own web sites online. Retailers began to provide online shopping for consumers. In the mid 1990s the Internet gained huge global growth momentum and began to become as pervasive as computers themselves.

As time went on, an increasing number of machines and users became connected to the Internet and institutions used internet technologies such as HTTP on their own internal *intranets*. With so many computing resources now linked to each other and communicating with standardised protocols, it naturally followed that the utility of these resources could be increased if they were to work together. While many researchers had been investigating the field of network computing over the previous decades, industry demand over the last decade has led to a proliferation of abstractions, complexity and variation in distributed systems.

1.1 A Closer Look at Network Computing

In this section, the evolution of various concurrent computing paradigms is presented. The discussion begins with simple parallel machines and follows the evolution of parallel computing towards the modern field of grid computing.

Parallel Computing

Parallel computing has been around for many decades and is based on the simple premise that multiple processors should complete a job quicker than a single processor. This led to the idea of parallel computers, single machines with multiple processors. These are popularly categorised into *Single Instruction, Multiple Data*(SIMD)[7] and *Multiple Instructions, Multiple Data*(MIMD)[18] machines. A SIMD architecture consists of a single CPU and many "sub-processors" which all execute the same task on different sets of data. The CPU divides the data into portions that are send out to the sub-processors for parallel execution and then processes the returned results. A MIMD computer consists of a number of independent and equal CPUs; it is more flexible and suits parallel tasks where different types of computations may be performed simultaneously. Many MIMD machines are *Shared Memory Processors*(SMP)[31] where a single memory address set is shared amongst the CPUs for speed and ease of communication between tasks. Applications for these parallel machines are generally tailored specifically for the machine itself and an iterative programming cycle is often employed to "tweak" the application for best performance. This approach suits a task which, after its development, will be run many times. There have been many developments in the design of parallel machines, for example, the number of physical links between processors in a machine is limited by practical restrictions on how many physical links can exist in the hardware. With this in mind, various interconnection topologies between processors have been investigated and most have been found to be optimal for particular types of applications[14].

Distributed Computing

In parallel computing in general, there is a tradeoff between the number of processors cooperating on a task and the communications overhead that introducing more processors brings. The parallel machines discussed so far benefit from having very fast links between processors as they usually share the same board or bus, and thus the communications overhead is minimised. A development of the parallel computer idea, known as *Distributed Computing*[64, 12, 8] has become popular where separate machines work together, usually over slower and less reliable links such as ethernet. A common example of this is *Networks of Workstations*(NOWs) where machines on the

same network act together in a parallel fashion. Programming APIs such as PVM and MPI facilitate the message passing requirements in distributed computing applications, or network protocols are often hardcoded into the application itself. To construct an application for a NOW, the programmer must, therefore, encode a lot of overhead logic to manage the network. This logic must perform such tasks as monitoring available machines, balancing the load on the machines to optimise execution time, dealing with breakdown of machines or links between machines[40, 22]. In addition, the programmer must specify how the parallel task is to be divided amongst processors and manage the reincorporation of subtask results. NOWs provide parallelism at a reasonably cheap hardware cost when compared to purchasing parallel machines with large numbers of processors. In fact, many organisations will usually have existing NOWs that can be utilised for parallel tasks. A common goal of distributed computing systems, therefore, is to allow the harnessing of preexisting underutilised hardware or *cycle stealing*[57, 37].

Metacomputing

In an attempt to make distributed computing easier to use, the field of *Metacomputing* has been developed. Commonly, the goal of a metacomputing system is to allow the application developer to view a collection of machines, such as a NOW or a set of machines connected over a WAN or the internet, as a single computing platform[61, 5, 50, 21]. A metacomputing system attempts to automate network management and hide as much of it as possible from the programmer. There have been a number of different metacomputing platforms such as Charlotte[6, 30], Bayinihan[58, 39], Javelin[15] and WebCom[49, 50, 47, 62]. Most metacomputers are written in Java to take advantage of platform independence, allowing the metacomputer to transparently employ any hardware type that supports a Java Virtual Machine(JVM). Usually, a metacomputer requires that the programmer only specify which parts of a task may be executed in parallel and how to divide the operand data and process the returned results. At this point, metacomputer development reflects that of original parallel processing. Some metacomputers are SIMD type metacomputers where there is a single master and multiple slaves that execute the same task on different data, e.g. Charlotte. Others, such as Bayinihan, are MIMD type metacomputers where different slaves may perform different tasks. The computing system presented here, WebCom, is also a metacomputer. The original prototype for WebCom used Java Applets[41] to act as volunteer workers for a single master. This provided a MIMD 2-tier metacomputing platform, similar to Charlotte. It will be shown that the WebCom implementation discussed here has been developed far beyond that model[45, 46, 27].

Peer to Peer Computing

Peer to peer(P2P) computing[23, 9, 19] is another recent phenomenon, motivated by the increased network connectivity of otherwise unrelated machines. In a P2P environment, many machines may work together, usually *Volunteer Based*, but all machines are considered equal in every way – true peers. This contrasts with the traditional meta- and distributed computing paradigm where some machines are considered "workers" for other machines.

P2P systems tend to be organic and unpredictable, suiting environments where machines may choose to join or leave the computation at any stage or where machines wish to interact without a "middle man". This anonymous interaction has led to P2P filesharing becoming so popular that it threatens traditional media distribution industries.

Grid Computing

Grid computing is a natural successor to distributed computing. Grid computing[1, 2, 60] attempts to make distributed computing available on a global scale where many clusters or networks are connected by high speed links. In this way the grid may be seen to be comparable to the Internet or simple distributed computing. However, the grid attempts to address practical organisational issues and the commercial feasibility of such a large scale distributed computing environment. It provides a means to organise resources into *Virtual Organisations*(VOs) with well defined boundaries and access mechanisms. VOs are connected to the grid via gateway machines that may offer services from their underlying network. Currently the most common service offered on the grid is the use of local high performance computing clusters. It is envisaged that the grid will eventually enable true integration between businesses, allowing them to avail of each others' services in a standardised, reliable fashion. Organisations on the grid offer actual business services to each other, such as human expertise. A Grid application can then harness these services to construct applications incorporating goods and services from various expert domains and geographical locations.

1.2 Fault Tolerance Techniques for Distributed Systems

The growth in complexity of the distributed systems described above has led to many challenges, not least of which is fault tolerance. As the number of components in a system increases, so too does the chance of one or more components failing. As a result, many techniques have been developed to overcome these faults and increase the reliability of distributed systems. Additionally, many distributed systems are facilitating long running

applications which might traditionally have been executed on supercomputers, e.g., numerical applications working with huge data sets such as ocean flow analyses[35]. The duration of the application is another factor in the probability of a fault occurring. The following survey focusses on software techniques in distributed systems. Hardware specific techniques are not considered and the *fail-stop* model (in which a failed processor stops and loses any unsaved state information) is assumed.

Transactional Programming *Transactional programming*[13] was originally developed for database applications. A transaction system operates through a number of discreet, atomic, and reversible transactions. Transactions between components are logged during execution and, in the event of a failure, they can be *undone* – effectively rewinding the overall execution to a point before the failure occurred. Execution can then be resumed after the faulty component has been replaced or removed from the system. Transactional programming is still mainly used for data oriented applications since operations in these applications tend to be easily reversible.

Rollback Recovery with Uncoordinated Checkpointing Another prevalent fault tolerance technique is *rollback recovery*[25, 66]. Processes involved in the distributed system periodically save their computational state to some form of stable storage. This state is referred to as a *checkpoint.* Should a fault occur, a failed process is restarted from a previous checkpoint, i.e., the computation is *rolled back.* Using this technique, all work prior to the checkpoint is recovered. This is of great benefit to long running applications where the use of a checkpoint may eliminate large amounts of recomputation.

Metacomputing systems tend to be based on a system of message passing (not necessarily transactional) between often heterogenous processors. These messages therefore cause dependencies between processes in the system. These dependencies can complicate the rollback recovery procedure by causing what is known as *rollback propagation.* For example, if one process(P) fails, it rolls back to a previous, fault free, state(S). This requires any other processes that received messages from P *after the point S was saved at* to rollback to their checkpoints from before the messages were received. The rollback of these other processes can cause more rollback and this domino effect may repeat until the distributed computation is rolled back to its beginning.

Coordinated Checkpointing The domino effect can be avoided through the use of *coordinated checkpointing*[4], in which processes coordinate with each other to periodically store a system wide checkpoint. This checkpoint then becomes the earliest state to which the execution may be required to

roll back. Coordination ensures that the domino effect is eliminated and simplifies the recovery procedure. It also prevents any redundant checkpoints from being saved, i.e., checkpoints that are not required for the global state. A disadvantage is that individual processes have less autonomy, they lose the ability to save a checkpoint when it suits them and every process must participate in each checkpoint, thereby reducing scalability. A coordinated checkpoint may be blocking or, more commonly, use a non-blocking approach which greatly reduces the cost of a checkpoint. The scalability issue of coordinated checkpointing is somewhat addressed by introducing *Minimal Checkpointing*[34] in which only those processes that have received a non-deterministic event participate.

Communication Induced Checkpointing A variation of the coordinated checkpointing technique is *communication induced*[56] checkpointing. In this technique, processes tell each other when to checkpoint. Systems using this induced checkpointing are required to ensure that there is always a consistent state for the whole computation stored at all times during the execution. Processes retain a degree of autonomy by having two types of checkpoint, forced and local. A process periodically makes local checkpoints itself when it is not costly to do so (e.g. if temporarily idle). It may also be forced to make a checkpoint due to a communication from another processor. These forced checkpoints represent the earliest consistent global state but during recovery any more recent local checkpoints are incorporated instead, to provide a more advanced state.

Log Based Rollback The benefits of rolling back from a checkpoint can be increased through the use of *log based*[16] rollback recovery, which incorporates the notion of transactional programming into the fault recovery. To use log based recovery, each nondeterministic event is logged. The recovery procedure relies on the *piecewise deterministic*(PWD) assumption[65], i.e., it must assume that all nondeterministic events can be identified and logged. The replaying of these logs can then be used to rebuild a computation from its beginning, if necessary. This technique allows the system to reduce the frequency of checkpoints with a small increase fault recovery time. This tradeoff will often be preferable since a complete checkpoint on a single processor may be very costly in terms of time and storage space used. This is especially true when many non-deterministic events occur during execution and log based rollback helps to alleviate the *output commit* problem[65] (the problem of ensuring that the state from which an output from the system is sent can always be recovered, since entities outside the system cannot be rolled back).

Log based rollback protocols can be classified as optimistic, pessimistic or causal. A pessimistic log based protocol logs every nondeterministic event,

guaranteeing that all processes can be reproduced. The logging operation must be complete before the event is processed, reducing runtime performance. This makes recovery in the event of a fault very simple but can be expensive during failure free execution.

Optimistic log based protocols accept the risk of losing work, allowing nondeterministic events to be processed whilst they are being logged. While this increases the runtime performance of the system, it also introduces the chance of rollback propagation if a process fails before logging an event it has already acted on. This chance of rollback propagation increases storage costs and the complexity of garbage collection as multiple checkpoints must be maintained on each processor.

More recently, various causal based logging techniques have been proposed[51, 68], mainly in conjunction with *distributed shared memory*(DSM)[63, 43] machines, which combine the advantages of both pessimistic and optimistic log based rollback. When a processor sends a message, it includes with the message the deterministic events that led to that message, using the *happened before*[36] relation, and the receiver logs these events with the message. In this way, each processor knows the events that need to be replayed by other processors to rebuild any part of the computation it is involved in. The runtime overhead of this logging may be large, but it can often be reduced by linking lists of determinants with already logged lists to avoid logging the same sequence multiple times.

1.3 A Model for Modern Computing

The WebCom framework was designed to primarily operate using the Condensed Graphs ($\mathcal{CG}$) model of computation. However, the physical implementation of the computational model forms a modular component capable of being readily exchanged for any other computational model. The $\mathcal{CG}$ model is preferred because it offers novel solutions to problems such as dynamic load balancing and fault tolerance. The latter two are dealt with in Chapters 2 and 4 respectively.

To familiarise the reader with the concepts of the $\mathcal{CG}$ model, the following description has been taken verbatim from [47]:

"While being conceptually as simple as classical dataflow schemes [24, 3], the Condensed Graphs ($\mathcal{CG}$) model is far more general and powerful. It can be described concisely, although not completely, by comparison. Classical dataflow is based on data dependency graphs in which nodes represent operators and edges are data paths which convey simple data values between them. Data arrive at operand ports *of nodes along input edges and so trigger the execution of the associated operator (in dataflow parlance, they cause the node to* fire*). During execution, these data are consumed and a resultant datum is produced on the node's outgoing edges. This result acts*

as input to successor nodes. In cyclic dataflow graphs, nodes may fire more than once and so operands belonging to different executions are distinguished from each other using labels. When data with the same label are present on every operand port, a complete operand set *is formed. Operand sets are used as the basis of the firing rules in data-driven systems. These rules may be* strict *or* non-strict*. A strict firing rule requires a complete operand set to exist before a node can fire; a non-strict firing rule triggers execution as soon as a specific proper subset of the operand set is formed. The latter rule gives rise to more parallelism but also can result in overhead due to remaining packet garbage (RPG).*

Like classical dataflow, the CG model is graph-based and uses the flow of entities on arcs to trigger execution. In contrast, CGs are directed acyclic graphs in which every node contains not only operand ports, but also an operator and a destination port. Arcs incident on these respective ports carry other CGs representing operands, operators and destinations. Condensed Graphs are so called because their nodes may be condensations, or abstractions, of other CGs. (Condensation is a concept used by graph theoreticians for exposing meta-level information from a graph by partitioning its vertex set, defining each subset of the partition to be a node in the condensation, and by connecting those nodes according to a well-defined rule [20].) Condensed Graphs can thus be represented by a single node (called a condensed node*) in a graph at a higher level of abstraction. The number of possible abstraction levels derivable from a specific graph depends on the number of nodes in that graph and the partitions chosen for each condensation. Each graph in this sequence of condensations represents the same information but in a different level of abstraction. It is possible to navigate between these abstraction levels, moving from the specific to the abstract through condensation, and from the abstract to the specific through a complementary process called* evaporation.

The basis of the CG firing rule is the presence of a CG in every port of a node. That is, a CG representing an operand is associated with every operand port, an operator CG with the operator port and a destination CG with the destination port. This way, the three essential ingredients of an instruction are brought together (these ingredients are also present in the dataflow model; only there, the operator and destination are statically part of the graph).

*A condensed node, a node representing a datum, and a multi-node CG can all be operands. A node represents a datum with the value on the op*erator *port of the node. Data are then considered as zero-arity operators. Datum nodes represent graphs which cannot be evaluated further and so are said to be in* normal form*. Condensed node operands represent unevaluated expressions. They cannot be fired since they lack a destination. Similarly, multi-node CG operands represent partially evaluated expressions. The processing of condensed node and multi-node operands is discussed below.*

Any CG may represent an operator. It may be a condensed node, a node whose operator port is associated with a machine primitive (or a sequence of machine primitives) or it may be a multi-node CG.

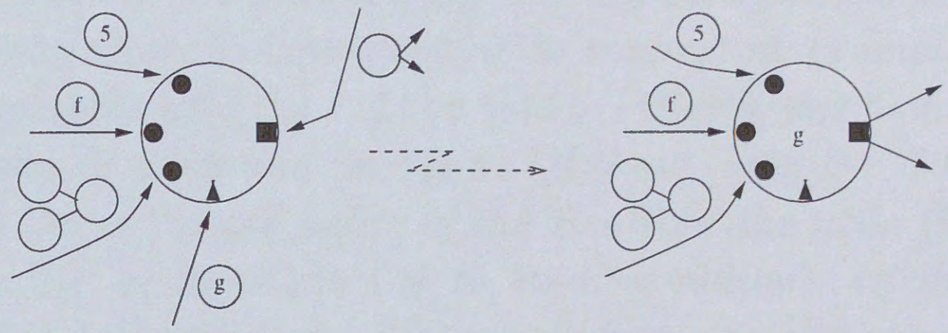


Figure 1.1: *CGs congregating at a node to form an instruction*

The present representation of a destination in the CG model is as a node whose own destination port is associated with one or more port identifications. The expressiveness of the CG model can be increased by allowing any CG to be a destination but this is not considered further here. Fig. 1.1 illustrates the congregation of instruction elements at a node and the resultant rewriting that takes place.

When a CG is associated with every port of a node it can be fired. Even though the CG firing rule takes account of the presence of operands, operators and destinations, it is conceptually as simple as the dataflow rule. Requiring that the node contain a CG in every port before firing prevents the production of RPG. As outlined below, this does not preclude the use of non-strict operators or limit parallelism.

A grafting *process is employed to ensure that operands are in the appropriate form for the operator: non-strict operators will readily accept condensed or multi-node CGs as input to their non-strict operands. Strict operators require all operands to be data. Operator strictness can be used to determine the strictness of operand ports: a strict port must contain a datum CG before execution can proceed, a non-strict port may contain any CG. If, by computation, a condensed or multi-node CG attempts to flow to a strict operand port, the* grafting *process intervenes to construct a destination CG representing that strict port and sends it to the operand.*

The grafting process thus facilitates the evaluation of the operand by supplying it with a destination and, in a well constructed graph, the subsequent evaluation of that operand will result in the production of a CG in the appropriate form for the operator. The grafting process, in conjunction with port strictness, ensures that operands are only evaluated when needed. An inverse process called stemming *removes destinations from a node to prevent it from firing.*

Strict operands are consumed in an instruction execution but non-strict operands may be either consumed or propagated. The CG operators can be divided into two categories: those that are 'value-transforming" and those that only move CGs from one node to another in a well-defined manner.

Value-transforming operators are intimately connected with the underlying machine and can range from simple arithmetic operations to the invocation of sequential subroutines and may even include specialised operations like matrix multiplication. In contrast, CG moving instructions are few in number and are architecture independent. Two interesting examples are the condensed node operator and the `filter` *node. Filter nodes have three operand ports: a Boolean, a* `then` *and an* `else`*. Of these, only the Boolean is strict. Depending on the computed value of the Boolean, the node fires to send either the* `then` *CG or the* `else` *CG to its destination. In the process, the other operand is consumed and disappears from the computation. This action can greatly reduce the amount of work that needs to be performed in a computation if the consumed operands represent an unevaluated or partially evaluated expression. All condensed node operators are non-strict in all operands and fire to propagate all their operands to appropriate destinations in their associated graph. This action may result in condensed node operands (representing unevaluated expressions) being copied to many different parts of the computation. If one of these copies is evaluated by grafting, the graph corresponding to the condensed operand will be invoked to produce a result. This result is held local to the graph and returned in response to the grafting of the other copies. This mechanism is reminiscent of parallel graph reduction [54] but is not restricted to a purely lazy framework.*

CGs which evaluate their operands and operator in parallel can easily be constructed by introducing `spec` *(speculation) nodes to act as destinations for each operand. The spec node has a single operand port which is strict. The multi-node CG operand containing the spec node is treated by non-strict operand ports in the same way as every other CG, however, if it is associated with a strict port, the spec node's operand is simply transferred to that port. If that operand had already been fully evaluated, it could be used directly in the strict port, otherwise, it is grafted onto the strict port as described above. This is illustrated in Fig. 1.2.*

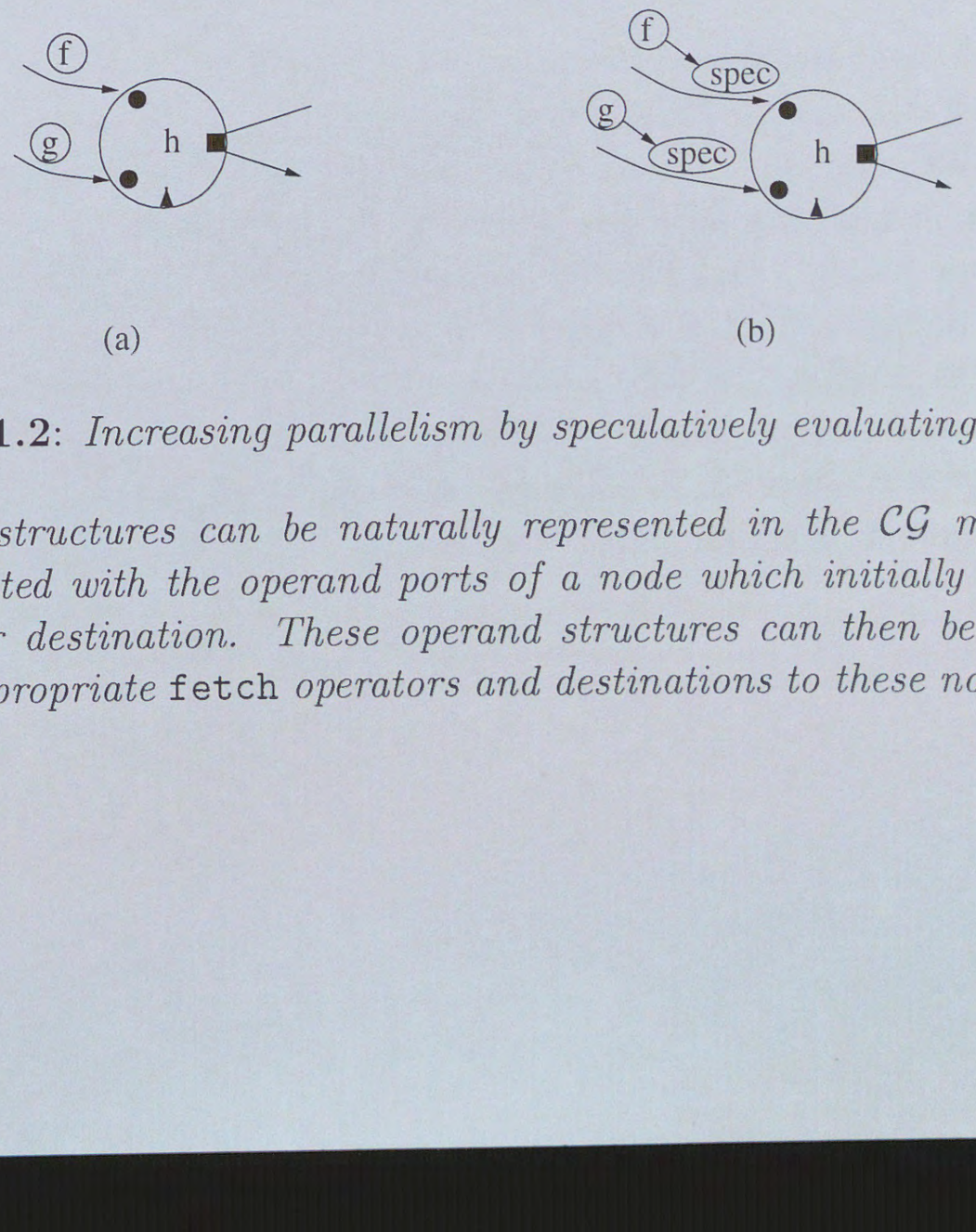


Figure 1.2: *Increasing parallelism by speculatively evaluating operands*

Stored structures can be naturally represented in the CG model. CGs are associated with the operand ports of a node which initially contain no operator or destination. These operand structures can then be fetched by sending appropriate `fetch` *operators and destinations to these nodes. These*

fetch operators also form part of the CG moving operators and so are machine independent.

The power of the operators in the CG model can be greatly enhanced by, associating with each, specific deconstruction semantics. *These specify if CGs can be removed from the ports of a node after firing. In general, every node will be deconstructed to remove its destination after firing. This renders a node incomplete and prevents it from being erroneously refired. The deconstruction semantics of the fetch operator cause the operator and the destination to be removed after firing. This leaves the stored structure in its original state, ready for a subsequent fetch. Fig. 1.3 illustrates the process of fetching a stored structure.*

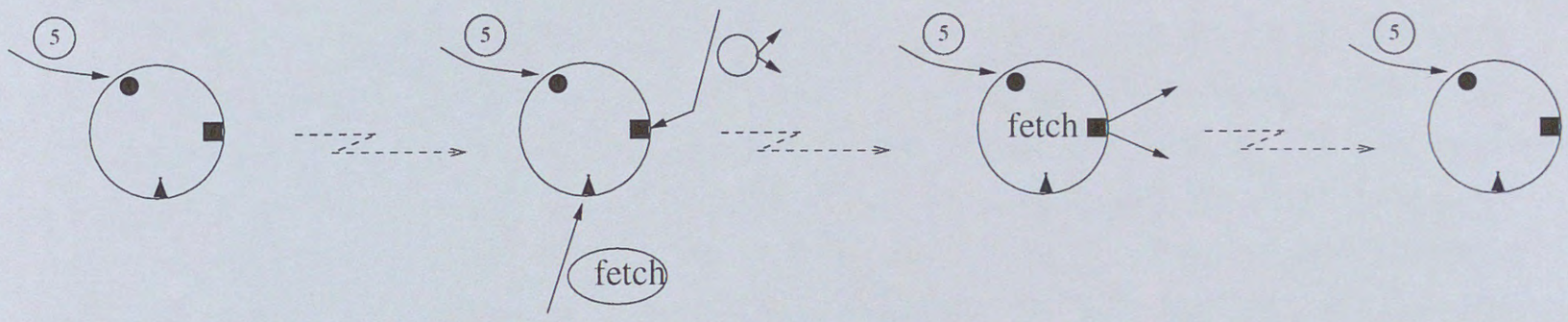


Figure 1.3: *Sequence of events showing the fetching of a stored structure and subsequent node deconstruction*

The CG moving instructions are extended to include assignment which, with appropriate deconstruction semantics, implement mutable structures. Side-effect computations are thus encapsulated in graphs representing mutable, stored structures. Condensed nodes representing these structures may flow on graphs linearised to ensure that the side-effects are carried out in the correct order. Those parts of the computation not affected by the side-effects can proceed as normal, in parallel. This can be likened to the monadic approach for introducing side-effects into pure, non-strict languages: where the monad structure[44] is used to encapsulate the side-effects and ensure correct sequencing[28].

By statically constructing a CG to contain operators and destinations, the flow of operand CGs sequences the computation in a dataflow manner. Similarly, constructing a CG to statically contain operands and operators, the flow of destination CGs will drive the computation in a demand-driven manner. Finally, by constructing CGs to statically contain operands and destinations, the flow of operators will result in a control-driven evaluation. This latter evaluation order, in conjunction with side-effects, is used to implement imperative semantics. The power of the CG model results from being able to exploit all of these evaluation strategies in the same computation, and dynamically move between them, using a single, uniform, formalism."

1.4 Motivation and Contribution

Condensed Graphs provide an exciting model for distributed computing. To investigate this model, the WebCom metacomputer presented here was developed. WebCom utilises features of Condensed Graphs to provide a novel distributed computing environment with a number of distinguishing features. These include the abstraction of application development from the underlying hardware, thus allowing solutions to be expressed independently of implementation. The application developer is not required to specify where tasks execute or how many machines are employed, since WebCom provides automatic exploitation of inherent parallelism in the application graph.

WebCom is distinguished by its dynamic topology management in which machines can be redirected to connect to others on the network, allowing complex interconnection topologies to evolve. Apart from being a useful tool to facilitate richer load balancing strategies, this mechanism leads to an n-tier topology which is particularly vulnerable to faults, as large portions of the execution platform can become isolated by the failure of a single machine. Problems of this kind are very typical and seen as major issues to be resolved before the proper advent of autonomic computing[67]. Without proper mechanisms, it is estimated that the cost of fault recovery in a general n-tier system can amount to doubling the computing effort for every fault. A novel and autonomic fault tolerance routine, which addresses this problem for WebCom, is presented here as part of the WebCom metacomputer. The basic idea behind this work has also been previously published by the author in[48, 49, 47, 50, 45].

Apart from the design and implementation of novel, autonomic fault tolerance mechanisms, the work of the author includes a significant engineering contribution to the realisation of the WebCom metacomputer, i.e., in the specification, design and implementation of the WebCom architecture. To adequately test the fault tolerance procedures expounded in this thesis, the author also embarked on the creation of the hitherto largest Condensed Graphs application. This research exercised many ideas for the first time and was important for refining large scale Condensed Graph executions. The extent of the engineering effort is reflected in the following statistics: the WebCom metacomputer is comprised of over nine thousand lines of Java code, distributed across ninety classes. The $\mathcal{CG}$ Chess application is comprised over nineteen hundred lines of code in thirty eight classes. A Java implementation of the Condensed Graphs model was co-implemented by the author and amounts to over four thousand lines of Java code in twenty seven classes.

1.5 Overview of Thesis

This thesis presents the design and implementation of WebCom and an associated fault tolerance routine to address WebCom's unique requirements. Chapter 2 describes the components of a single instance of WebCom, a Unit, and how multiple units interact. WebCom utilises networks of units to perform distributed computations. WebCom's components are described in detail, beginning with the core Backplane component. Each of the other modules and their contribution to the overall system are then discussed. An Engine Module performs actual task execution and various subclasses which utilise different underlying technologies are discussed. A Communications Manager allows WebCom units located on different machines to collaborate on computations. A Load Balancing Module manages the distributed computation and a Fault Tolerance Module reacts to unit failures. A Security Module provides a trust management based security layer.

Chapter 3 presents in detail the default WebCom Engine Module, the Condensed Graphs engine. Some sample graphs are discussed and the development of a large scale application to stress test WebCom is presented. This application, $\mathcal{CG}$ Chess, is a Condensed Graphs implementation of the Jamboree algorithm for choosing the next move in a game of chess. The $\mathcal{CG}$ Chess application provides enough parallelism to test WebCom's topology management.

WebCom's ability to deal with faults in its dynamic topology are discussed in Chapter 4. A Unit replacement algorithm is presented and it is then extended to allow the replacement of a failed unit and the reincorporation of existing partial results. Variations of the routine to deal with more complex fault scenarios are then discussed.

Chapter 5 presents the results of a number of experiments performed. The $\mathcal{CG}$ Chess application is used to test WebCom's runtime performance. The results of these tests are used as a baseline to measure the cost of fault tolerance housekeeping and the performance of the fault tolerance routine itself. The ability of WebCom's fault tolerance to limit the impact of a fault is measured using these results.

Chapter 6 presents conclusions derived from the development of WebCom and its fault tolerance routine. The implications of the experimental results are discussed. Some ongoing development work is then discussed and a number of additions to WebCom's architecture are proposed. Finally, the chapter presents the future of WebCom as it progresses towards a system for Grid computing.

Chapter 2

WebCom

WebCom is a flexible distributed computing platform, the functionality of which can be dynamically extended by plugging modules into a core *Backplane* component. This philosophy makes WebCom an ideal development platform for experimenting with various distributed computing components. The independence offered by the modular design means that those components under development may be "plugged" in and out of a WebCom *unit* without affecting the operation of the rest of the system.

Many of the pleasing characteristics of WebCom follow directly from its default compute engine. An implementation of the Condensed Graphs model, it allows for strong locality in the execution, for referential transparency, for hierarchical execution and for inherent storage of execution state.

2.1 Minimal Configuration

The minimal WebCom configuration consists of a single component known as the *Backplane*. The Backplane provides a mechanism for passing work and results to and from WebCom and has an associated user interface through which it may be instantiated. Application interface hooks provided by the Backplane allow other third party applications, such as enterprise applications or daemons, to replace the associated user interface. These applications may then generate work for WebCom and consume the associated results. These applications will be referred to here as *Generator/Consumer*(GenCon) applications. Fig. 2.1 depicts the relationship between the GenCon and the Backplane.

At startup, the Backplane may load WebCom modules according to a configuration file that specifies Java class names for each module. However, at runtime the Backplane may dynamically load and unload modules as required. In addition to this management of modules, the Backplane's functionality also includes facilitating communication between modules. This is

accomplished via a dedicated WebCom message passing system.

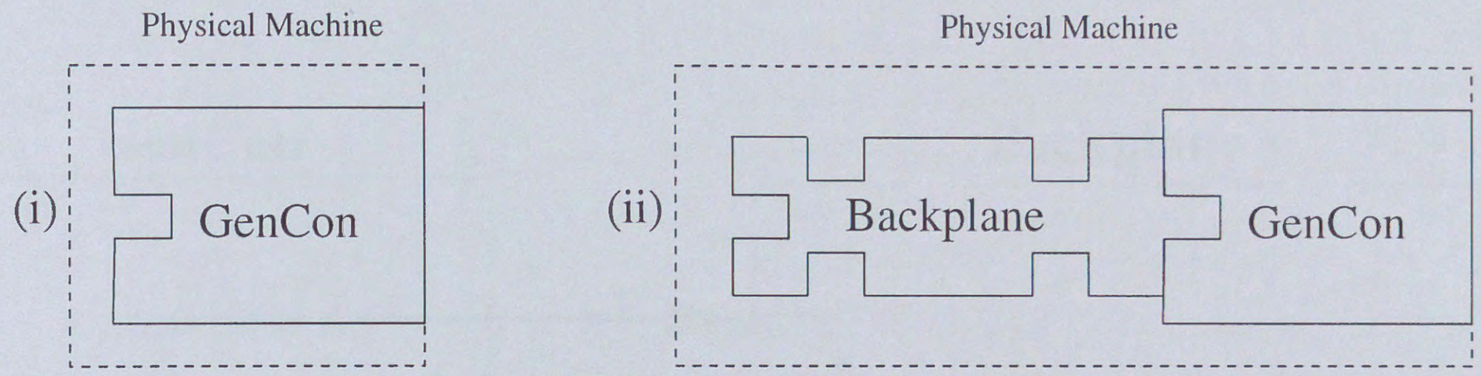


Figure 2.1: *(i) Single machine running a GUI or application(GenCon) with no WebCom configuration. (ii) The GenCon instantiates a minimal WebCom configuration by creating a* ***Backplane****. In this instance, the Backplane loads no other modules.*

2.1.1 Creating a Module

To create a WebCom Module, a Java class must implement WebCom's `Module` interface, shown in Code 2.1. The `processMessage()` method must be implemented to allow the Module to consume WebCom Message objects and the remaining methods require the Module to maintain a reference to the Backplane.

```
public interface Module
{
    public boolean processMessage(Message m);
    public void setBackplane(Backplane s);
    public Backplane getBackplane();
}
```

Code 2.1: *The Module Interface. All WebCom Module objects must implement these methods.*

2.1.2 Interfacing with WebCom

A GenCon application using WebCom must include a component object which implements the WebCom's `Resultee` Java Interface. This object contains an implementation for the method `receiveResult()`.

To utilise WebCom the GenCon application instantiates a *`Backplane`* object. It is through this Backplane that the GenCon will interact with WebCom. Fig. 2.2 depicts a minimal abstract machine in which a GenCon application instantiates a Backplane object.

To execute a task on WebCom, the GenCon must first wrap it in a WebCom `Instruction` object. This object contains the operand data for

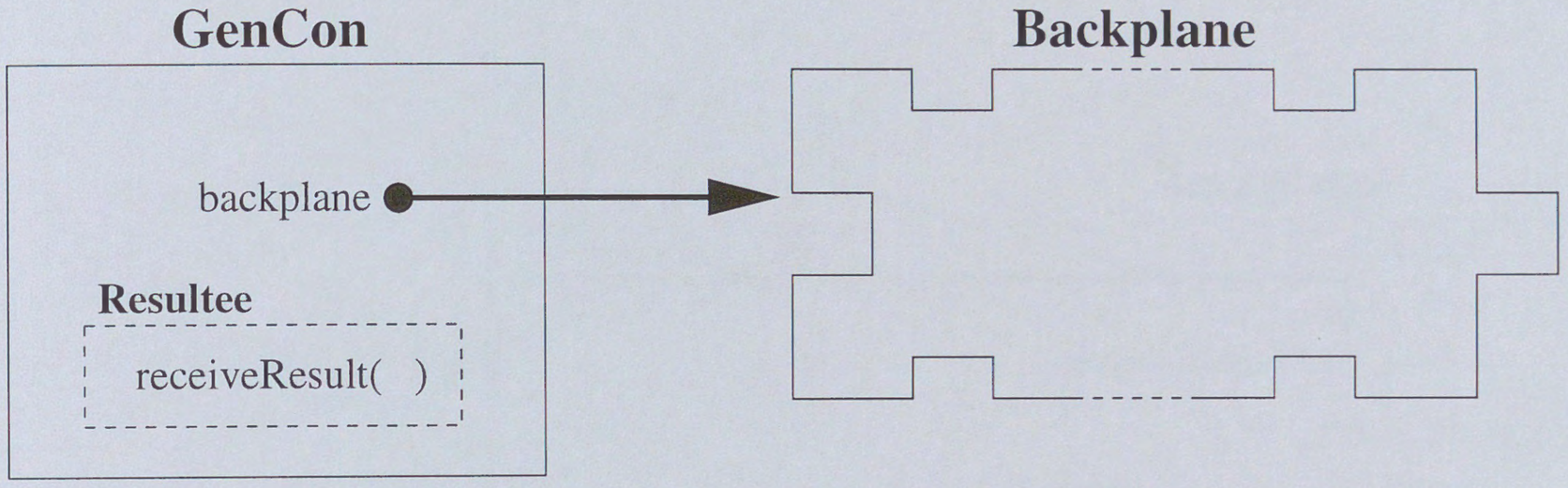


Figure 2.2: *A Generator/Consumer application (with a component* `Resultee` *object) instantiates a Backplane object and maintains a reference to it.*

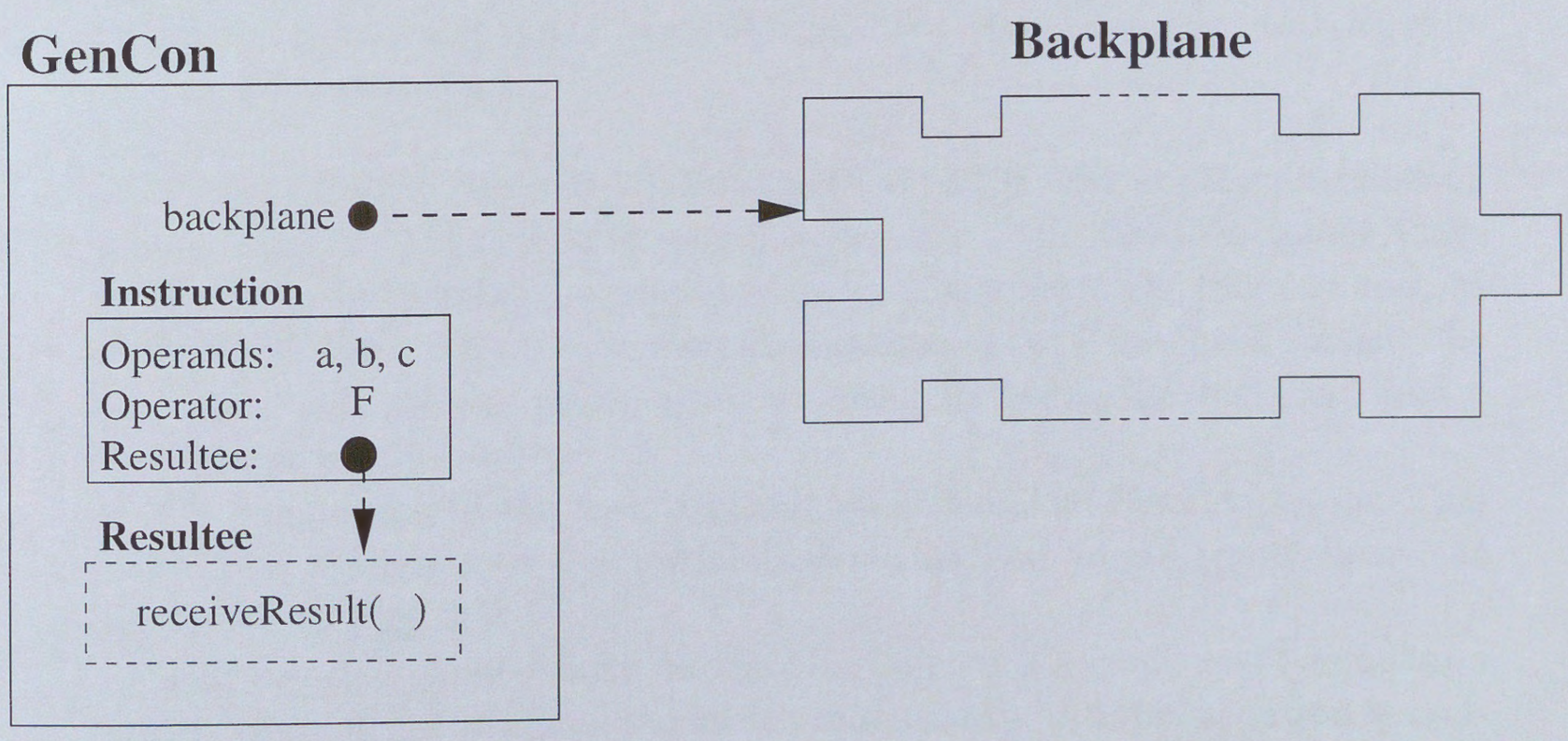


Figure 2.3: *The GenCon creates an* `Instruction` *object containing operand data, an operation and a* `Resultee` *reference.*

the task, the operator for the task, and a reference to a Resultee object which will eventually receive the result of the task. Fig. 2.3 depicts the GenCon after it has instantiated an Instruction.

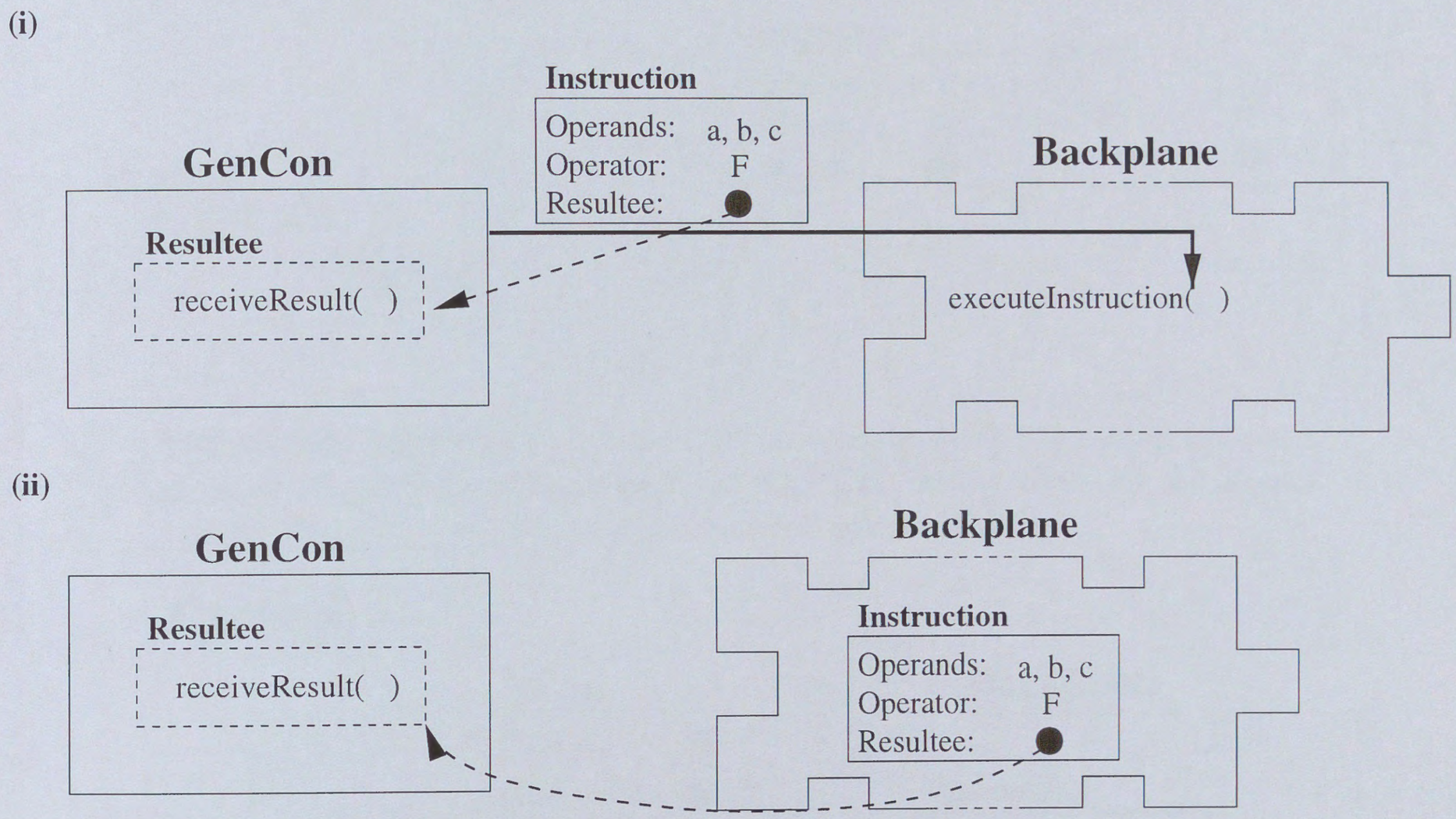


Figure 2.4: *(i) The GenCon passes an Instruction to the Backplane via its* `executeInstruction()` *method. (ii) The Backplane can now begin to execute the instruction.*

The Backplane contains a public method called `executeInstruction()` which takes an `Instruction` as its argument. The GenCon issues tasks to WebCom by passing Instructions to the Backplane via this method, as depicted in Fig. 2.4). Once `executeInstruction()` has been called, the Backplane has all the information required to begin the task and has a destination for the result.

On completion of the task it generates a WebCom `Result` object. This contains a reference to the initial Instruction and to its result value, as illustrated in Fig. 2.5.

The result is passed back to the GenCon via its component `Resultee` object where it is consumed via `receiveResult()` and its execution is considered complete.

As shown in Fig. 2.6, the `Resultee` reference is used to pass the Result back to the GenCon. The GenCon's Resultee consumes the result and the execution is considered complete (Fig. 2.7).

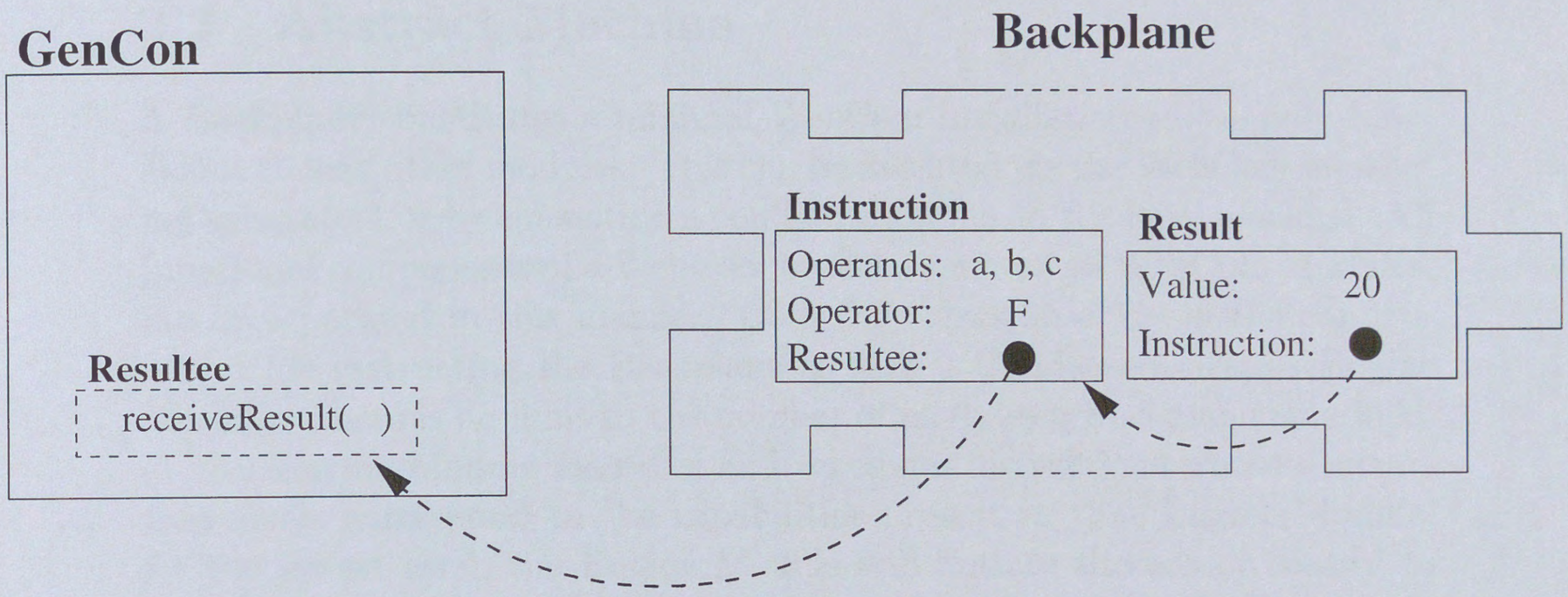


Figure 2.5: *WebCom has completed the execution of the Instruction and generated a corresponding* `Result` *object. This object contains the actual result value and a reference to its "source" instruction.*

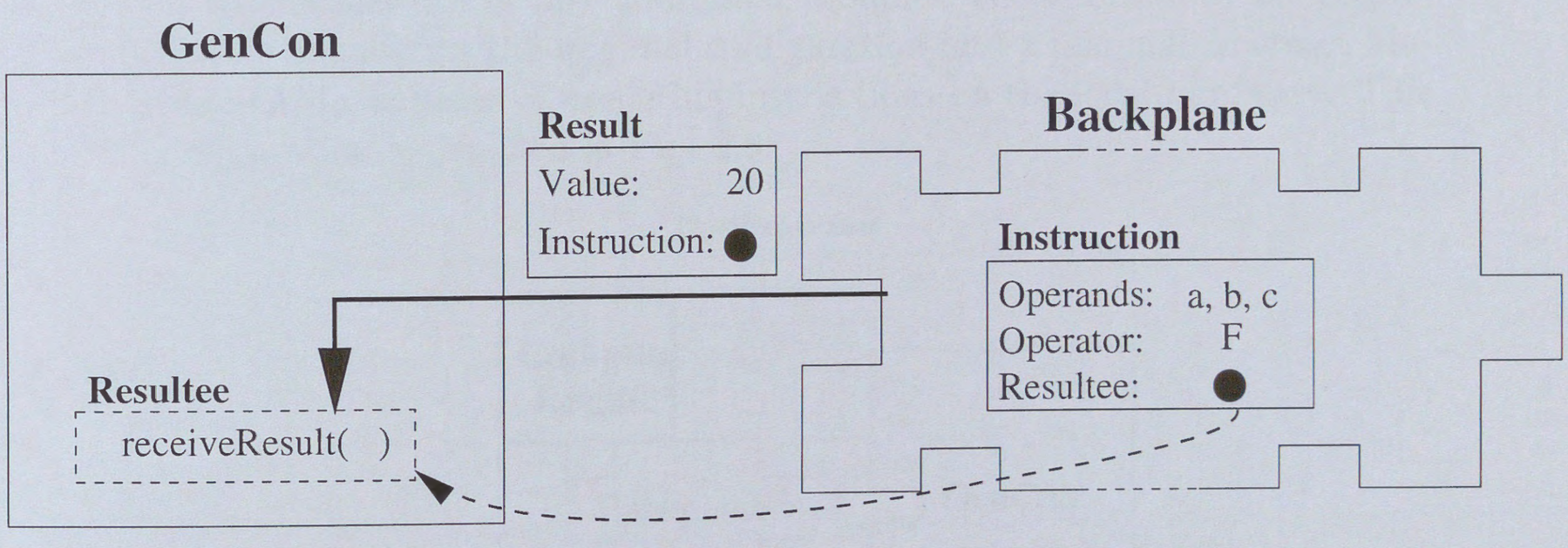


Figure 2.6: *The source instruction's* `Resultee` *reference is used to make a call to* `receiveResult()` *in the* `Resultee` *object itself.*

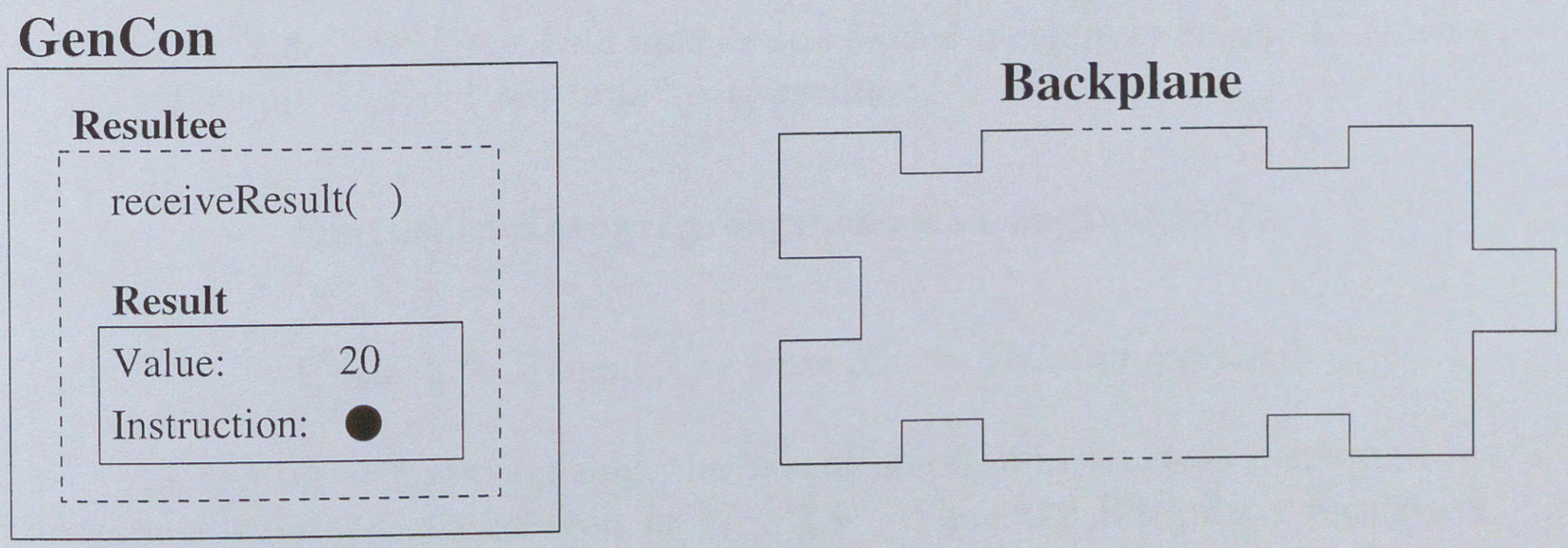


Figure 2.7: *WebCom has finished executing the instruction and the GenCon consumes the result.*

2.2 Abstract Machine

A Backplane constitutes a minimal WebCom installation whose only function is to load other modules. This can be initiated via the WebCom messaging subsystem or by consulting a configuration file on the local machine. All functional components of a WebCom installation, i.e., all WebCom Modules, are incorporated in this manner. Code 2.2 shows an entry in the configuration file instructing the Backplane to load a Condensed Graphs Engine Module. There is no limit to the number of modules a Backplane may load.

An Engine Module identifies and schedules instructions whose abstraction levels correspond to the capabilities present in that Engine Module. At the lowest level, the Engine Module will initiate threads of control to start the execution of machine level instructions on the underlying hardware. If the WebCom installation has the capability of executing more abstract instructions then the Engine Module will target instructions to those capabilities appropriately.

In the absence of any additional Modules, the addition of an Engine Module transforms the minimal configuration into a minimal Abstract Machine (AM), capable of executing instructions on the local hardware. This configuration is depicted in Fig. 2.8.

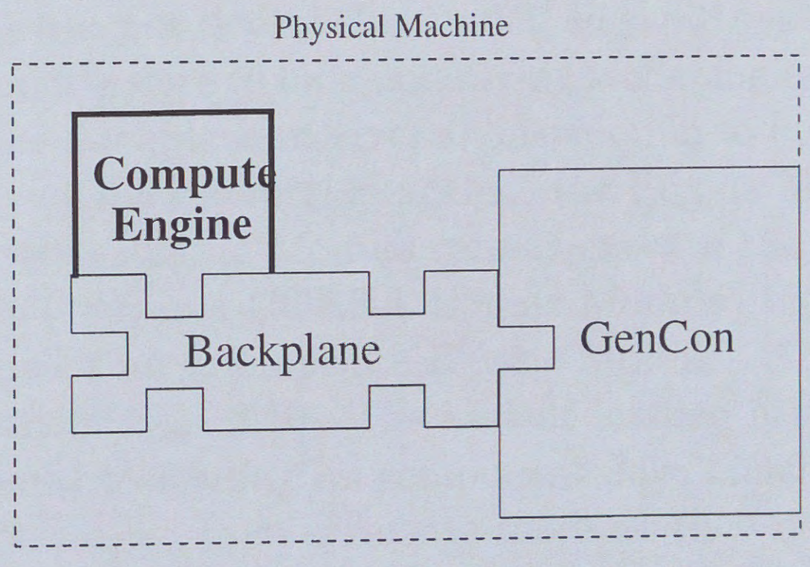


Figure 2.8: *The Backplane module has loaded a compute engine to become a "Minimal Abstract Machine" configuration.*

```
EngineModule=cg.rg.enginemodule.EngineModule
```

Code 2.2: *Sample line from file "modules.properties"*

A sample scenario showing the execution of an instruction generated by a user interface can be seen in Fig. 2.9. When the Backplane receives a message containing an instruction it routes the instruction to the Engine Module. The Engine Module then "unwraps" the instruction, executes it and generates a result object which is then passed to the Backplane where

it is subsequently consumed by the GenCon.

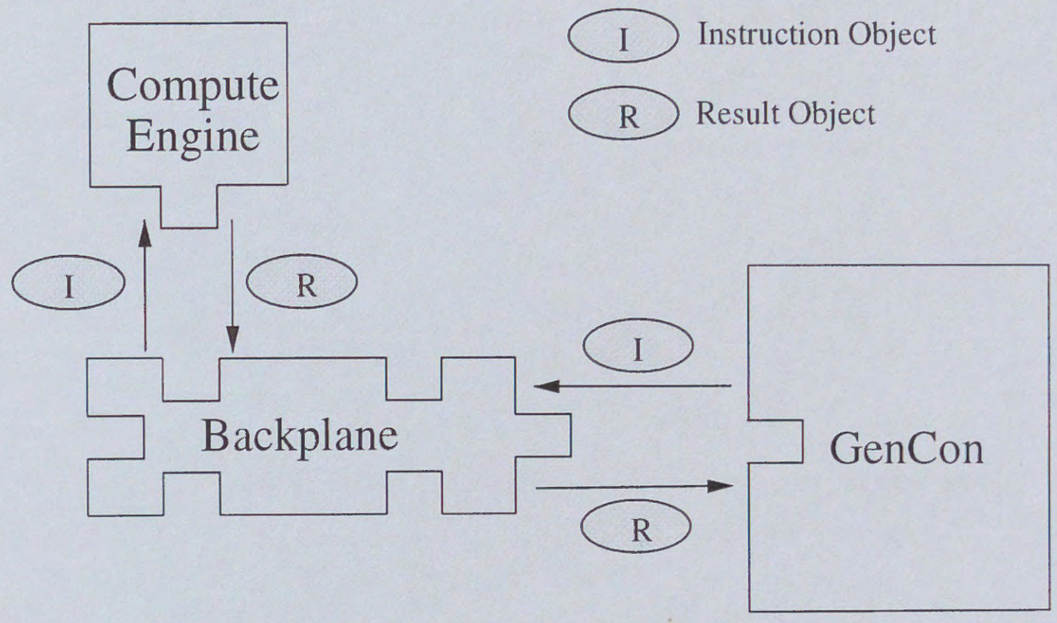


Figure 2.9: *The interface generates an instruction for the WebCom unit, which is given to the Backplane. The Backplane routes it to the Engine Module for execution. When the Engine Module has executed the instruction, it gives the result to the Backplane which then passes it back to the interface.*

Different compute engines may be developed for different kinds of instructions. Typically, WebCom uses a Condensed Graphs Engine Module, or $\mathcal{CG}$ Engine, but it is not tied to this model. In the *Anyware*[] project, WebCom is used as middleware to hide underlying technologies such as CORBA or COM. When the Backplane receives an instruction to be executed locally, it checks the type of the instruction against the Engine Module implementation. If the current Engine Module cannot execute the instruction (e.g., a COM instruction versus a CORBA Engine Module) then the Backplane will load and connect an appropriate Engine Module. An example of this process is depicted in Fig. 2.10. The cost of loading a locally stored Engine Module equates to loading its component Java classes. If the Module definition must be loaded from a remote machine then additional network latency costs will be incurred and will depend on the physical size of the Engine Module archive file(s).

In practice, the Backplane may run a number of different Engine Modules simultaneously. This advances the configuration from being a minimal Abstract Machine to one which can utilise an arbitrary number of technologies to execute instructions. Fig. 2.11 depicts a WebCom unit which, interfacing through its dedicated Engine Modules, simultaneously employs many underlying technologies to generate results.

This abstraction frees the programmer from managing the explicit targetting of specialised instructions to particular technologies. The system as a whole can be seen as a single abstract machine.

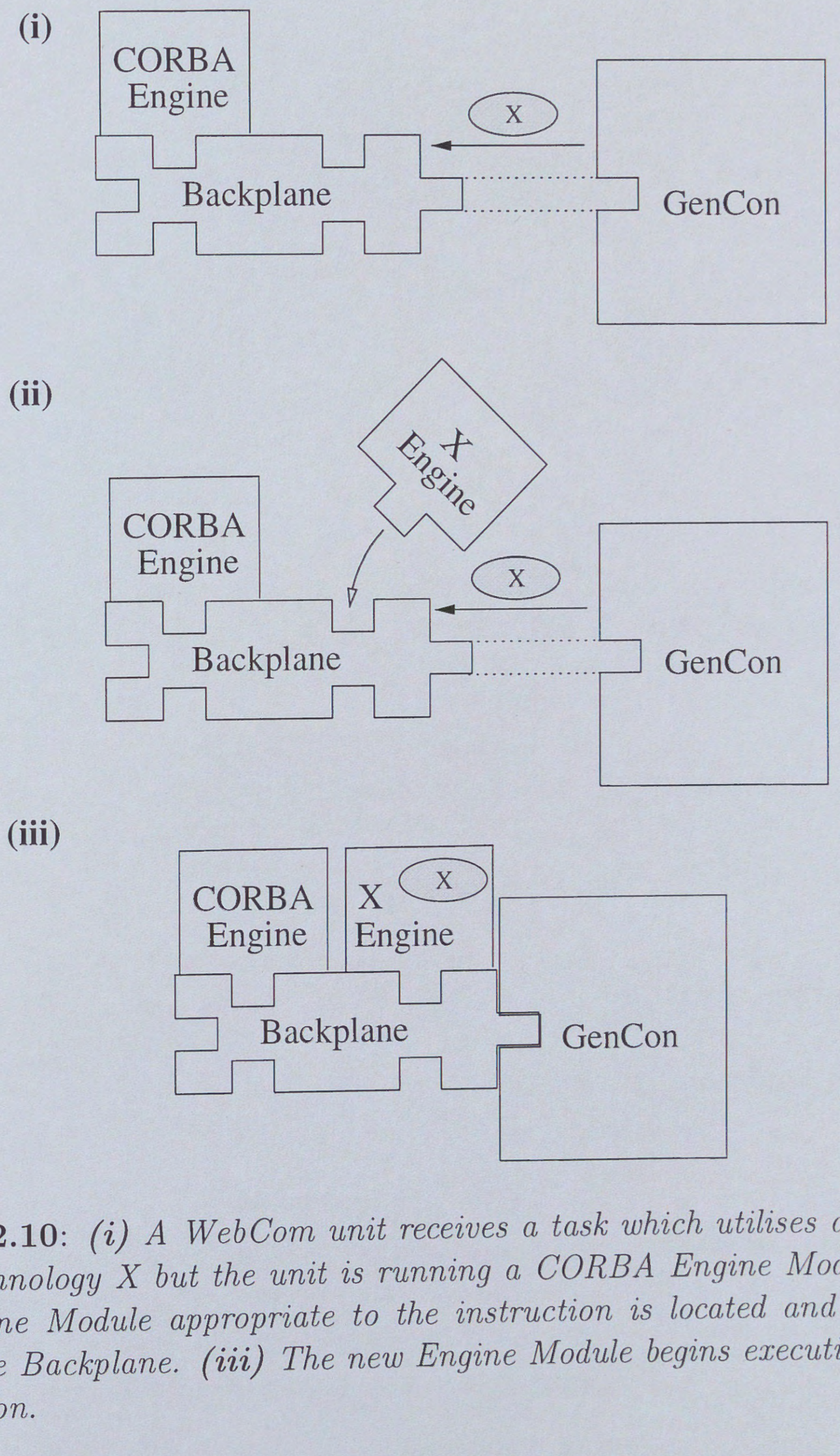


Figure 2.10: *(i) A WebCom unit receives a task which utilises an underlying technology X but the unit is running a CORBA Engine Module. (ii) An Engine Module appropriate to the instruction is located and "plugged in" to the Backplane. (iii) The new Engine Module begins execution of the instruction.*

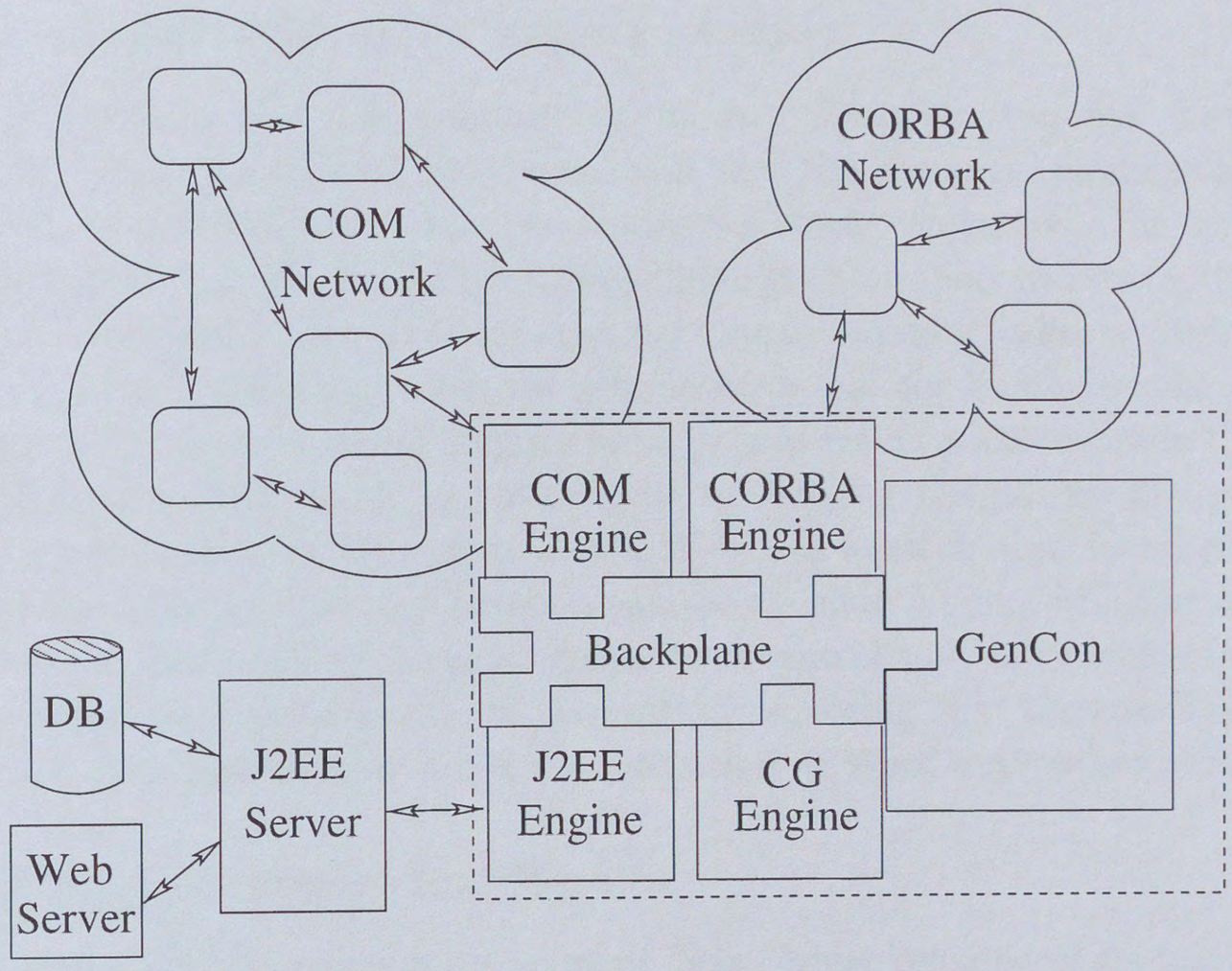


Figure 2.11: *A WebCom configuration with multiple Engine Modules, each interfacing with different underlying technologies. A single WebCom application could contain instructions utilising each of these technologies.*

```
public class EngineX extends EngineModule {
    public void executeInstruction(Instruction instr) {
        // Determine the operation to be carried out.
        XOperation op = LoadXOperation(instr.getOperation());

        // Run it with the operand data
        Object xResult = op.launchExecution(instr.getOperands());

        // At some point, launchExecution will return a value
        // that must be passed back to WebCom.

        // Wrap the result value
        Result result = new Result(xResult, instr);

        // Pass to the Backplane
        getBackplane().sendResult(xResult);
    }
```

Code 2.3: *Skeleton implementation of an Engine Module which utilises technology "X".*

2.2.1 Implementing an Engine Module

The integration of a new compute engine into WebCom requires that the default Engine Module be subclassed and that the `executeInstruction()` method be overridden. When a result has been generated, the Engine Module must call the BackPlane's `sendResult()` method to return the result to WebCom. The `executeInstruction()` method takes a WebCom `Instruction` object and unwraps it to create a task for its underlying technology. Current additional Engine Module implementations include COM, CORBA and J2EE modules. When the underlying technology has generated a result, it must be wrapped in a WebCom `Result` object and passed to the Backplane. Code 2.3 shows a sample skeleton Engine Module implementation. Each call to the `executeInstruction()` method is made from a new thread (one per executable instruction) ensuring that the execution of the task does not directly block the execution of WebCom's other threads.

WebCom Instructions and Results

Instruction objects contain an array of Java object references, representing the instruction's operand data, and a String representation of the name of the operator. These and other fields of an `Instruction` object are listed in Code 2.4. A `Result` object contains a single `Object` reference representing the result value and a reference to the `Instruction` object that led to the result. These fields are shown in Code 2.5. By employing these representations, any Java-to-X bridge may be used to link WebCom to any technology X.

Java bridges or application independent interfaces, such as network sockets or email interfaces, exist for most mainstream technologies. In the most primitive case an EngineModule may make a system call with command line operands to have the underlying operating system run an application. It may then use the application's return code or suitable file data as a result.

This architecture localises the complexity of dealing with an underlying technology to the particular machine that employs that technology and hides it from the rest of the WebCom system and to some extent, the programmer. As far as the application is concerned, a CORBA primitive node is conceptually no different from a J2EE node. Similarly, to the WebCom topology a CORBA instruction is no different to a J2EE instruction and a "CORBA-enabled" WebCom unit is no different to a J2EE WebCom unit. The routing of specific instructions to appropriate machines is discussed in Section 2.4.

```
public class Instruction
    implements Serializable
{
    // Graph or primitive instruction
    private CGType type;

    // Name of the instruction, either name of the graph or
    // name of the operator
    private String operation;

    // List of operands to the instruction
    private Object[] operands;

    // Node that this instruction represents
    private transient Node sourceNode;

    // Node that is created to run this instruction
    private transient Node executionNode;

    // Time instruction was sent/received.
    private transient long timestamp;

    // String representation of the sourceNode reference.
    private String sourceNodeRef;

    // Envelope is attached to instructions that have been sent
    // to remote host.
    private Message envelope;

    // Flag whether this instruction is referentially transparent
    // or not, i.e. will it always give the same result for the
    // same operands and operator.
    private boolean referentiallyTransparent;

    //Resultee object should receive this instruction's result.
    private transient Resultee resultee = null;
}
```

Code 2.4: *The fields of a WebCom* `Instruction` *object.*

```
public class Result
    implements Serializable
{
    // Node that generated this result
    private transient Node sourceNode;

    // The actual result value
    private Object value;

    // Message this Result may is attached to
    private Message envelope;

    // String representation of source node
    private String sourceNodeRef;

    // Resultee object that should receive this result.
    private Resultee resultee;
}
```

Code 2.5: *The fields of a WebCom* `Result` *object.*

2.3 Networking WebCom

It has been shown how a single WebCom unit configured as an Abstract Machine is a powerful tool in its own right. However, its resources are limited to those employed by its Engine Module(s). The WebCom system naturally extends by allowing separate WebCom units to communicate for the purpose of task execution. This is accomplished by a Connection Manager Module (ConMan). A ConMan Module extends the message passing capabilities of the WebCom Backplane to allow a Module on one machine to send a message to any other Module on any other machine with which it is in contact (see Fig. 2.12).

Connections between units are represented locally by *`Descriptor`* objects, illustrated in Fig. 2.13. A `Descriptor` contains information about the connection it represents, such as the actual network socket itself, whether the network socket is active or not, what outstanding instructions it has sent, as depicted in Fig. 2.14 . The Descriptor also manages listening for incoming messages via its own execution thread. This thread performs a blocking read on the Socket. When a Message object is read, the thread passes it to the ConMan module and waits to read the next message. The thread is idle whilst in this "wait" state, preserving CPU resources.

There is currently no need to have more than one WebCom unit running on a machine, i.e., more than one Backplane. It is quite possible that more

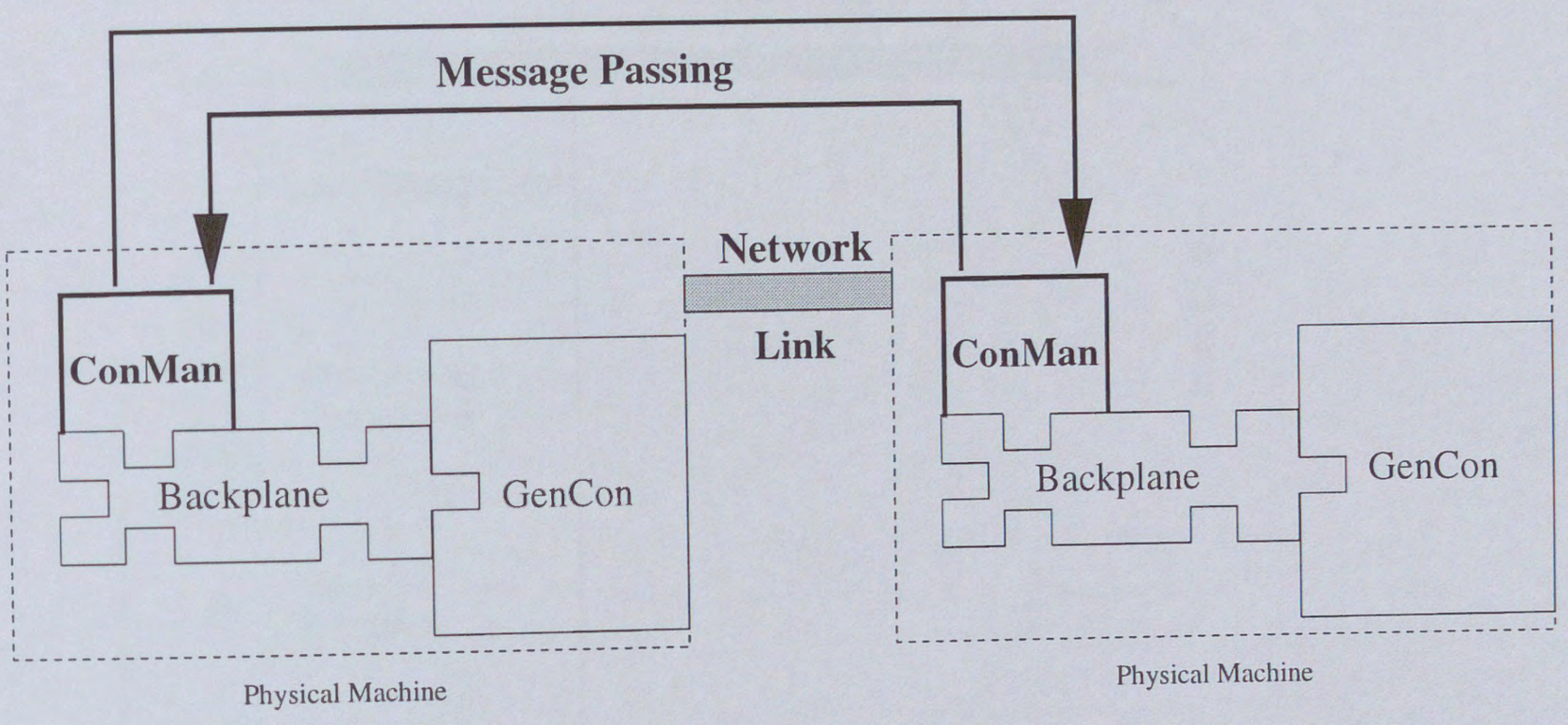


Figure 2.12: *WebCom units on two network connected machines. The addition of Connection Manager Modules to the configuration allows WebCom's message passing system to extend across the network.*

Figure 2.13: *WebCom units communicating via their ConMan modules. Within each ConMan module, connections are represented and managed by Descriptor objects.*

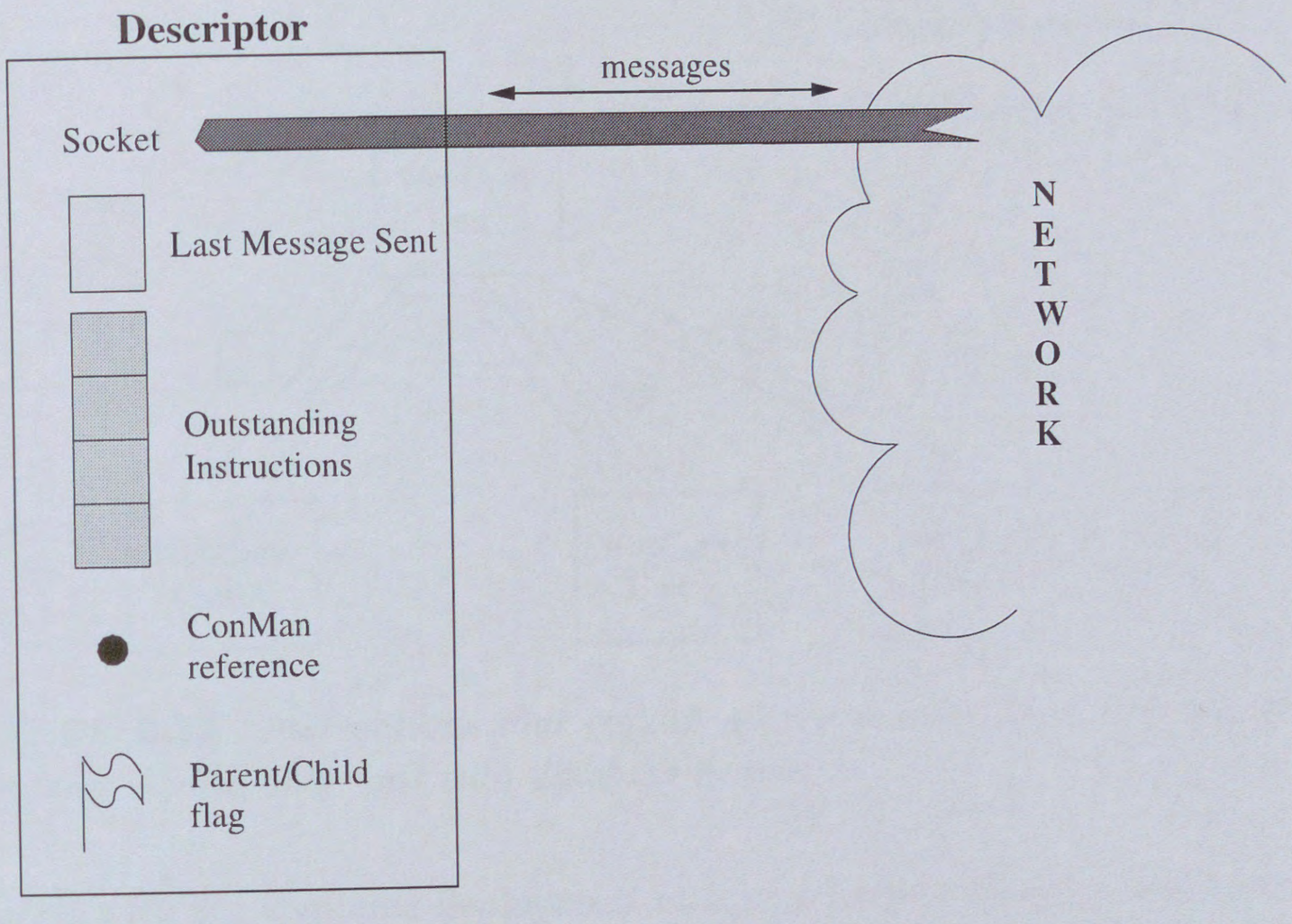


Figure 2.14: *A WebCom* `Descriptor` *object containing a network socket, a reference to its ConMan Module, a flag indicating whether the unit is seen as a "parent" or "child" of the remote unit, and a message history retained for fault tolerance and load balancing purposes.*

than one Engine Module may be required as discussed above, and other situations may require multiple versions of other modules too. For example, A machine with two network interfaces may require two different ConMen to manage the protocols used by each. Since the Backplane can always load multiple instances of a module type it would be both redundant and potentially troublesome to employ multiple Backplanes, e.g. two or more Backplanes may load a default ConMan which listens on a network port.

2.3.1 Distributing Computations

Messages sent between units may carry compute engine instructions or results. In this manner, connected WebCom units may send work to each other and a collection of WebCom units may be viewed as a single distributed execution platform. This facilitates the sharing of resources between machines. Fig. 2.15 depicts a set of WebCom units where Unit A has exclusive access to a particular database. Since the database connection cannot be moved to machines that wish to utilise it, instructions which require the connection are instead targetted to Unit A for execution.

To allow for this targetting, each connection made between WebCom units includes an initial handshake procedure in which each unit tells the other what instructions it can execute. This targetting mechanism, com-

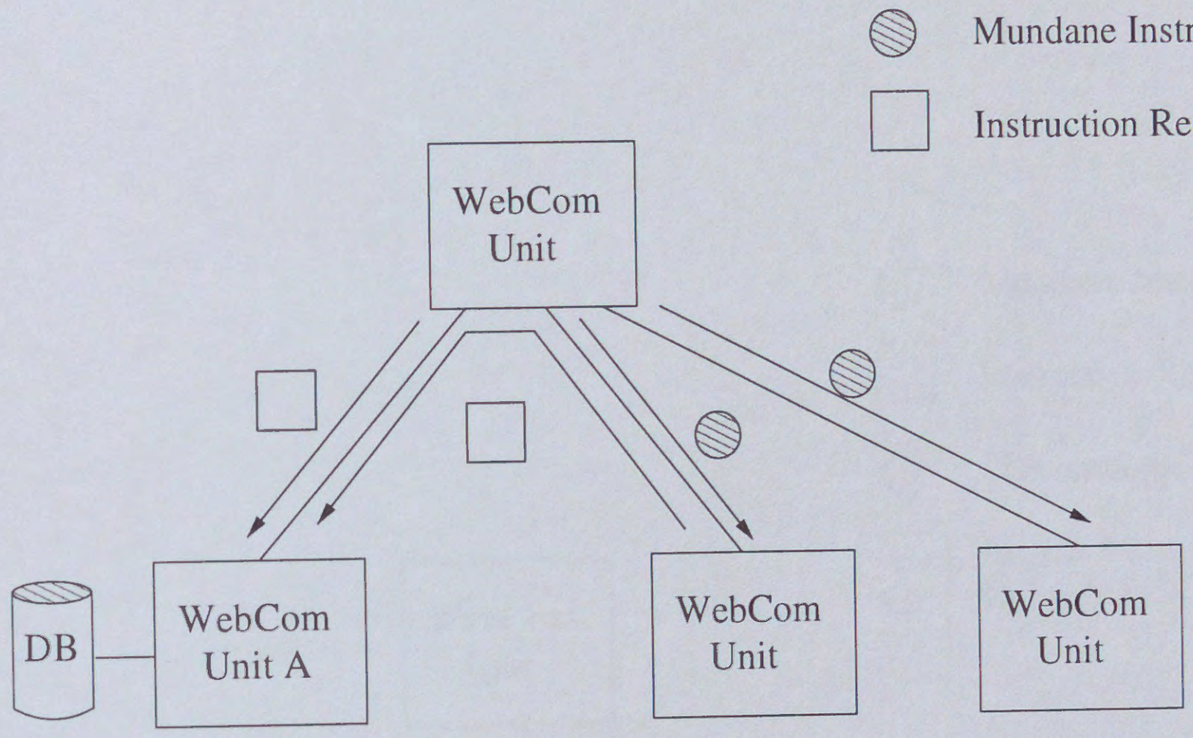


Figure 2.15: *Instructions that require access to a database are targetting to Unit A, the only unit with database access.*

bined with the dynamic deployment of actual Engine Modules and instruction definitions, simultaneously allows the movement of instructions to resources where required, and the movement of resources to instructions where possible. These processes are depicted in Fig. 2.16.

At this point, the term Abstract Machine describes not just a WebCom unit with a compute engine, but a collection of connected units. The system as a whole may be viewed by the application developer as a single distributed computing platform, or *metacomputer*.

2.3.2 Message Passing within WebCom

At runtime, the backplane acts as an intermediary between the actual functional modules. Since active module references may change dynamically, any intermodule communication must be done through the backplane to ensure up-to-date references to currently loaded modules. For essential functionality requirements, some modules may directly call methods in other modules, e.g., the current load balancing module calls the `getClients()` method in `ConnectionManager` to get an up to date reference to the list of currently connected clients. However, the majority of intermodule communication is done through WebCom's message passing system.

A WebCom `Message` object contains 3 references, two WebCom `Address` objects (source and destination) a data reference and a timestamp. The data reference is of the type `java.lang.Object` and so may refer to any type of object. The `Message` structure (and its associated `Address` object structure) is depicted in Fig. 2.17. The `Object` data reference allows complex message structures to be attached to the message. In practice, each module implementation often defines its own message type containing various information.

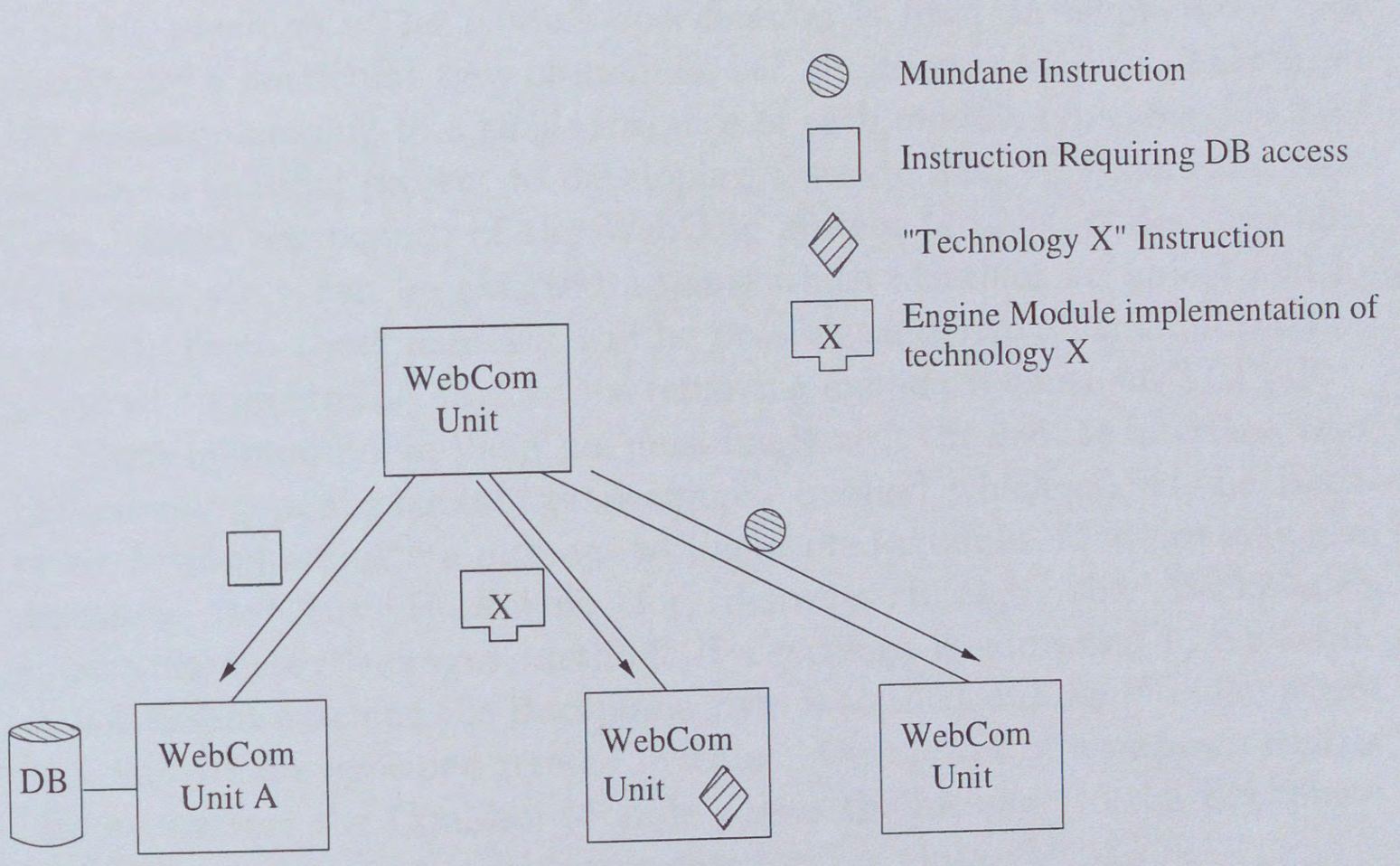


Figure 2.16: *WebCom scenario in which instructions requiring DB access are targetting to Unit A which has exclusive access to the resource. Unit B acquires a new Engine Module to execute its instruction.*

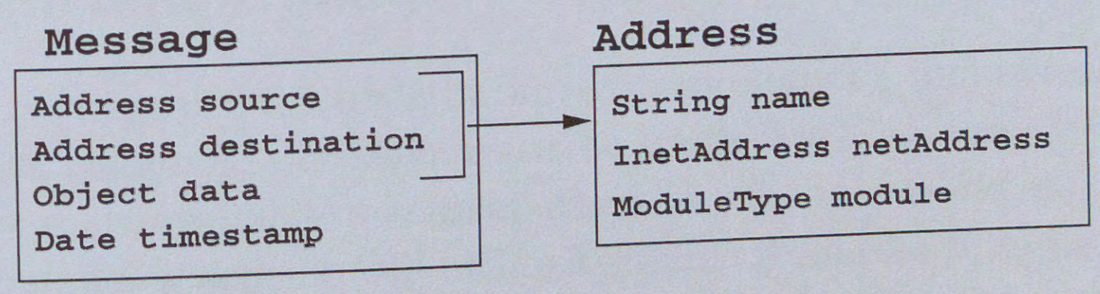


Figure 2.17: *Data fields of the WebCom* `Message` *object and its associated* `Address` *object.*

The `Address` object allows a message to be targeted to a particular module on a particular machine. The machine is specified by a network IP on the assumption that there will only be a single WebCom unit running on a single machine. The module specification is from an enumerated type specifying a particular *type* of module, i.e., the Engine Module. This limits the message passing to a single instance of each module type. Section 2.5.3 outlines a parallel project to developing a secure naming system for WebCom. Each component of the WebCom system (graphs, nodes, modules, messages, etc.) can be assigned a name which identifies an object and its context. From these names it will be possible to derive all the information required to locate the object thus removing explicit reliance on TCP/IP.

Since all modules in WebCom must implement the `Module` interface, they implement a `processMessage(Message)` method which allows the Backplane to communicate a message to the desired module. Modules may give messages to the Backplane for delivery though the Backplane's `routeToModule(Message)` method. If a message is addressed to a module on a different machine the Backplane gives it to the ConMan Module, where it is sent to the specified remote machine. On receipt of a message from a remote source, the ConMan Module passes the message to the Backplane which then routes it to the appropriate local module.

2.3.3 Relaying Messages

A WebCom unit connected to two or more other units may act as a relay for message communication. Fig. 2.18 shows a scenario in which Units A an B reside on separate subnetworks with no direct link between them. However, Unit C resides on some gateway machine and is visible to both of the other machines. Units A an B may communicate indirectly by having Unit C relay messages between them. The WebCom configuration on Unit C need consist only of a Backplane and ConMan Module.

2.4 Efficient Collaboration

To efficiently organise the distribution of work across a number of machines various *load balancing* decisions must be made, such as how much work to allocate to each available machine, when to schedule work, etc. An optimal load balancing scheme will utilise available resources to obtain maximal efficiency. In WebCom, load balancing means scheduling the execution of instructions, mapping instructions to available units appropriately and making "helper" units available to busy units appropriately.

An investigation into more advanced mapping techniques is underway in [55] where the Load Balancing functions mentioned above are dealt with by separate modules – an extra *Scheduling Module* is employed. However, for the purposes of this thesis, a simplistic approach to Load Balancing is

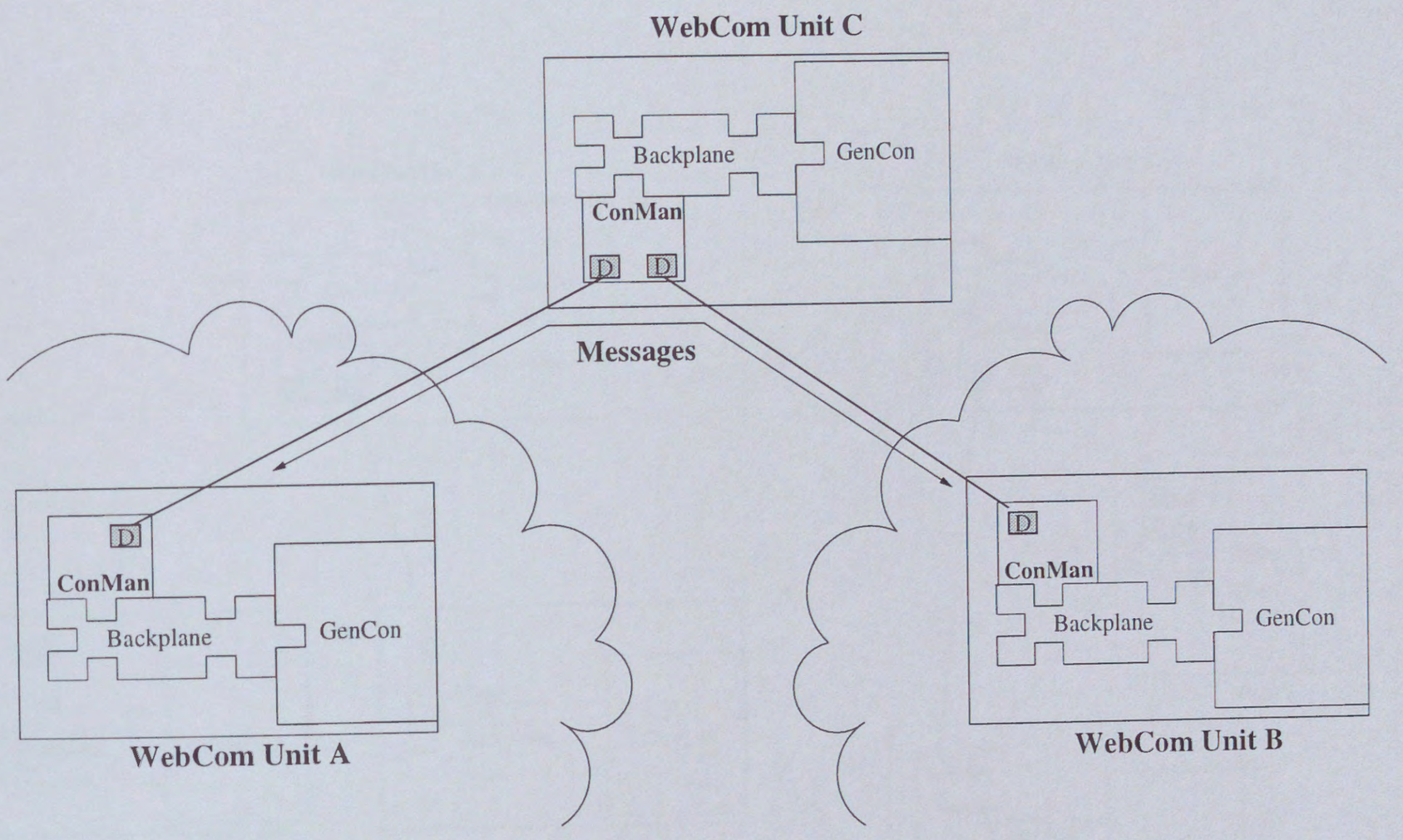


Figure 2.18: *Units A and B cannot directly connect to each other. Both instead connect to Unit C, which relays messages between them.*

taken. The Load Balancing Module utilised employs a simple Round-Robin system to decide which client a particular sub-task may be mapped to.

With this mechanism, a client unit that uncovers a large sub task of the computation may request help to execute its additional load. This area of decision making in WebCom is referred to as topology management and is performed by the Load Balancing Module in a WebCom unit.

The Load Balancing Module automatically employs its policies to effectively execute the computation on available resources. It therefore relieves the application developer from the burden of including load balancing code within the application and also caters for the dynamic addition of machines (The abrupt removal of machines is dealt with by the fault tolerance mechanisms described in Chapter 4). This further increases the abstraction of the "WebCom machine" from the physical configuration of component WebCom units by hiding the dynamics of the underlying topology.

2.4.1 Redirection Mechanism

The actual allocating of a unit is achieved through a *redirection* mechanism in WebCom. A WebCom unit may send a *redirect* message to another unit connected to it, a *client*. This message will contain the address of a third unit, to which the client will connect instead. Fig. 2.19 depicts this process as Unit X redirects Unit A to Unit B.

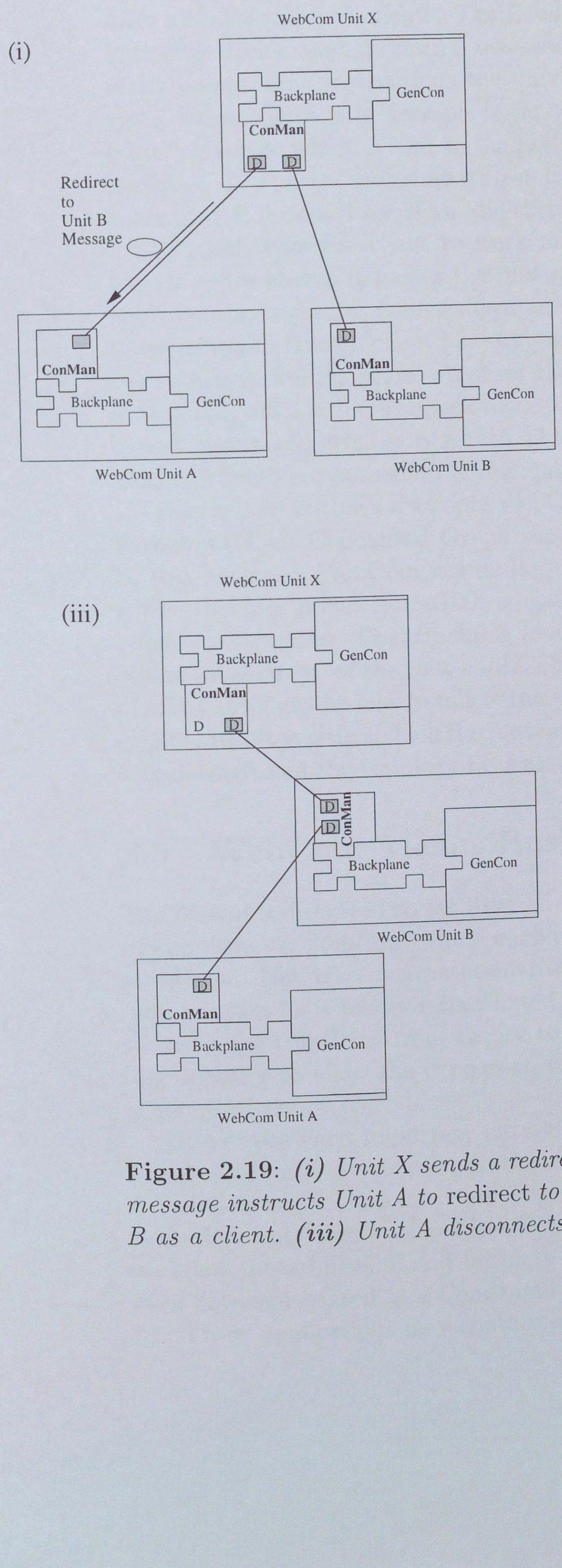


Figure 2.19: *(i) Unit X sends a redirect message to its child, Unit A. This message instructs Unit A to* redirect *to Unit B. (ii) Unit A connects to Unit B as a client. (iii) Unit A disconnects from Unit X.*

2.4.2 Topology Management

The currently topology management follows a simplistic policy to dynamically allocate units as clients. The Load Balancing Module on a particular unit may decide that its unit is overloaded. This decision is based on how many outstanding instructions the unit has waiting to be executed at any given time. Once this decision is made, the unit creates a "Request for client" Message which is sent to the Load Balancing Module on its "parent" machine. The parent examines its own clients and determines whether it can spare any. If it can, it sends an idle client a "Redirect" message (see Section 2.4.1) which tells the client to work for the busy machine. If the parent has no spare clients it passes the message to its own parent and so on. In this manner, units are passed down an evolving topology tree as sub-tasks uncover work. Once a client has completed its sub-task and sent its parent the result, it "relinquishes" all of its clients to its parent. As sub-tasks are completed, units will move back towards the root of the topology where the overall result will eventually return. Any idle Units that connect to an idle Unit as clients are immediately relinquished.

Figure 2.20 outlines a sample WebCom topology evolution in which the **PowersOfTwo** Condensed Graph shown in Fig. 2.21 has been executed. In this example, WebCom is running on five nodes of a dedicated cluster where the first node, Node 100, is considered the "root" and is given the original instruction. The remaining four nodes begin the execution as helper nodes, or *children*, of the root node. The execution shows the redistribution of effort through the tree to follow the workload of the execution. When the original result is returned via the root node, all other machines are eventually relinquished and the topology returns to its original configuration.

2.5 WebCom Means Business

The Modules discussed so far have all been concerned with the addition of mechanisms to WebCom which enable it to distribute and optimise computations. However, computational efficiency is just one of a number of requirements for a modern distributed computing platform. For a platform to make the transition from theory to industry it must provide additional functionality to allow the expression of business concerns in the utilisation of the platform.

One of the most important industry concerns with any networked tool, especially a general execution platform such as WebCom, is that of security. A WebCom application may vary from simple numerical computations, such as the PowersOfTwo example (see Fig. 2.21), to the complex automation of workflow procedures[29] and business practices. An example of a business work flow represented as a Condensed Graph is depicted in Chapter 3, Fig. 3.1. These applications may contain sensitive data which can only be used

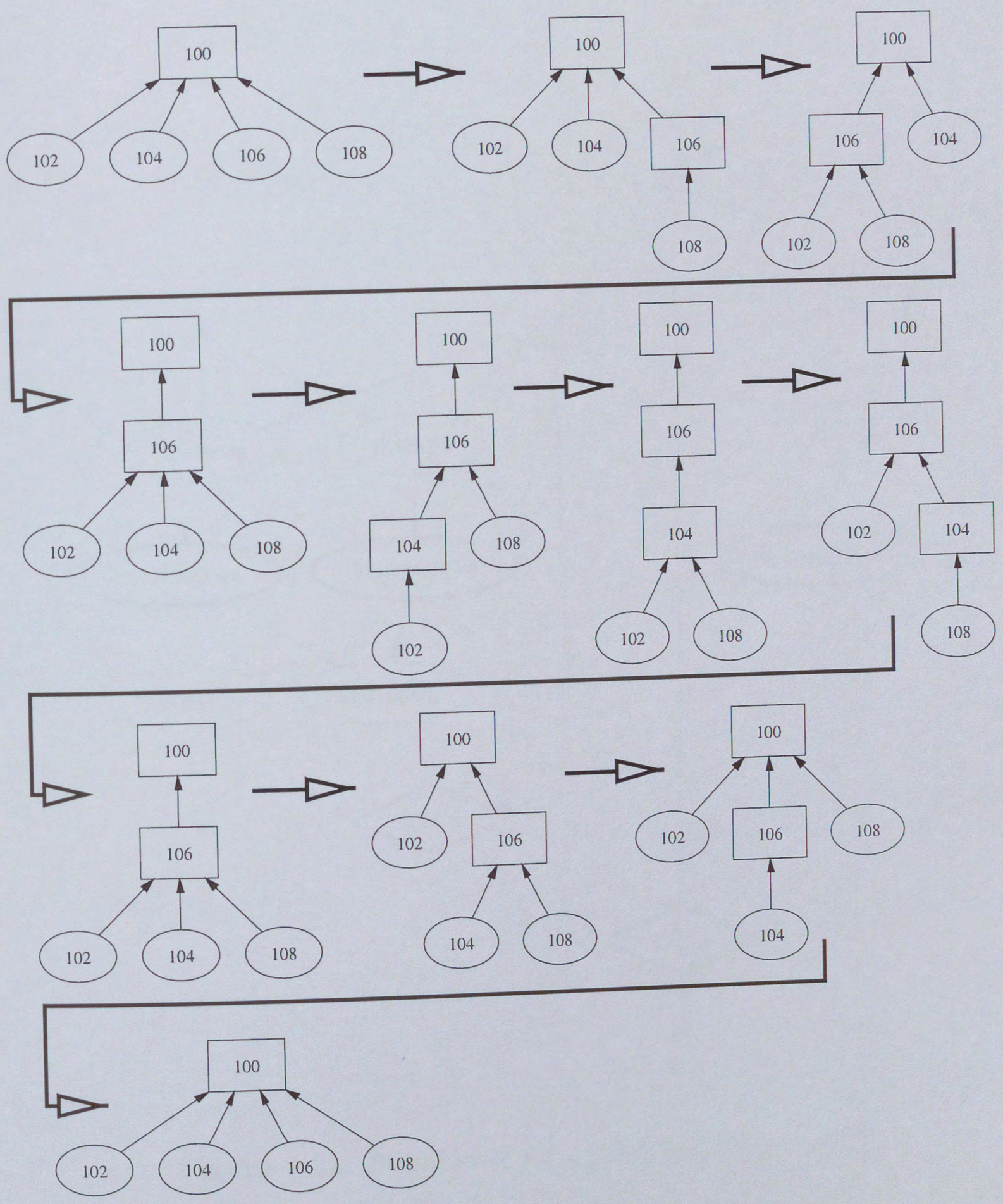


Figure 2.20: *Evolution of a WebCom topology consisting of 5 machines during the execution of a PowersOfTwo graph with $n = 10$. Unit 100 begins as a "parent" to the remaining units. As the execution begins, its "children" request helper units and the topology tree grows deeper. Helper units are relinquished as subtasks complete, eventually returning the topology to its initial state when the computation has completed.*

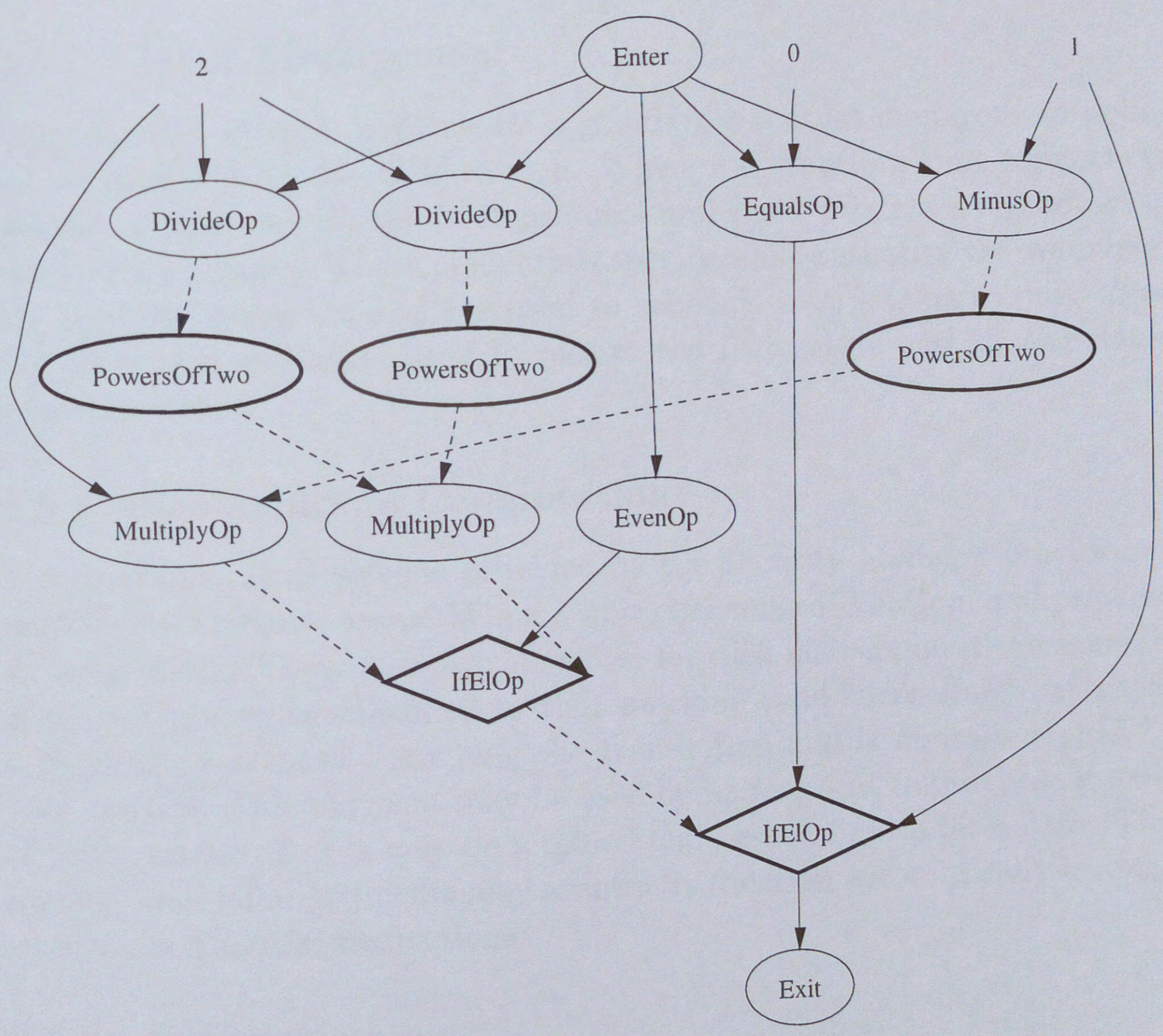


Figure 2.21: *Initial state for a "PowersOfTwo" graph.*

on certain trusted machines, or may require access to restricted resources such as bank account information or a customer's credit rating. These constraints can be expressed in WebCom through the addition of the *Security Manager Module* to the configuration, which introduces the concept of *Trust Management* into both application design and runtime management. A secure naming scheme and a system of micropayments to bring an economy to WebCom are also under development.

The Security Manager Module that provides these features is under development in a parallel project. It is summarised here for completeness.

2.5.1 Trust Management

The Security Module implements a standardised trust management policy by utilising the KeyNote[10] system. When one WebCom Unit connects to another it does so via the SSH protocol and must provide a signed set of security *credentials*. These credentials specify what authority the volunteering machine possesses and are used to schedule instructions to only those machines that are authorised to receive the instruction and its associated operand data.

2.5.2 Economics of Computations

The micropayments scheme provided by the Security Manager provides an economy of resource usage. When a unit connects to WebCom and provides its credentials, it may also provide a cost for each instruction it can execute. If an instruction is scheduled to that machine (and successfully executed) a payment is created describing the transaction and is digitally signed by both parties. This payment may be used by its recipient to purchase the use of other resources or it may be retained and cashed in at a later date. This cashing in of micropayments may eventually function automatically through traditional financial institutions.

2.5.3 Secure Naming

The secure naming under development for WebCom allows entities to be described in terms of their context, rather than a simple unique identifier. This is a powerful tool for metacomputing as there are many cases in which an object can be referred to in terms of its function, or its location in a topology. This facility will be employed by WebCom to allow a more generic message passing system, specifically by removing the current dependency on a TCP/IP to identify unique WebCom units. In fact, the need to directly reference a unit may be eliminated, i.e., a message could be sent to the unit with an Engine Module that is currently executing a particular node *N*. In addition, the Fault Tolerance mechanisms discussed in Chapter 4 can be optimised using this mechanism.

2.6 Summary

This chapter has presented the architecture and features of the WebCom metacomputer. A WebCom metacomputer is composed of one or more units, each of which are in turn composed of modules. These modules provide functionality such as load balancing, fault tolerance and actual task execution. Management decisions such as connection management and load balancing are made by WebCom at runtime. These burdens are therefore removed from the application and the developer is free to concentrate on expressing the desired functionality. This expression can be facilitated by the Condensed Graphs model, which provides a rich set of semantics for parallel applications. The following chapter discussed that model and its role in the development of WebCom applications.

Chapter 3

WebCom Applications

3.1 WebCom with Condensed Graphs

The Condensed Graphs model, discussed in Section 1.3, is a particularly expressive model for distributed application development. As such, it is the default Engine Module for a typical WebCom unit. Arbitrarily complex tasks and graphs can be defined in libraries of nodes. These nodes are then hooked together by the programmer to create an application. A $\mathcal{CG}$ application may be a very simple graph containing one or two primitive instructions or a complex multi-graph application, such as the $\mathcal{CG}$Chess program described Section 3.2. The graphs depicted in Fig. 3.1 show how nodes are not restricted to computational functions: the *OrderParts* graph utilises COM to invoke Microsoft Word and print an appropriate purchase order.

The firing rule in the $\mathcal{CG}$ model allows implicit determination of parallel tasks and the condensing of whole graphs into condensed nodes allows a high degree of abstraction during application development. In addition, features of the model itself facilitate natural heuristics for load balancing and fault tolerance. Legacy code may be easily transformed into a Condensed Graphs application by breaking it into functional units which are represented by nodes. Even if these units are very large grain with little or no parallelisation possible, the application will still take advantage of WebCom's resource management and the programmer does not have to worry about scheduling, fault tolerance etc.

3.1.1 The Condensed Graphs Engine

As discussed above, the implementation of the Condensed Graphs model as an Engine Module for WebCom provides novel features of which WebCom can take advantage. The implementation itself utilises the Object Oriented design methodology to create a system of nodes and graphs. These graphs inherently contain their current execution state.

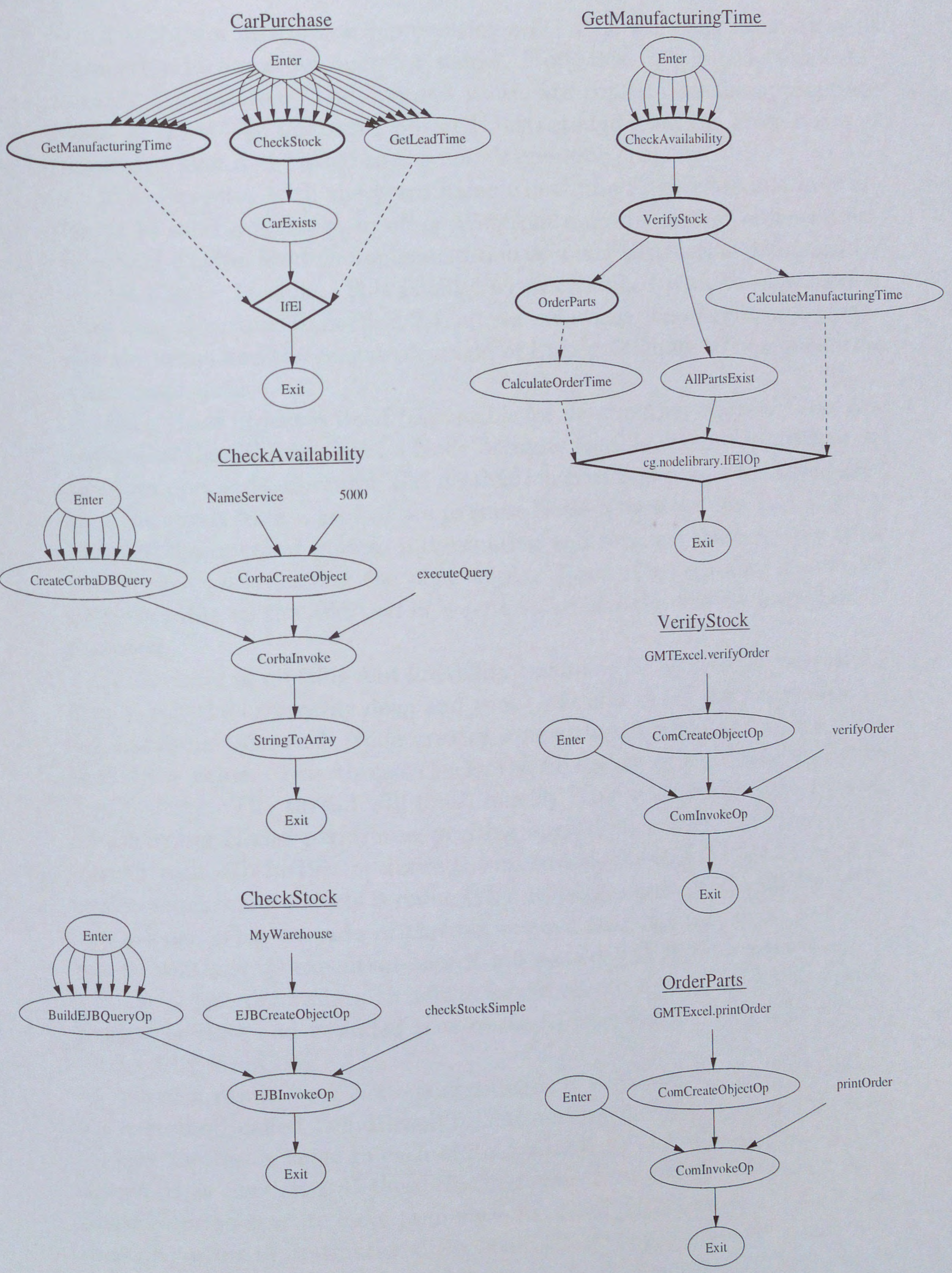


Figure 3.1: *The* CarPuchase *and its associated subgraphs. These graphs demonstrate the use of COM and EJB services in WebCom for a fictional workflow process. When a car is ordered on a web site, the graphs determine the availability of parts and place the order.*

A WebCom instruction representing a $\mathcal{CG}$ node will contain an array of operand values and an operator name. From this, the initial "top-level" node is instantiated. The operand values are copied into the appropriate operand ports. An `Operator` object is instantiated from the given name in the instruction and passed to the node's operator port.

If no operator with the given name exists, the Engine Module may attempt to send a message to other WebCom units to request either an appropriate Engine Module implementation or a new instruction definition for a local Engine Module. This facility, when combined with the Instruction targetting described in Section 2.4, allows the utilisation of existing technology platforms and the easy deployment of Engine Modules to new platforms where appropriate.

Each Node object is itself responsible for determining its fireability. According to the $\mathcal{CG}$ firing rule, a Node becomes fireable when it has values on all of its operands, operator and destination ports. Where a Node requires only the result from a *grafted* arc to some Node N (see Section 1.3), it gives Node N the operand port as a destination and removes the reference from its operand port, leaving the port empty. Node N's fireability must now be tested due to the addition of a port value and the testing procedure is repeated.

This chain of grafting and fireability tests may be arbitrarily long (since graphs may be arbitrarily deep and wide) and may delay the active thread. To overcome this, each Node creates a new thread when any of its ports is given a value. This thread checks the fireability of the Node and does any grafting. The thread will finish quickly since it does not have to wait on any other Nodes performing grafting operations – they will create their own threads. To further optimise the execution, the costs of allocating and deallocating many objects is reduced by utilising a pool of threads for Node operations. This cascade of threads ensures that the initiating thread is free to continue its execution once it has passed a Port value to a Node. If a Node object determines that it is executable it reports its status to the Engine Module and is added to a queue of executable nodes that will be serviced by WebCom.

When a result for a Node is generated, it is passed to the Node object via a method called "sendResultToDestinations()". This causes the Node to copy the result value to each of its destinations and then deconstruct(as described in Section 1.3) these destinations. It is therefore possible for a single Node to execute more than once if it is subsequently given a new destination, or set of destinations. Section 3.5.3 presents a sample scenario in the CGChess application where this occurs. It also presents an optimisation that may be used during graph construction to retain referentially transparent results for later consumption using the stored data characteristics of the Condensed Graphs model. When the result of an unnecessary speculative computation arrives, the $\mathcal{CG}$ containing its destination may have already

finished and no longer exist. In this case, a Java `NullPointerException` is detected and the result is simply discarded.

The execution of condensed graph nodes follows the same $\mathcal{CG}$ firing rule. However, when a $\mathcal{CG}$ node fires it causes the instantiation of a new `CondensedGraph` object. A `CondensedGraph` object contains an "Enter" node, an "Exit" node and an arbitrary number of nodes which comprise the body of the graph. Some of these nodes will be populated with port values upon creation. Most will start with an operand value, some may start with "constant" operand values or grafted arcs, and some will start with a set of destinations (such as the Enter node). An example $\mathcal{CG}$ initial state can be seen in the PowersOfTwo graph of Fig. 2.21. Figures 3.2 to 3.4 illustrate the point in the execution of a PowersOfTwo graph where a recursive condensed node becomes fireable. In Fig 3.3 the Enter node is given a copy of the parent node's operand list and the Exit node is given a copy of the parent node's destinations. At this point the parent node is considered to have executed and its own destinations are deconstructed as normal, as depicted in Fig. 3.4.

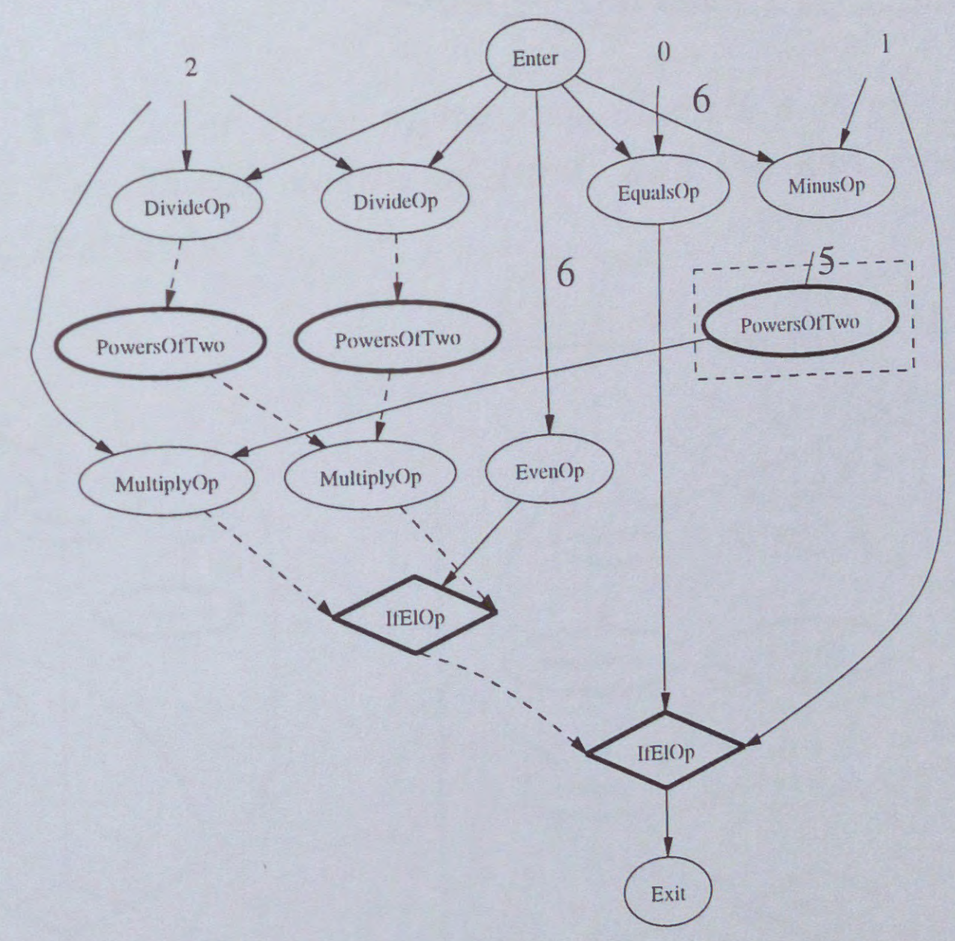


Figure 3.2: *The indicated Condensed Node in the PowersOfTwo graph $\mathcal{CG}$ becomes fireable as it has received all of its required operand, operator and destination port values.*

Once the allocation of the graph object is completed, it is given a reference to its Engine Module. This enables the creation of threads to test fireability and thus enables the firing of its component Node objects. The Enter node will report itself to the Engine Module for execution. Its firing will propagate more operand values throughout the graph until ultimately

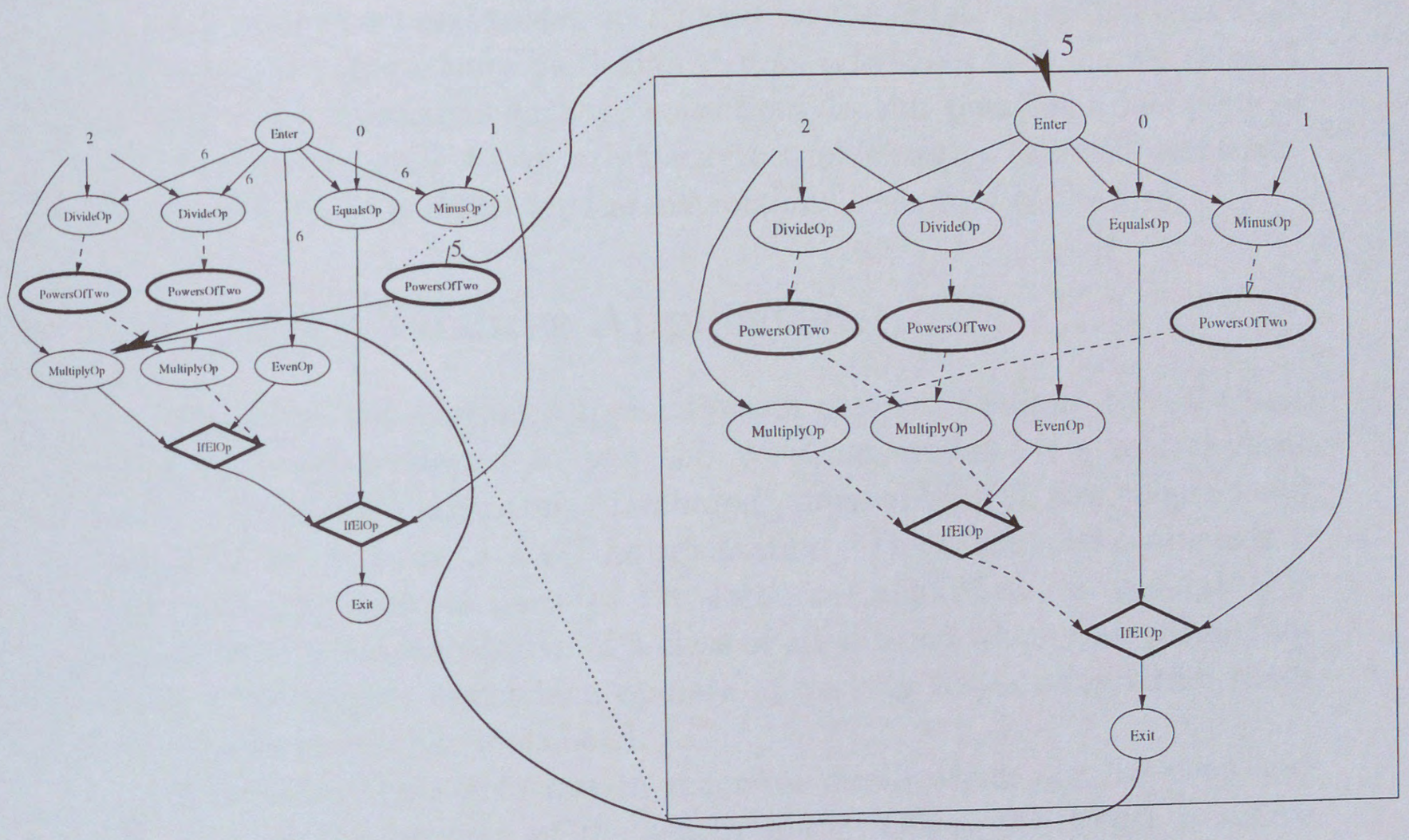


Figure 3.3: *The Enter Node of the new allocation is given a copy of its corresponding Condensed Node's operands and the Exit Node is given the corresponding destinations.*

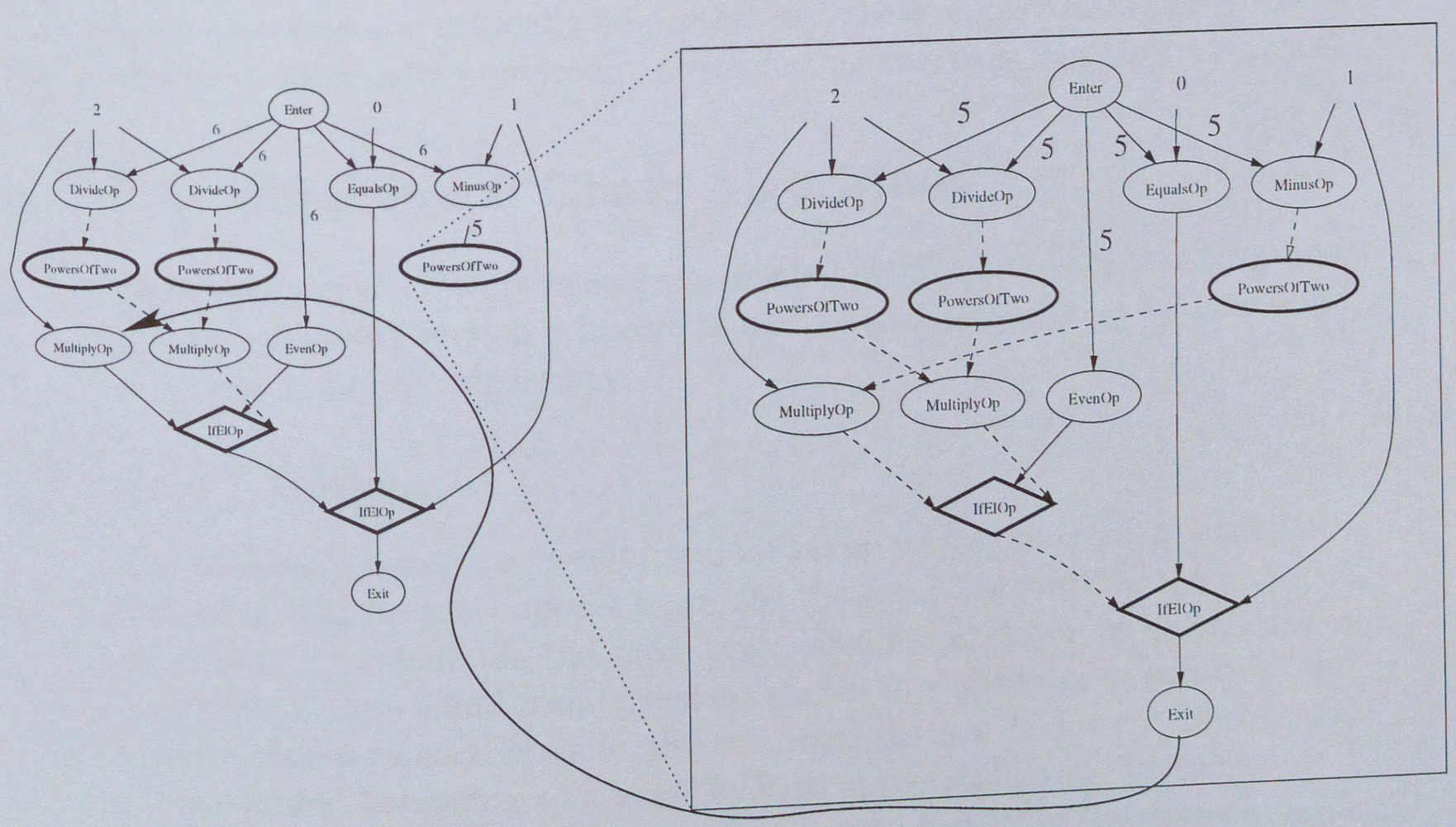


Figure 3.4: *The Condensed Node is considered to have executed and deconstructs its destinations. The newly allocated graph begins execution with its Enter Node.*

the Exit node fires and passes on a result for the graph. Once the Exit node has fired, the graph may be deallocated so references to it are set to `null` to flag it for automatic garbage collection. In this manner, a hierarchy of subgraphs may grow to an arbitrarily complex degree but will eventually collapse down to a result for the instruction at the root of the tree.

3.2 The CGChess Application

In contrast to the synthetic applications in previous Sections, the $\mathcal{CG}$ Chess program was developed to test the WebCom system on a real application. The parallel algorithm "Jamboree", created by [26], was adopted and adapted to conform to a $\mathcal{CG}$ representation. These adaptations include a selective execution of parts of the Jamboree algorithm. In general, this application is representative of a class of application which uses parallel recursive techniques and which operate at varying scales using either mesh refinement or n-stage lookahead.

Such algorithms give rise to recursive depth which may be deep and vary across the breadth of the search space. Often, the deeper recursive invocations are the most important, the most expensive and hence the most vulnerable to faults. If these applications are executed on a distributed platform, the occurrence of a fault may result in a partitioning of the computation and the orphaning of many subcomputations. Fault tolerance and fault recovery are critically important for these applications and so they provide a demanding environment for testing the WebCom fault tolerance.

3.3 Traditional Chess Algorithms

Minimax is at the root of most game related artificial intelligence algorithms and an optimised version is known as $\alpha\beta$. A parallel version of $\alpha\beta$ is referred to as the Jamboree algorithm.

3.3.1 Minimax

The Minimax algorithm is employed for many *perfect knowledge* games, i.e., games in which each opponent can see the complete state of the game at all times. It calculates the next best move for a player by generating a tree of legal moves and countermoves, known as a Minimax search tree, and applies scores to each node in the tree from the leaf nodes up. Leaf nodes are evaluated according to a game specific static evaluation function.

The goal of the algorithm at each step is to maximise the score for each of MAX's moves and to minimise the score for each of MIN's moves, where MAX and MIN are specific terminology given to the moving player and the opponent. Scores are usually implemented so that a positive final score is a

win for MAX and a negative score is a win for MIN. The larger the absolute value of the score the greater the win or loss. A Minimax algorithm that uses this scoring system is sometimes referred to as Negamax.

To show the steps taken by Minimax to choose a move a simple two ply (two move) game will be considered. Player 1 may make either Move A or Move B while Player 2 may make either Move C or Move D in response. The final game score for each combination of moves is shown in Fig. 3.3.1

MaxMove	MinMove	Evaluation
A	C	12
A	D	-2
B	C	5
B	D	6

Figure 3.5: *Example moves in a 2-ply minimax game*

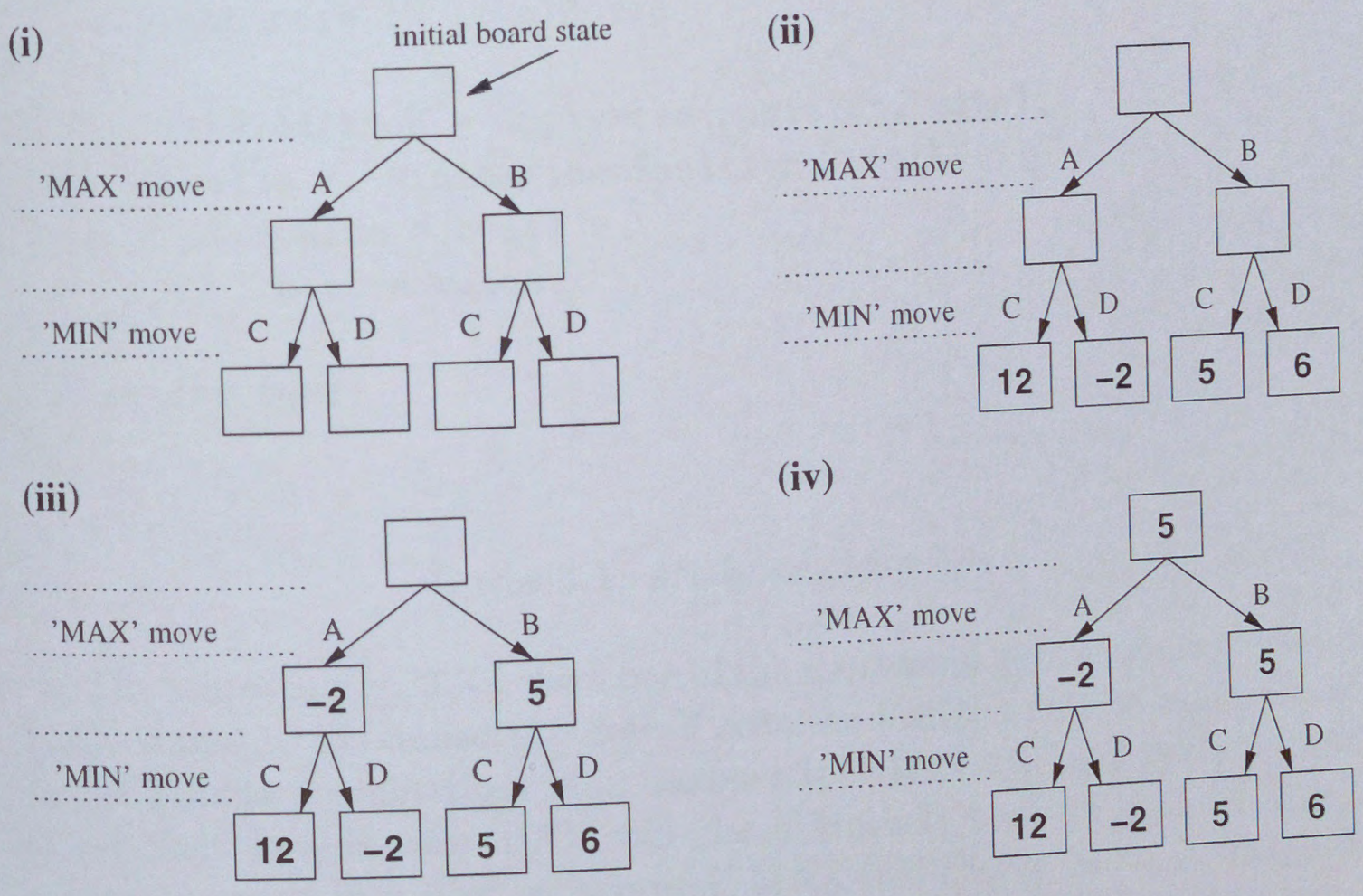


Figure 3.6: *Minimax search of two ply tree.*

The steps of the Minimax algorithm depicted in Fig. 3.6 are as follows:

Fig. 3.6 (i) shows an initial Minimax tree for a two-ply game where each player can make one of two moves. The moving player, "MAX", can make either Move A or Move B. The opponent, "MIN", can make either Move C or Move D in response.

In Fig. 3.6 (ii), leaf nodes of the tree are evaluated to yield a positive score representing a win for MAX or a negative score representing a win for

MIN. The higher the score the better the win for MAX; the lower the score, the better the win for MIN.

In Fig. 3.6 (iii), the countermoves that MIN would make in each case leading to the leaf nodes are evaluated by choosing the best outcome(the lowest score) for MIN in each case.

In Fig. 3.6 (iv), the MAX move is chosen by picking the best highest score from the newly evaluated child positions. MAX will chose Move B and MIN will chose Move C.

A pseudocode implementation of the Minimax algorithm is presented in Code 3.1. In this code, the opponents score is calculated as normal and negated.

```
score Minimax (position, depth) {
    if (depth == 0)
        return StaticEvaluation(position);
    best = -INFINITY;
    legalMoves = GenerateLegalMoves(position);
    foreach move IN legalMoves
    {
        newPosition = ApplyMove(position, move);
        value = -Minimax(newPosition, depth-1);
        if (value > best)
            best = value;
    }
    return best;
}
```

Code 3.1: *Minimax algorithm.*

The scores in Fig. 3.3.1 show one of the shortcomings of Minimax. Player MAX wants to maximise the overall score and MIN wants to minimise it. In the Minimax evaluation, Max assumes that MIN will play perfectly and vice-versa. In this case MAX will chose Move B because both of MIN's responses result in a positive score. If MAX had chosen Move A, it might have won by a larger margin. This confines the computer to "safety first" strategies which are not optimal.

Another practical problem with Minimax is that the search tree grows exponentially as its depth increases. This can be addressed in implementations by using a maximum depth parameter (as in Code 3.1) and by restricting the search tree to those branches that are likely to lead to a good move. The $\alpha\beta$ optimisation is employed for this purpose.

3.3.2 AlphaBeta

The AlphaBeta($\alpha\beta$) algorithm has been used since the 1960s to try to overcome the practical difficulties of searching a Minimax tree of any meaningful depth. A Minimax search is performed as before but is bounded by two additional values, α and β, which are maintained as the search progresses. In the investigation of a number of moves for a particular player the value of α is used to maximise the advantage while the value of β is used to minimise the advantage of the opponent for that move. In general, β represents an upper bound for the best move while α represents the best move determined at a particular point. If the algorithm terminates with α equal to β then an optimal move will have been found.

The following is a pseudocode representation of the $\alpha\beta$ algorithm:

```
score AlphaBeta (position, depth, alpha, beta)
{
    if (depth == 0)
        return StaticEvaluation(position);
    best = -INFINITY;
    legalMoves = GenerateLegalMoves(position);
    while (not Empty(legalMoves) && best < beta)
    {
        move = PopMove(LegalMoves);
        newPosition = ApplyMove(position, move);
        if (best > alpha)
            alpha = best;
        value = -AlphaBeta(newPosition, depth-1, -beta, -alpha);
        if (value > best)
            best = value;
    }
    return best;
}
```

Code 3.2: *AlphaBeta algorithm*

The efficiency of this algorithm is dependent on the order in which moves are searched. Optimally, the best move will be expanded first and all other moves quickly discarded in comparison. In the worst case it will search the moves in ascending order of value, in which case the $\alpha\beta$ search degrades into a pure Minimax search. The $\mathcal{CG}$ implementation presented here uses a simple static chess position evaluation, based mainly on material piece counts and capture positions. Section 3.7 presents some optimisations that can help to improve the move ordering and reuse the results of previous searches but these rely on structures not implemented in the $\mathcal{CG}$ version.

Under optimal move ordering conditions, $\alpha\beta$ has to search

$$W^{\frac{D+1}{2}} + W^{\frac{D}{2}} - 1$$

positions, which is still exponential in D[33]. $\alpha\beta$ will usually search twice as deep as a basic Minimax search with the same effort, allowing the algorithm to look twice as far ahead in terms of game moves.

3.3.3 Aspiration Search

An aspiration search is a minor modification of the $\alpha\beta$ search. Usually, the $\alpha\beta$ search begins with -INFINITY and +INFINITY as the initial values for α and β, thus guaranteeing some move will be chosen as its score must be within the range defined by α and β. In an aspiration search, the α β values are set to define a smaller window around some expected result. In this way, many cut-offs are quickly achieved and so the search tree is drastically reduced. One drawback is that this algorithm may fail if no moves are found that score within the search window. However, evaluated moves that score outside the window will either fail high or low. This information is used in the parallel optimisation, the Jamboree algorithm.

3.3.4 Jamboree Algorithm

In order to achieve savings (through early cut-offs pruning the majority of the search tree), each iteration of the algorithms presented above must use the result of the previous iteration. Therefore, this approach does not lend itself well to parallel execution. In fact, it has been shown that if the serialisation requirements of $\alpha\beta$ pruning are ignored, the parallel program will only achieve $\sqrt{\mathbf{n}}$ speedup on **n** processors[17]. A variant on $\alpha\beta$ search called Scout[52] has been parallelised in the Jamboree algorithm used by the *Socrates Cilk-chess application[11]. It is this algorithm that is adopted by the $\mathcal{CG}$ Chess application.

As can be seen from Code 3.3,which shows Jamboree in pseudocode form, the algorithm consists of three main parts. The first is a full window search of the first child position, to give an initial reasonable value for α. Similar to $\alpha\beta$, the efficiency of the Jamboree algorithm is very dependent on the initial move ordering of child positions. If the score for the first move examined is close to optimal it should allow subtrees of the other moves examined to be rapidly pruned and discarded. If it is a very poor score, it will have a detrimental effect on the next part of the algorithm, the "testing" part.

With the initial α value, all the other moves are tested by applying the Jamboree algorithm to them with a zero width search window. This test should be much quicker than a normal full window search and will show whether each remaining move is better or worse than the initial move. Moves which testing showed could be better than the initial move are searched

```
score Jamboree(position, alpha, beta) {
    if(isLeaf(position))
        return StaticEvaluation(position);
Part I
    // Full test of first move
    legalMoves = GenerateLegalMoves(position);
    bestScore = -Jamboree(legalMoves[0], -beta, -alpha);
    if(bestScore >= beta)
        return bestScore;
    if(bestScore > alpha)
        alpha = bestScore;

Part II
    // Parallel testing of moves with zero window search
    forall move in legalMoves) {
        move.score = -Jamboree(move, -alpha-1, -alpha);
        if(move.score > bestScore)
            bestScore = move.score;
        if(move.score >= beta)   // "abort-and-return". Destroy any
            return move.score;   // subcomputations underway.
    }
Part III
    // Serial search of valid moves.
    foreach move in legalMoves {
        if(move.score > alpha)
        {
            move.score = -Jamboree(move, -beta, -alpha);
            if(move.score >= beta)
                return move.score;  // Abort-and-return
            if(move.score > alpha)
                alpha = move.score;
            if(move.score > bestScore)
                bestScore = move.score;
        }
    }
    return bestScore;
}
```

Code 3.3: *Jamboree algorithm.*

sequentially in the normal manner, i.e., full window search and the best score returned. If the initial move ordering is poor, many moves will "pass" the testing phase and must later be searched sequentially.

The Jamboree algorithm gambles that enough moves will be eliminated during the testing phase to outweigh the cost of searching some nodes twice (zero and full window searches) and empirical evidence suggests that it usually wins[52].

3.4 $\mathcal{CG}$ Chess

The following subsections describe the $\mathcal{CG}$ representation of the Jamboree algorithm presented above. The algorithm has been slightly modified to return not just a score for a given board, but a move object representing the best move. This object contains amongst its attributes an evaluation field which contains the score for this move.

3.4.1 *PickMove*

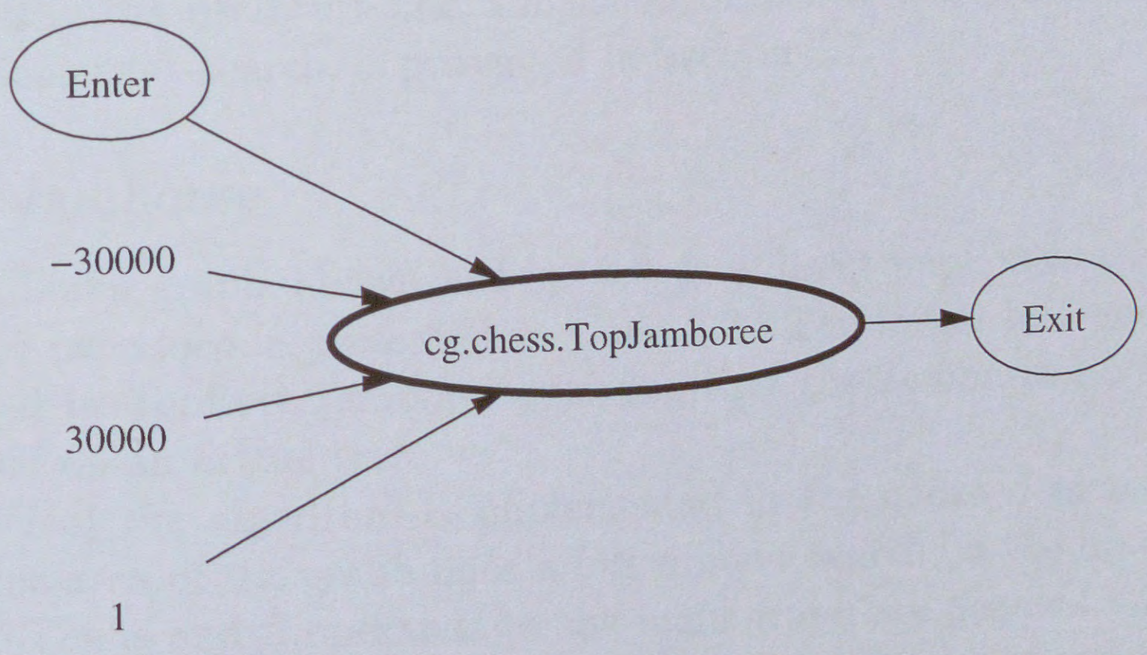


Figure 3.7: *PickMove - Top level graph. Adds initial values to a call to TopJamboree*

PickMove, depicted in Fig. 3.7, is the top level graph in the $\mathcal{CG}$ chess application. It is simply a wrapper around the *TopJamboree* node, adding initial α, β and depth values to the user defined board operand.

3.4.2 *TopJamboree*

The *TopJamboree* graph – see Fig. 3.8 – performs a depth check against a maximum depth constant defined internally. For the experiments in this section the depth was limited to 4 plies. Since the average number of moves available is 35, this implies a tree containing over 1,500,000 leaf nodes to be searched and provides more than enough parallelism to exercise the WebCom

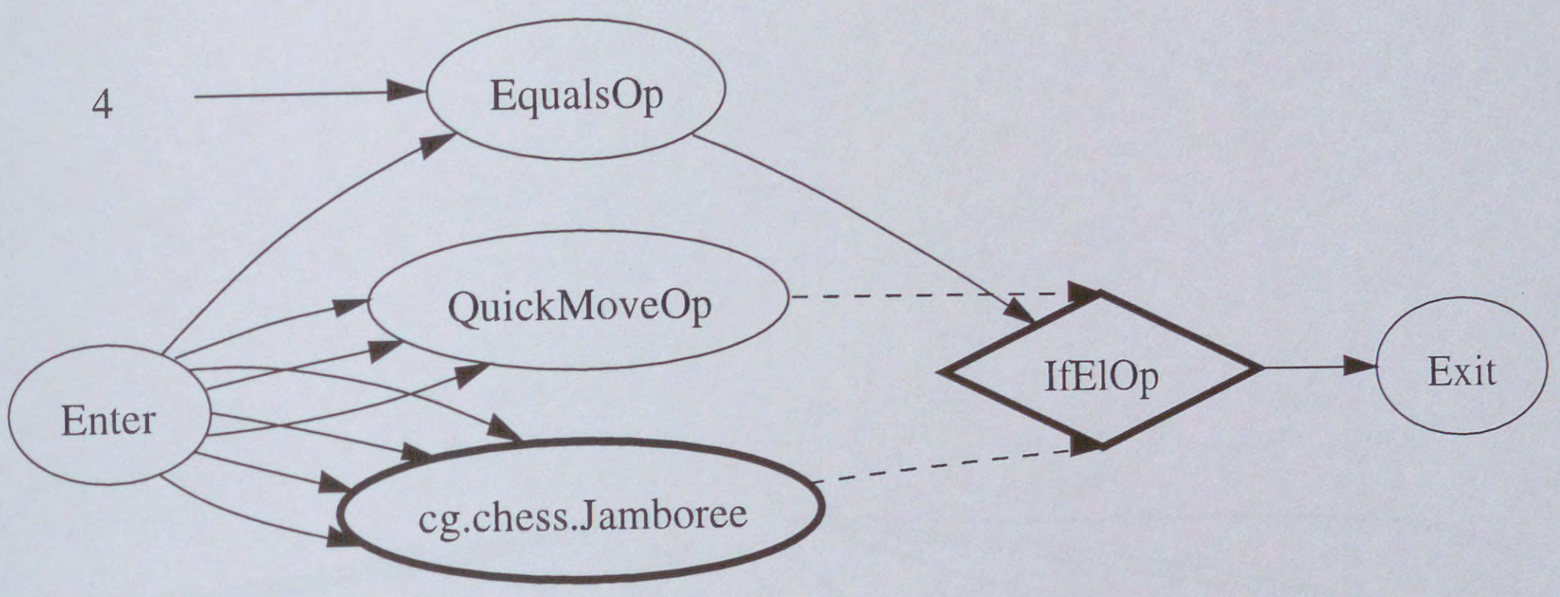


Figure 3.8: *TopJamboree - Performs a simple depth check before calling the Jamboree graph.*

system. This simple bounding mechanism has the advantage of helping to rigorously limit the work the application will perform but suffers from "horizon" effects, where a bad move may hide a much greater threat that lies just beyond the current search tree's depth. Again, this issue has been addressed in the literature and a mechanism to help avoid such situations, the "Quiescence" search, is presented in Section 3.7.

3.4.3 *Jamboree*

The *Jamboree* graph, shown in Fig. 3.9, is a direct implementation of the Jamboree pseudocode presented in Code 3.3. The initial leaf node test is performed in *TopLevelJamboree* ensuring that the Jamboree node will be performed on an actual tree.

Part I of the algorithm is implemented in the nodes depicted in Fig. 3.10. This area of the graph does a full window search on the first move in the list. The α and β operands to the main graph are negated and passed to a new *TopJamboree* node as β and α respectively. An incremented depth count is passed in as is a new board derived from the board operand to *Jamboree* and the first move from a list of legal moves derived from that board. The *TopJamboree* node returns a move object, the score of which is then negated for use in the subsequent parts of the algorithm. Parts II and III of the algorithm are performed by condensed nodes representing the graphs *TestMoves* and *SerialSearch*(Fig. 3.11).

3.4.4 *TestMoves*

TestMoves, depicted in Fig. 3.12, performs a zero window search on the list of moves given as an operand. Each score returned is set as an attribute of the appropriate move object. The test is simply to determine whether the move is better or worse than the first move in the list, known by whether

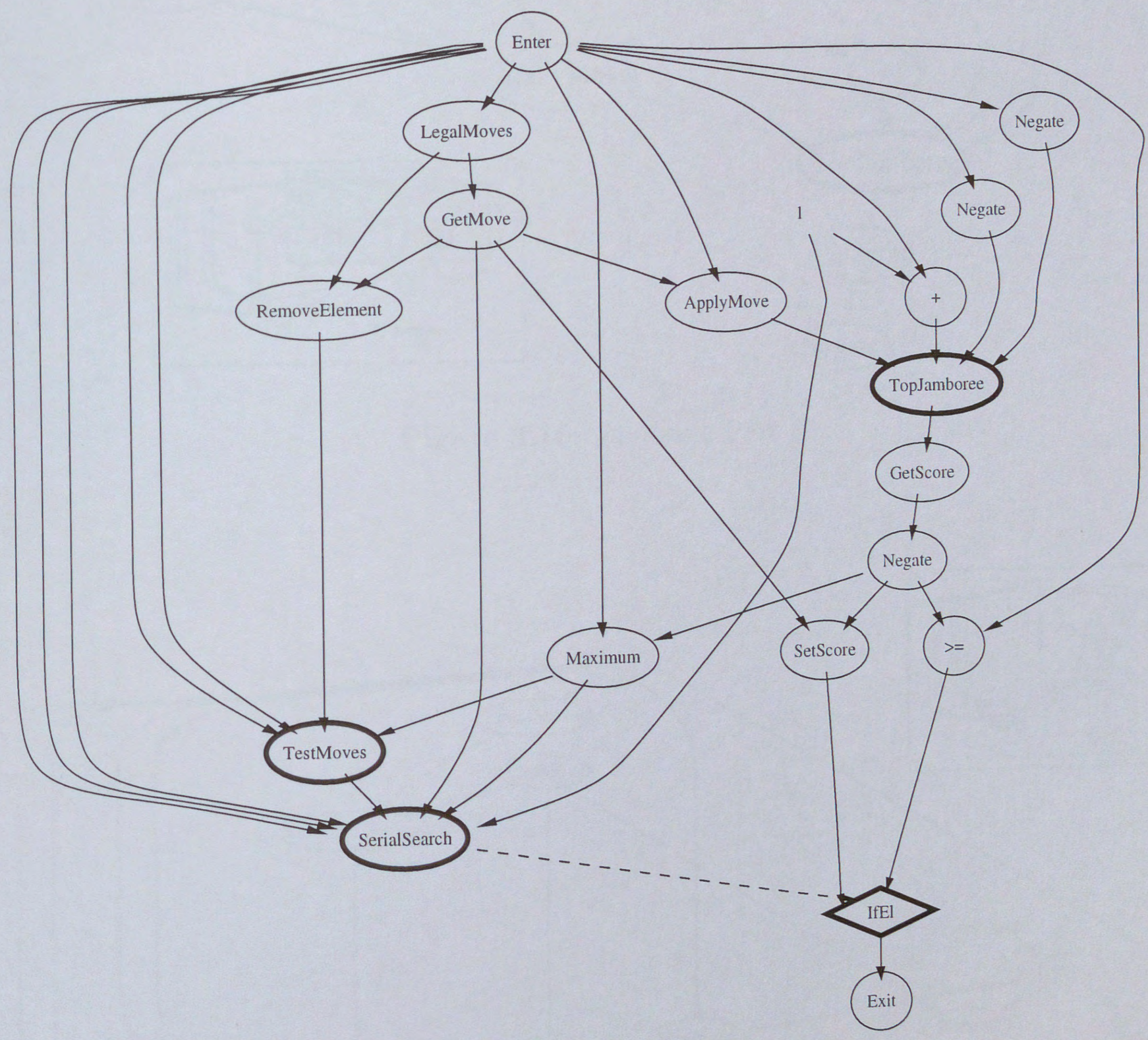


Figure 3.9: *Jamboree*

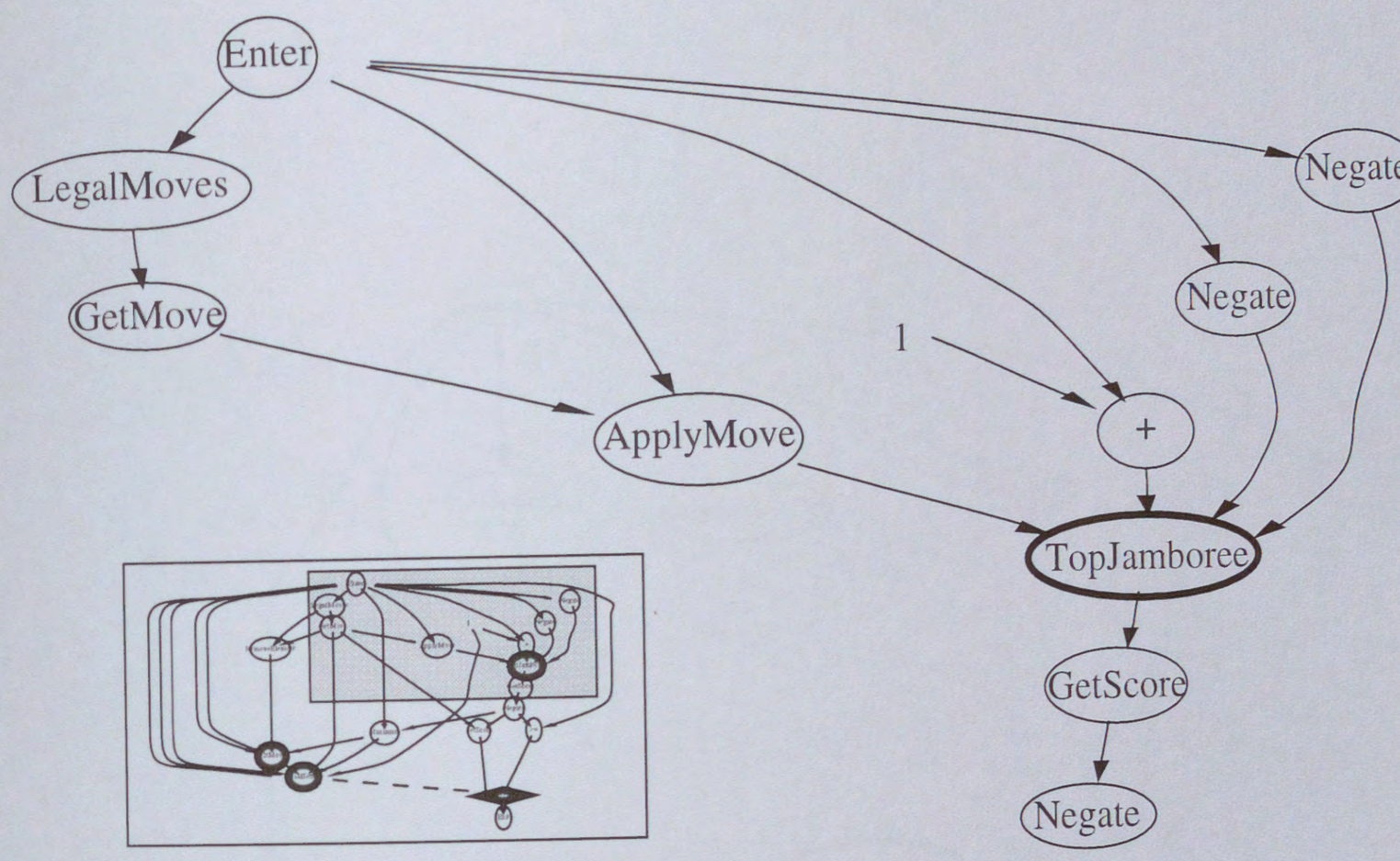


Figure 3.10: *Jamboree Part I*

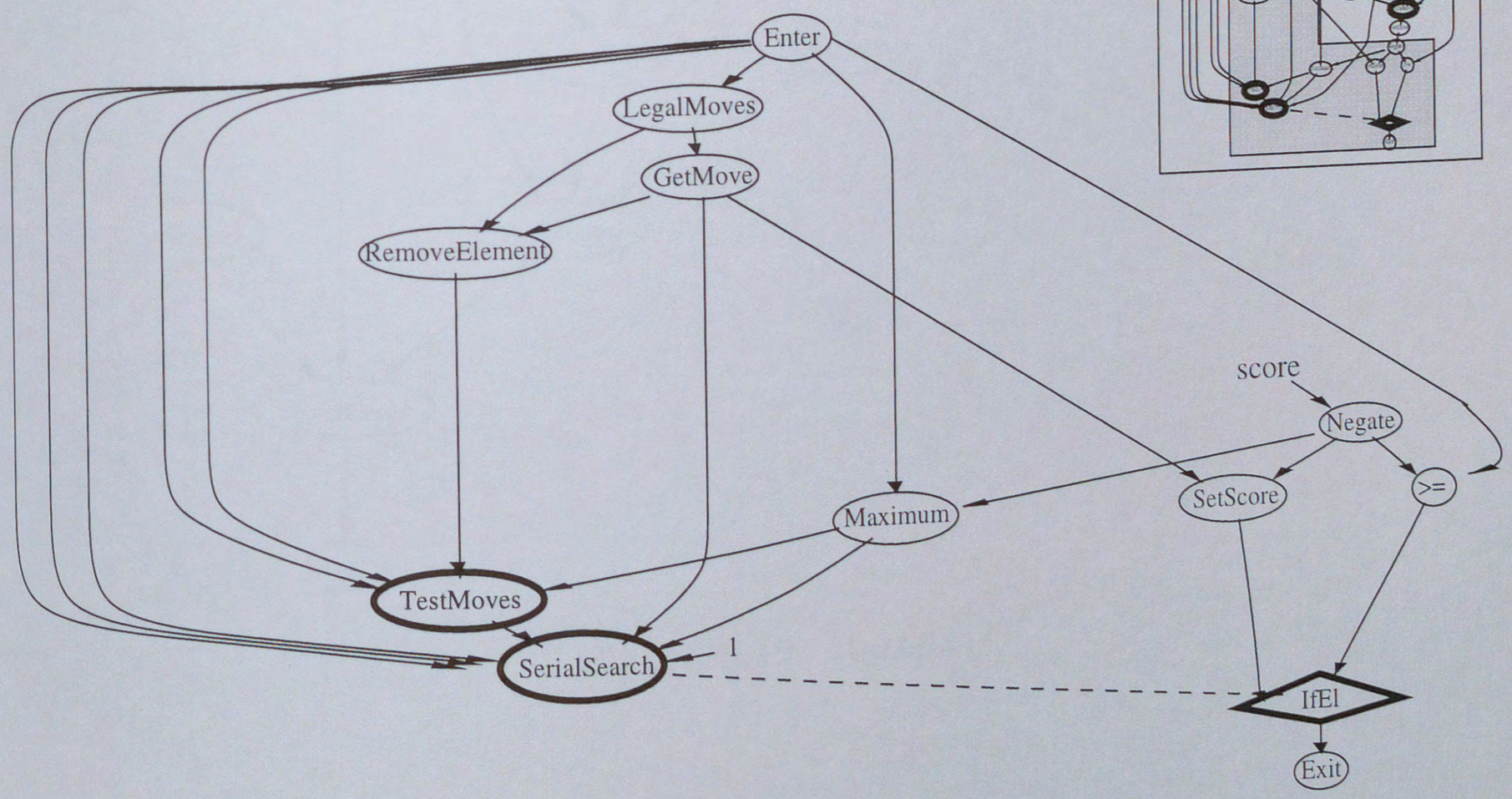


Figure 3.11: *Jamboree Parts II & III*

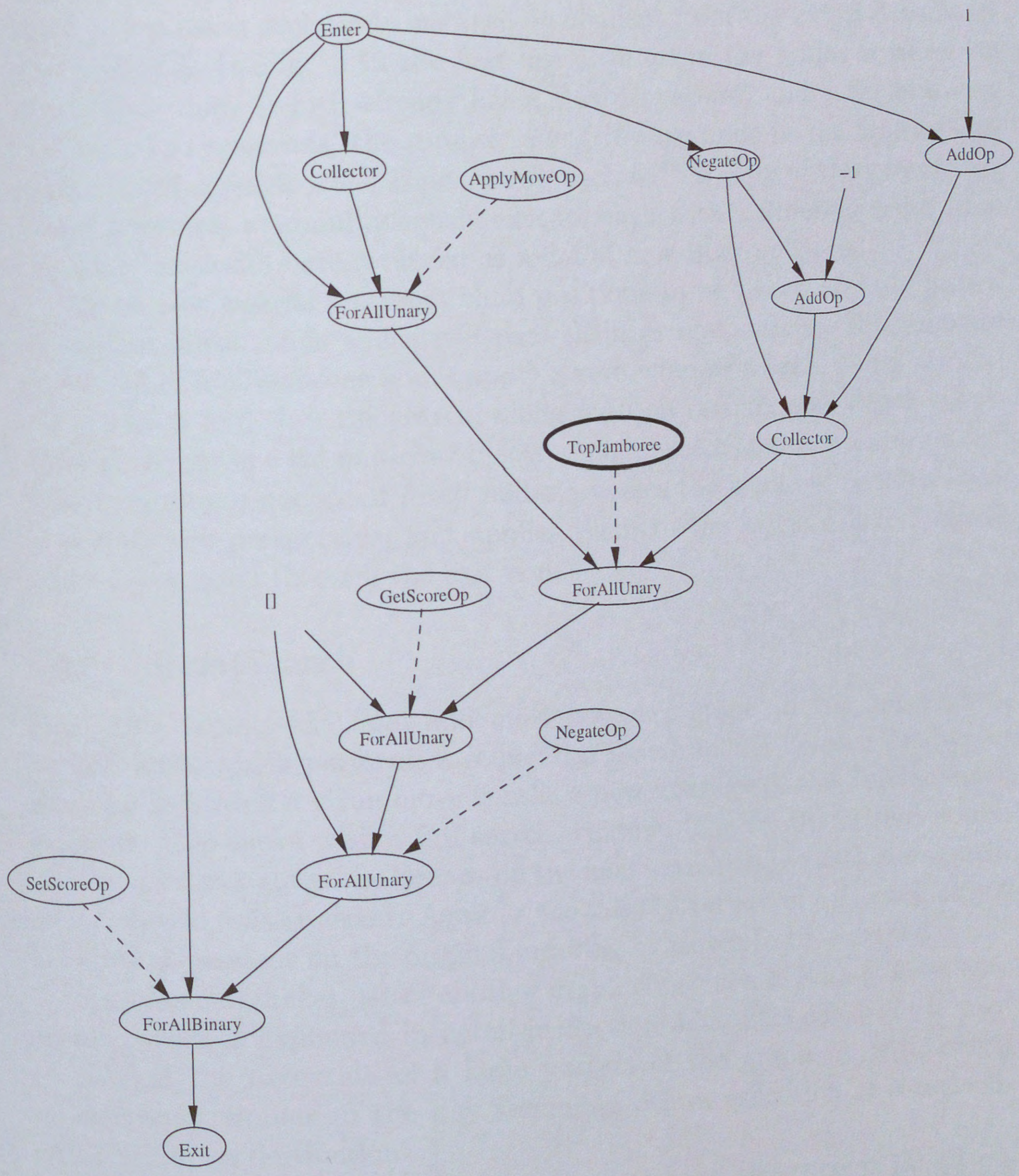


Figure 3.12: *TestMoves*

a (hopefully quick) zero-window search fails high or fails low. A complete search of the moves that "passed" the test takes place in the next graph, *SerialSearch*. *TestMoves* makes extensive use of the *forall* construct, which is explained in detail in Section 3.5 and is represented in the graph by *ForAllUnary* and *ForAllBinary* nodes. A *forall* node implements a parallel loop by taking a list of data and a $\mathcal{CG}$ node as its operands, replicating the node and passing each node instance an element from the list of data as its first operand. In Fig. 3.12 the first use of *forall* in the graph is where an *ApplyMove* node (which already has a Board operand) and a list of moves are passed as operands, the topmost *ForAllUnary* node in the figure. This node creates enough *ApplyMove* nodes, each with a copy of the preexisting Board operand, to simultaneously execute each with a different move from the list. The results are returned as a list of new Board objects.

These new boards represent child positions to be tested and are passed to another *forall* node which will pass them as operands to *TopJamboree* nodes. The *TopJamboree* nodes apply a zero window search, using $-\alpha$ and $-\alpha - 1$ as α and β in the search, along with an incremented depth count. This will result in a list of Move objects, each of which contains a move score. The remaining sequence of *forall* nodes accesses those scores, negates them to correct the perspective, and applies them to the original list of moves which is returned through the exit node.

3.4.5 *SerialSearch*

Fig. 3.13 depicts the final and most complex graph in the application, *SerialSearch*, which performs a sequential search of the "passed" moves in the list. To search a given move it calls a new instance of the *TopJamboree* graph on that move with a full search window. The results of that search may be used to tighten the bounds on the next search and so each subsequent search should finish quicker. Again, since this is a variant of $\alpha\beta$, its efficiency is highly dependent on the original ordering of moves to be searched.

SerialSearch makes use of another graph construct, a *Pass-If-False* construct, which is explained in detail in Section 3.5. This construct is used to prevent the execution of a large portion of the graph by withholding its operands, similar to the way stemming delays execution of a node by withholding its destination.

The graph takes the following as its operands: the current Board position, the list containing moves to be searched, the current depth, the α and β bounds, a list index which specifies the move to be searched, and the best move so far. The first decision made by the graph, i.e., the *ifel* node connected to the *exit* node, in Fig. 3.13, is a check to ensure there is another move left in the list. If there is not, then the *Best So Far* graph operand is returned. If there is a move left, its score is checked against the current α and if it has passed the *TestMoves* test it is searched in turn.

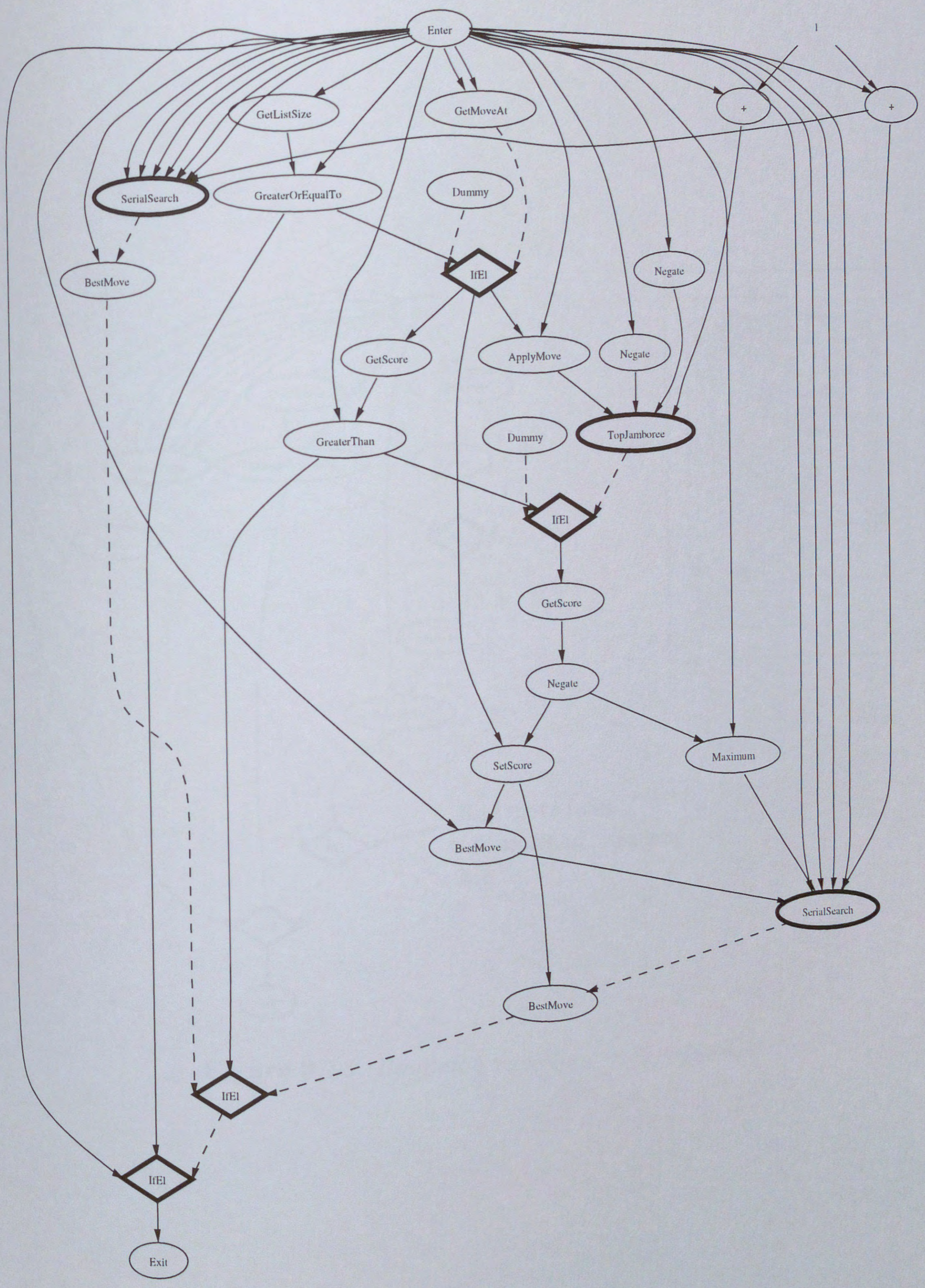
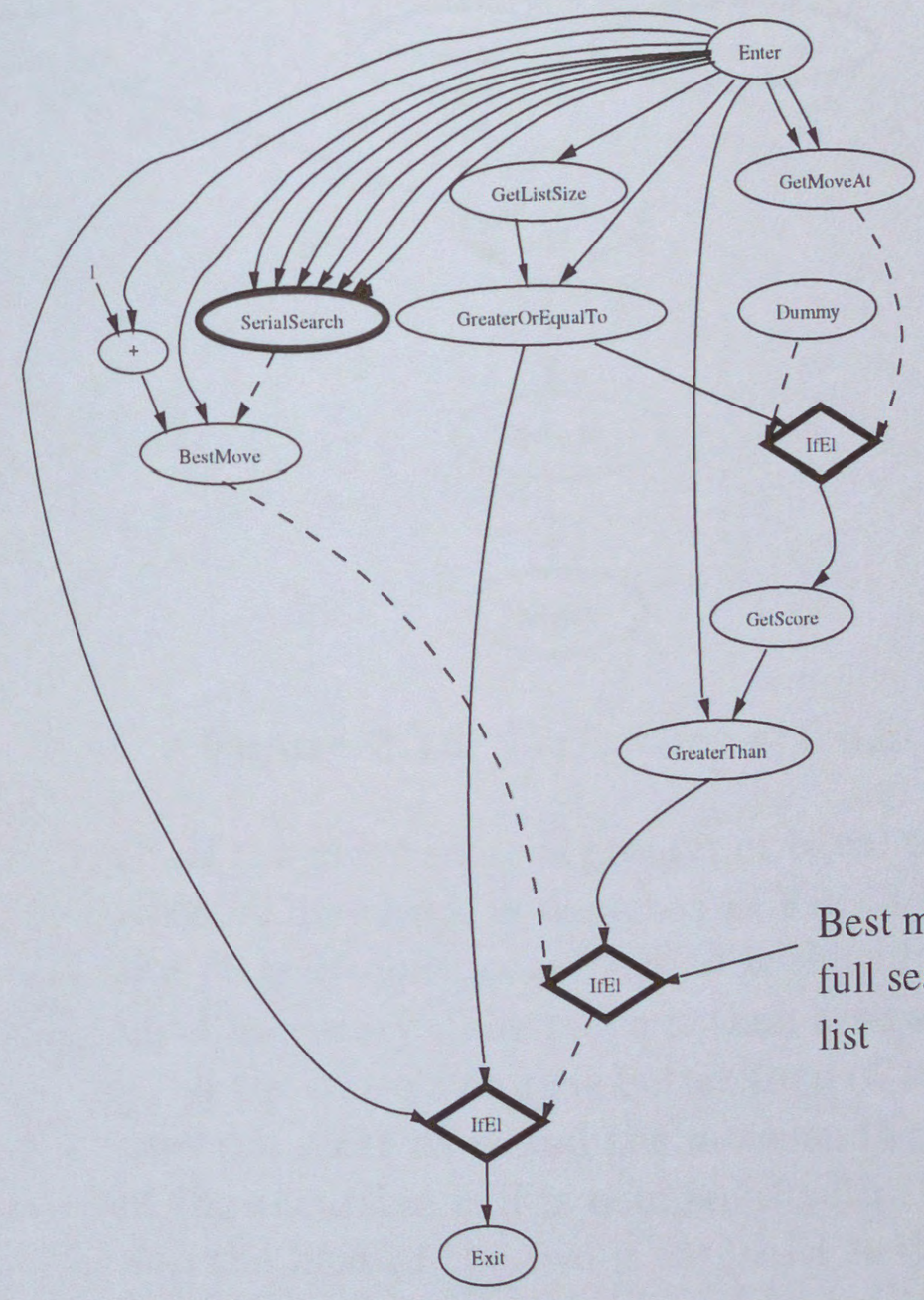


Figure 3.13: *SerialSearch*



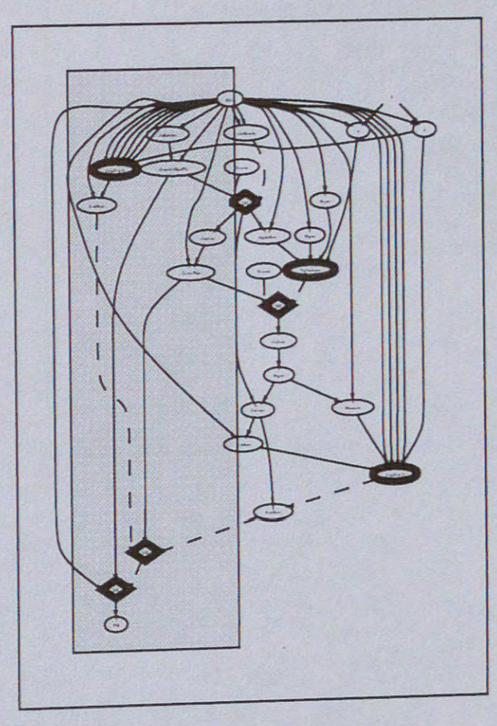
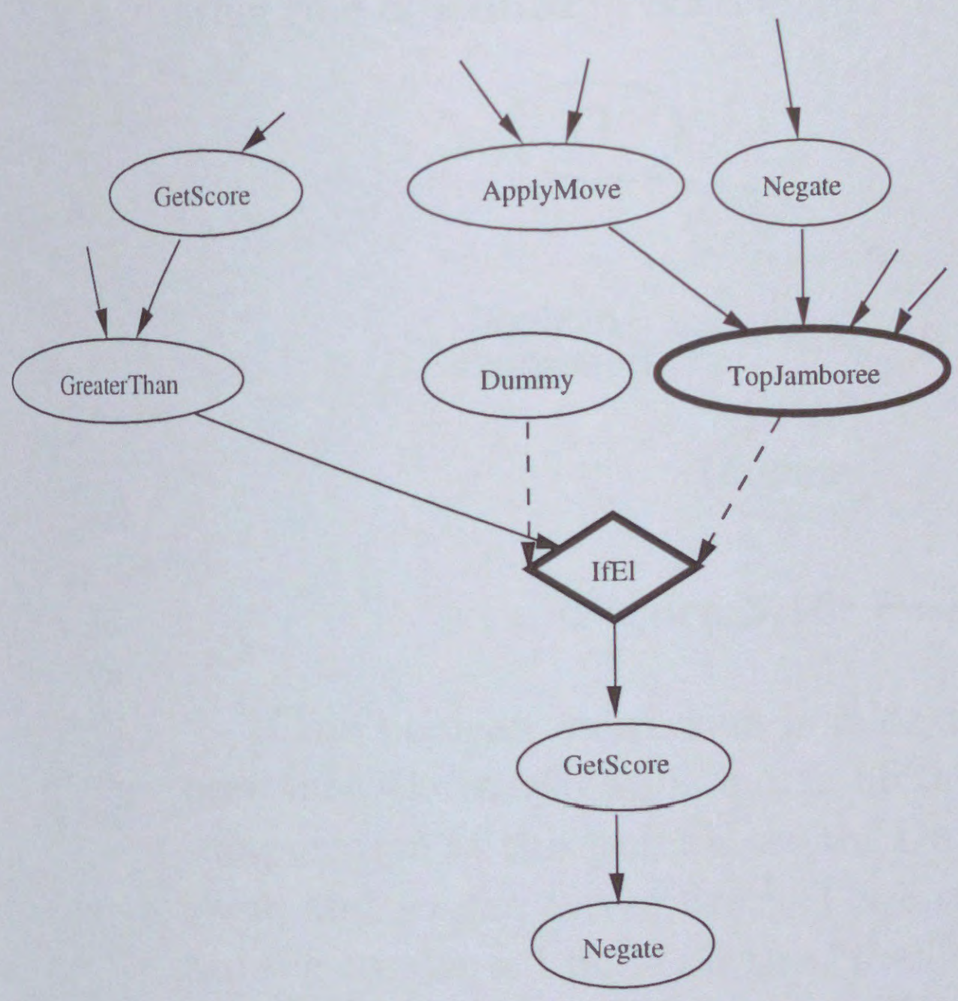


Figure 3.14: *Bounding recursion in* SerialSearch.

If the move score does not "pass", then the result of a new *SerialSearch* node is returned. With the exception of an incremented list index, all operands to the recursive call are unchanged. Fig. 3.14 highlights this area of the graph.



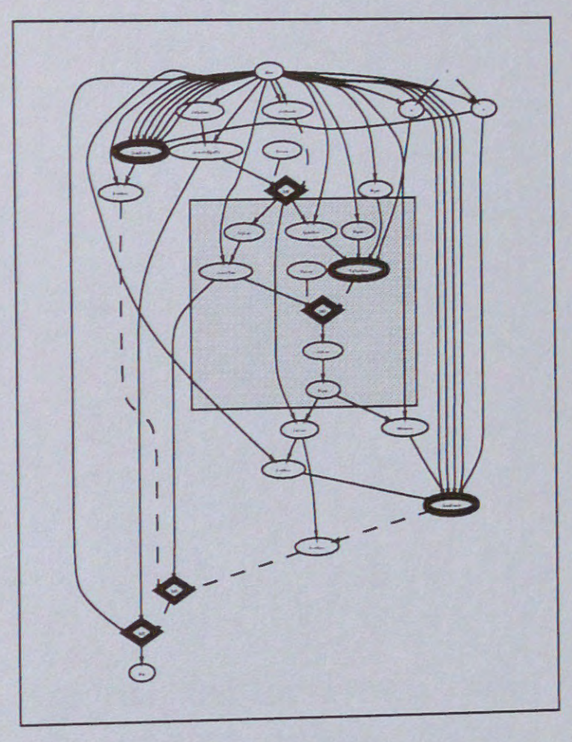


Figure 3.15: *Performing a* TopJamboree *search on valid position.*

If the move score is greater or equal to α it is searched by a *TopJamboree* node. This check is depicted in Fig. 3.15, which also highlights the use of the *Pass-If-False* construct to withhold execution of the *TopJamboree* call until necessary. The move is then assigned its new score.

If the move's score is better than α, another recursive call to *SerialSearch* uses the score as α and the move as the new *Best So Far* move. The result of the recursive call is compared with the move examined by this instance and the best of the two is returned as the overall result.

3.5 New $\mathcal{CG}$ Constructs

During the development of the $\mathcal{CG}$ Chess application, some new constructs were developed.

3.5.1 Pass-If-False

The first of these is the *Pass-If-False* construct, see Fig. 3.16. The *Pass-If-False* is a variation on the standard *ifel* node which does nothing if its boolean expression operand evaluates to false. This is accomplished in the model by the use of an *ifel* node and a "dummy" node which has one operand

port, one operation port and one destination port. The boolean operand to the *ifel* node and the "then" operand are set as normal. The "else" port of the *ifel* node contains a stemmed connection to the *Dummy* node. This allows the *ifel* node to form a complete triple and fire according to the $\mathcal{CG}$ firing rule described in Section 1.3.

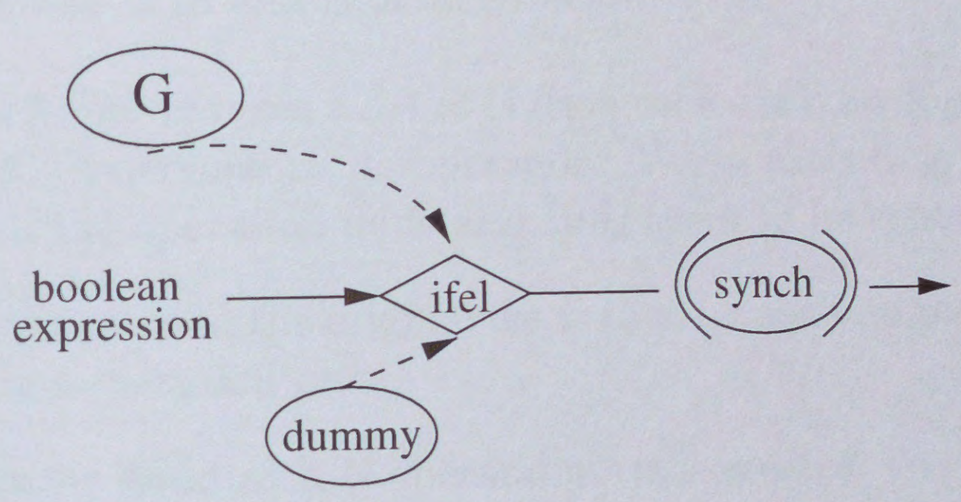


Figure 3.16: *Pass-If-False construct*

If the boolean expression is false, the *ifel* node will copy its destination port into the destination ports of the Dummy node. This will halt the computation at this point since the Dummy node will not receive an operand value and so can never fire.[1] The overall effect of this is to conditionally fire the condensed node on the "then" port of the *ifel* node. This alleviates the need for stemming all connections in a critical path as shown in Fig. 3.17 and in doing so avoids some of the complications discussed in Section 3.5.3. Examples of the pass-if-false if can be seen in the *SerialSearch* graph in Section 3.4.5.

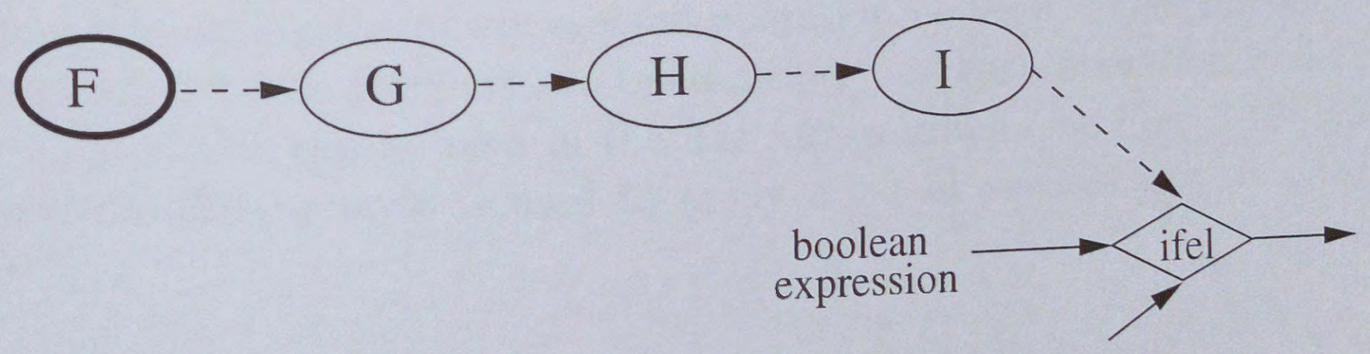


Figure 3.17: *If Node* ***F*** *is a very expensive computation then the programmer may wish to execute* ***F*** *only if needed, i.e., the boolean expression to* ifel *is true. To achieve this, all destinations must be stemmed from the* ifel *node back to Node* ***F****. This leads to complications if any node in that path has more than one destination.*

[1] A lack of operator would also prevent the Dummy node from firing. However, in the WebCom implementation the Dummy node has a special *DummyOp* operator for logging and auditing purposes and so it is critical that the dummy node have no operand value.

3.5.2 ForAll

The parallel loop in the Jamboree algorithm is implemented by a *ForAll* loop construct. The *TestMoves* graph (Fig. 3.12) utilises two versions of this node, *ForAllUnary* and *ForAllBinary*. The execution of the *ForAll* node is depicted in Fig. 3.18 and is detailed below:

1. The ForAll node takes a list of N data values and an "operation" node (Node **G**) reference as its operands. When executing, it makes N-1 copies of the operation node and adds them to its container graph.

2. Each data value in the original list is then copied into the first operand port of an operation node.

3. A *Collector* node with N operand ports is created.

4. Each of the **G** nodes is given one of these operand ports as its destination. Each **G** Node is now fireable.

5. The Collector node is given a copy of the *ForAll* node's destinations which are then deconstructed.

6. When all of the Collector node's operand ports have result values, it executes and its operator collates the results into a single list. This list of results is then passed to its destinations a single result value.

A variation on the *ForAll* node is the *ForAllBinary* node. This node takes three operands, an operation node as before and two lists of data. When populating the newly created operation nodes it copies one element from each list into the first two operand ports of each operation node. An example of this can be seen in the *TestMoves* graph (see Fig. 3.12) where the *ForAllBinary* node is used to apply a list of move scores to a list of moves.

3.5.3 Synch Node

An issue that arises in the conditional execution of nodes in the graph is whether to re-execute a node if its result is required at different points in time. Fig 3.19 provides an example situation where there are two stemmed connections to Node **F**. These nodes may each graft onto **F** at different times. When the first grafts on, **F** becomes fireable as normal and executes. Once it has finished executing, its destination port is deconstructed, returning the node to its pre-execution state. When Node B grafts on to **F**, this whole process is repeated. If a node is referentially transparent, it is desirable to return the already calculated result instead of recalculating it a second time.

A simple way to achieve this optimisation without altering the model is to add a *synch* node to the graph between Node **F** and its destinations, as

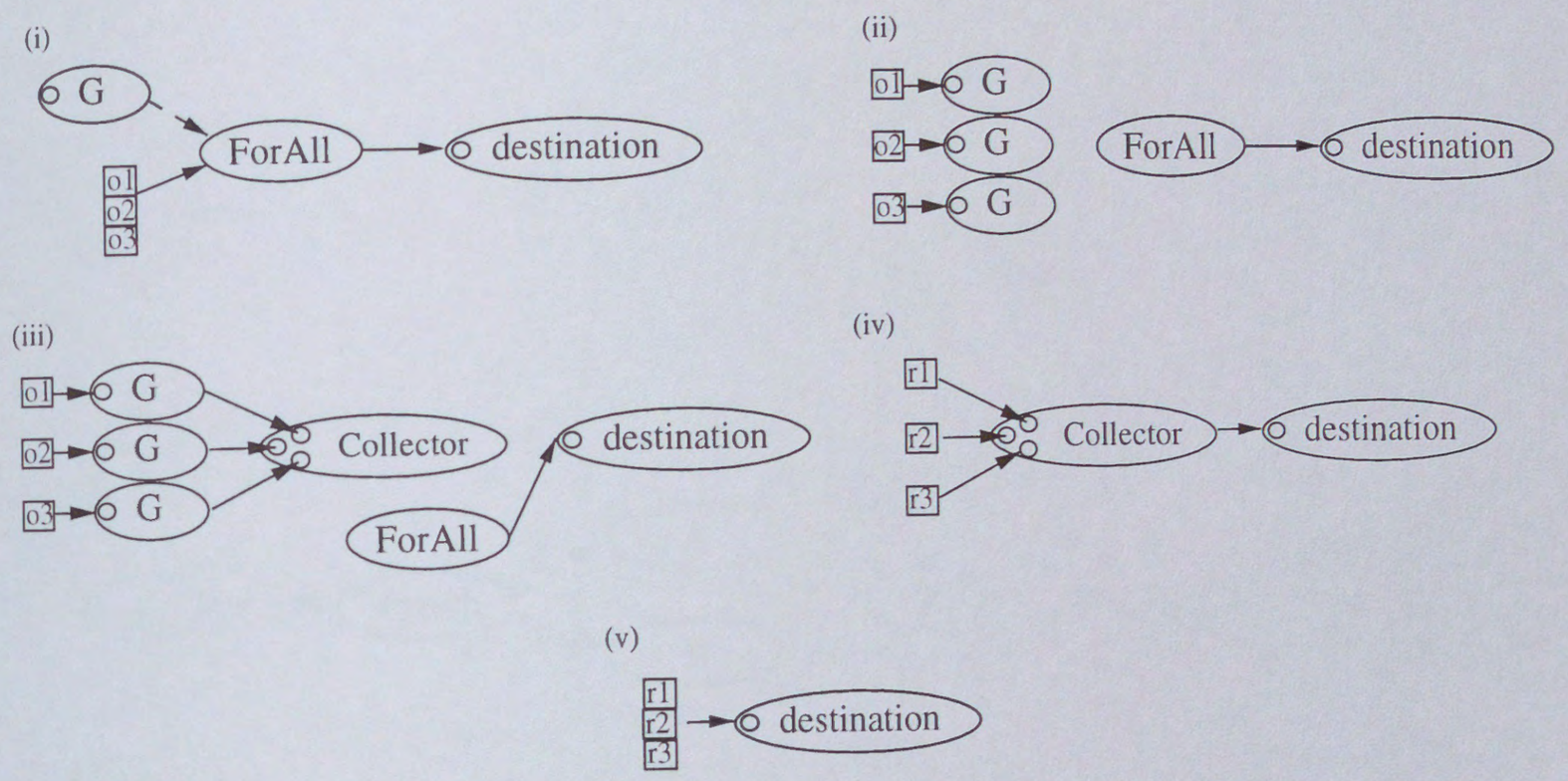


Figure 3.18: *Execution of the* ForAll *node. (i) A graph containing an executable* ForAll *node. (ii) Enough copies of Node* **G** *are created to match one to each of the N operands in the list. (iii) A* Collector *node is created with N operand ports acting as destinations for the* **G** *nodes which are all fireable at this point. (iv) The* Collector *node is given a copy of* ForAll*'s destinations as* **G** *nodes execute to provide operands for* Collector*. (v) When all of* Collector*'s operand ports are filled it fires, collating the results from the* **G** *nodes into a list which is passed on to its destinations.*

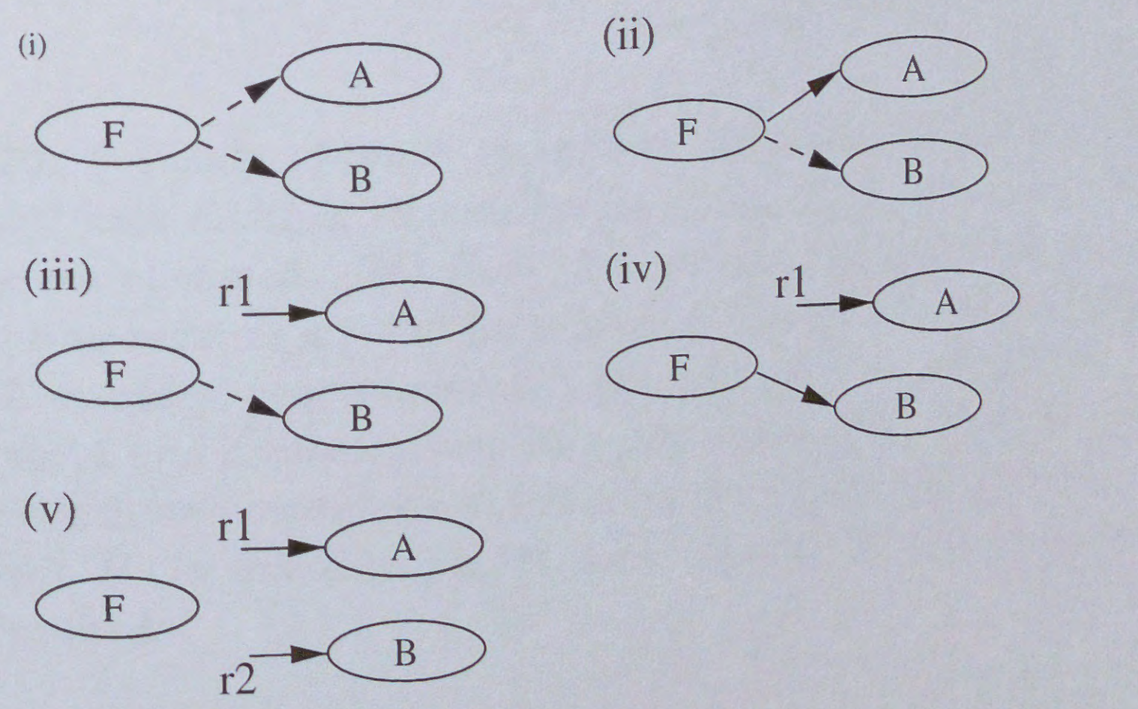


Figure 3.19: *Multiple stemmed connections scenario: (i) Nodes* **A** *and* **B** *both have stemmed connections to Node* **F***. (ii) Node* **A** *requires a result from Node* **F** *and so gives its operand port to Node* **F** *as a destination. (iii) Node* **F** *fires, producing result* **r1** *and deconstructing its destinations. (iv) Node* **B** *now requires a result from Node* **F** *and gives* **F** *a new destination. (v) Node* **F** *fires again, producing a new result,* **r2***.*

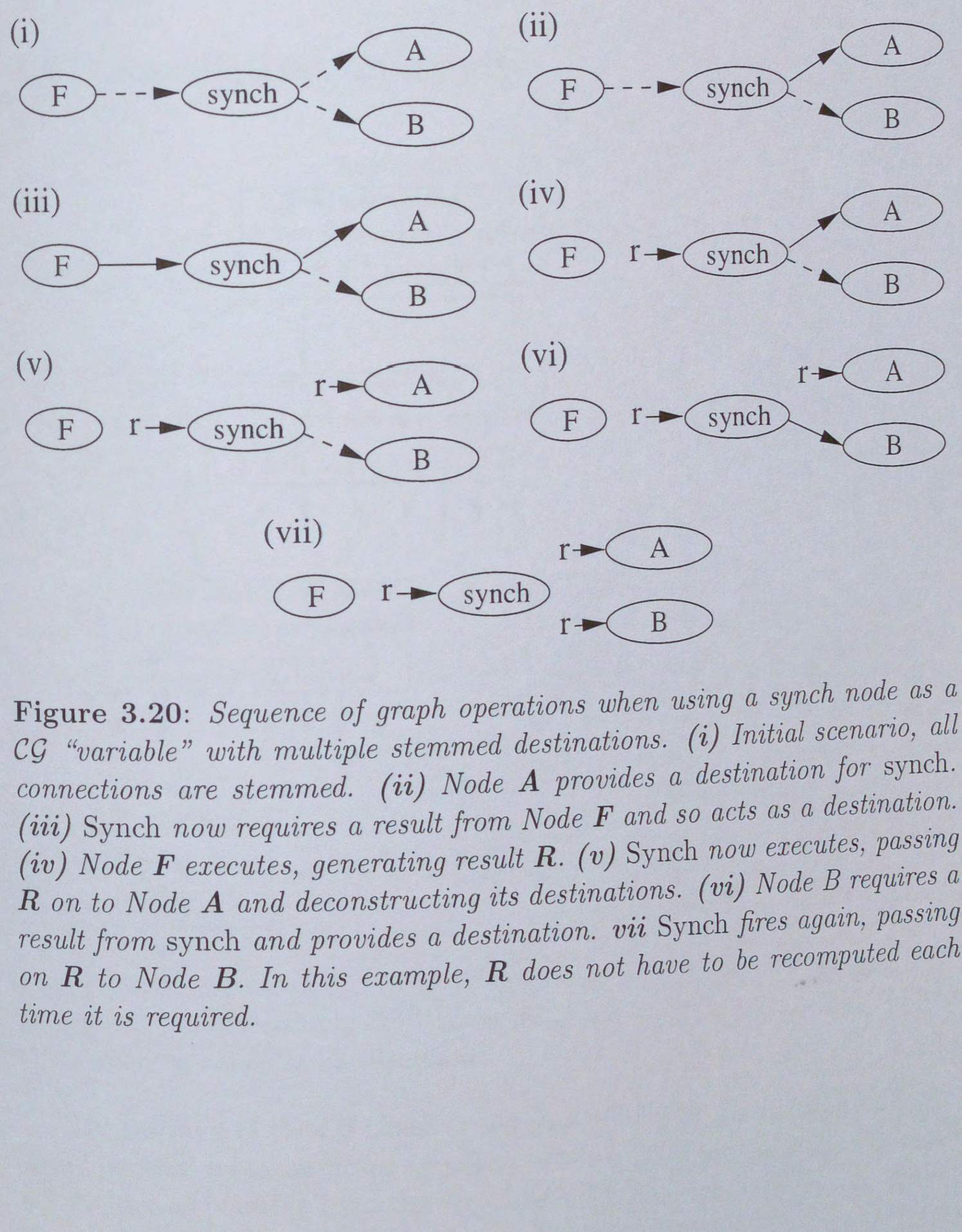


Figure 3.20: *Sequence of graph operations when using a synch node as a CG "variable" with multiple stemmed destinations.* ***(i)*** *Initial scenario, all connections are stemmed.* ***(ii)*** *Node* ***A*** *provides a destination for* synch. ***(iii)*** Synch *now requires a result from Node* ***F*** *and so acts as a destination.* ***(iv)*** *Node* ***F*** *executes, generating result* ***R***. ***(v)*** Synch *now executes, passing* ***R*** *on to Node* ***A*** *and deconstructing its destinations.* ***(vi)*** *Node B requires a result from* synch *and provides a destination.* ***vii*** Synch *fires again, passing on* ***R*** *to Node* ***B***. *In this example,* ***R*** *does not have to be recomputed each time it is required.*

depicted in Fig. 3.20. The function of the *synch* node is to simply copy its single operand value to each of its destinations. In this way, referentially transparent nodes which need only execute once can have their result stored in the *synch* node itself, which is acting a variable in the graph. Each time the result is required, the Synch node refires, passing on the result to destinations as they arrive. For nodes that are not referentially transparent (e.g., time dependent nodes), stemmed arcs connect directly with the **F** node and cause its refiring each time a graft takes place.

3.6 Parallelisation Study

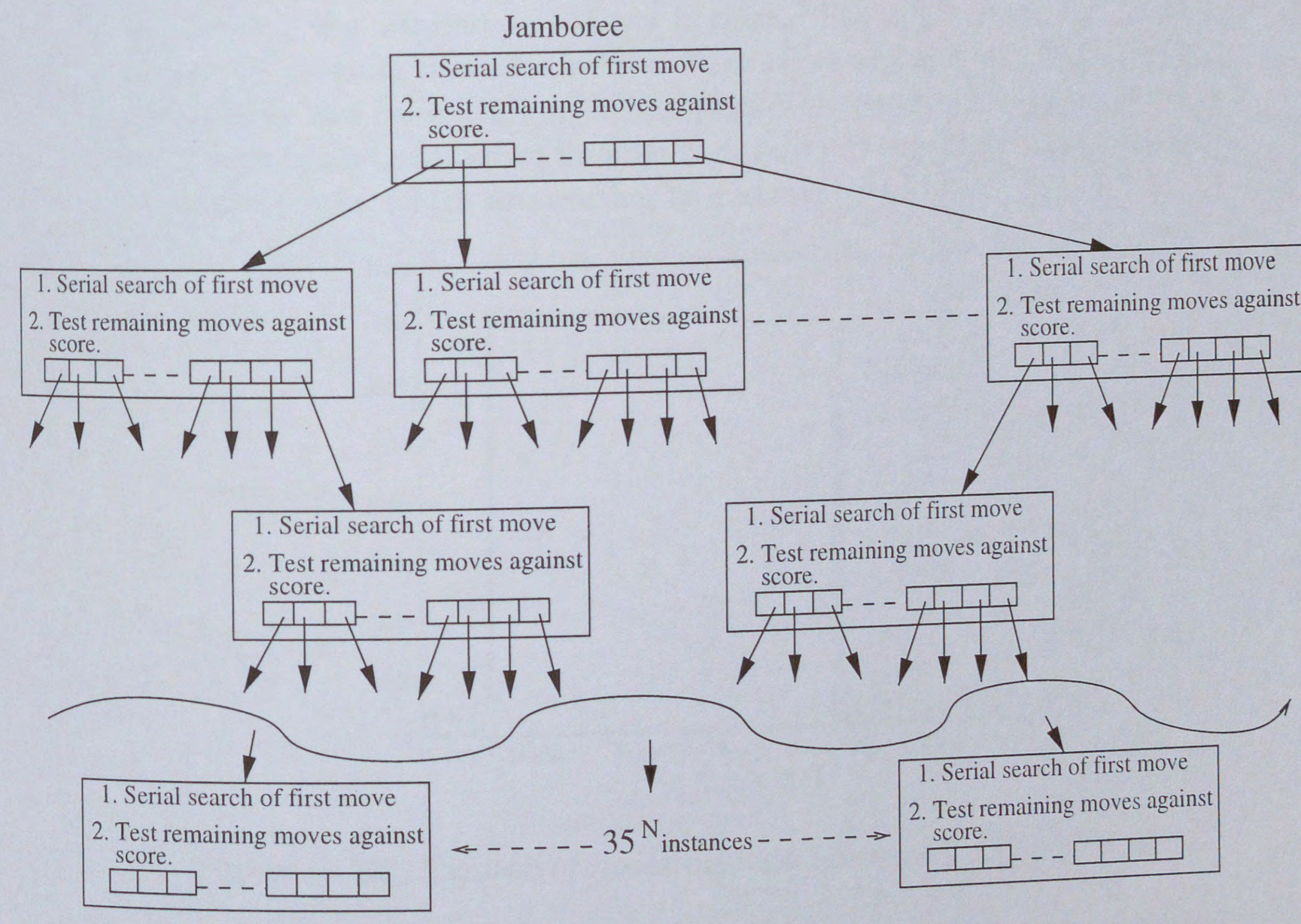


Figure 3.21: *Recursive calls while testing moves expose a large amount of parallelism(approximately 35^N where N is the depth of the search tree) in the Jamboree algorithm's execution.*

The purpose of the $\mathcal{CG}$ Chess application is to exercise a potentially massively parallel metacomputer and therefore it is desirable to extract as much parallelism as possible from the algorithm. This section attempts to predict how a dynamically reconfigurable parallel platform might be expected to execute the Jamboree algorithm. As described above, the main parallelisation step with the Jamboree algorithm is at the point where moves are

tested by searching them with a zero width search window. There will be, on average, thirty five branches from a given node in the search tree and all of these may be tested simultaneously. Each of these sub computations will test their own children, and so on, exposing a large amount of parallelism (Roughly 35^N where N is the number of plies in the search tree).

Fig. 3.21 shows a partial snapshot of a Jamboree execution. It is important to note however, that this execution "stack" will be followed by a sequential search of each move that "passes" the test, but each sequential search will employ the Jamboree algorithm on its children resulting in a new exposure of parallelism.

To test the parallelism exposed during the execution, the application was run on a single WebCom unit and some statistical information recorded. The application was limited to a 4 ply horizon. The application was asked to choose an opening move for White. Since there was only one WebCom unit involved in the computation, instructions were executed sequentially. Fig. 3.22 depicts the number of fireable nodes over time. These are nodes with complete triples which are waiting to execute.

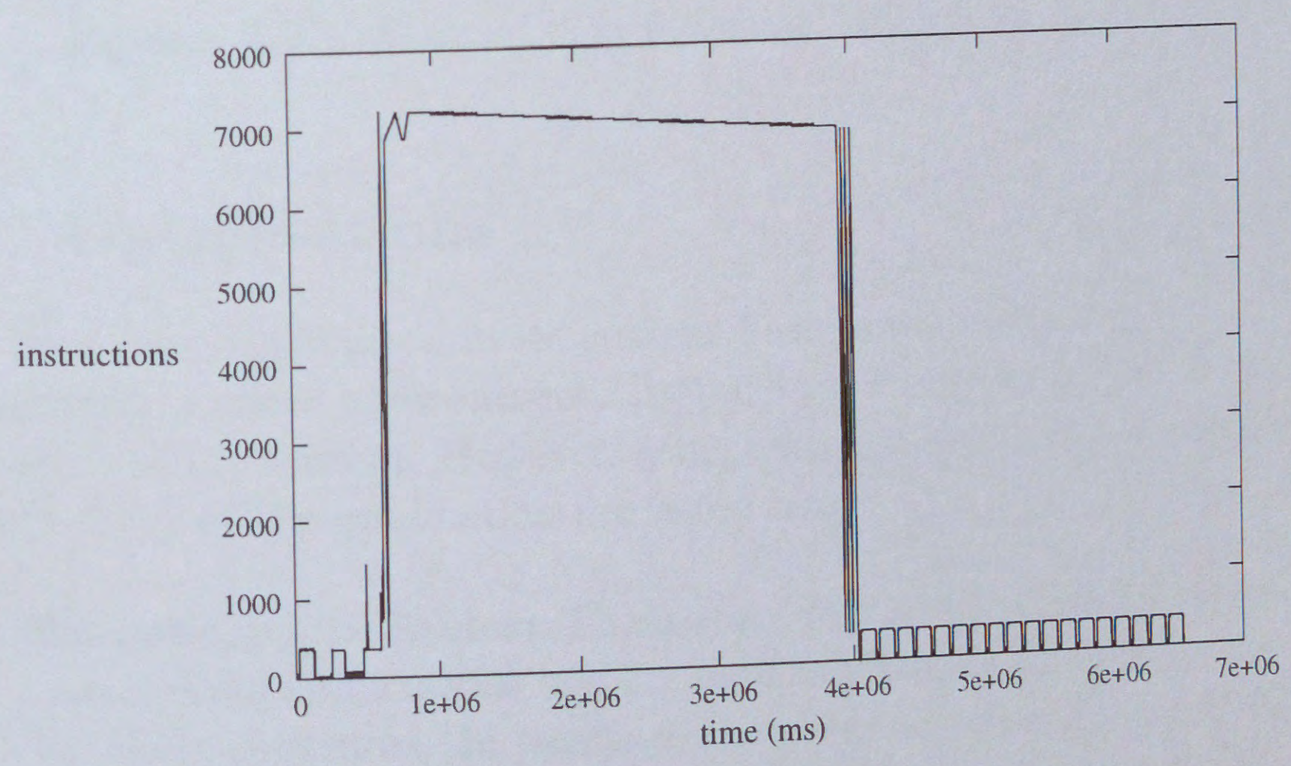


Figure 3.22: *Number of nodes available during CG Chess execution.*

Each measurement was recorded when a newly fireable node was added to the queue. As can be seen from the resulting graph, this number varied from 0 to over 7200. The trend of the graph shows the top level, sequential steps that the application takes. First, a list of possible moves is created and ordered by extremely terse evaluation. Once the list is in place, each move is evaluated. These tasks are distinguished in the graph by instruction counts at or near zero.

The large amount of parallelism initially exposed represents the first search that was forced to search deep to fully evaluate the first move in the list. Searches that occurred after this plateau were able to quickly abort as the first move was indeed the best under the criteria applied. The resulting

move was **1 f3** (see Fig. 3.23).

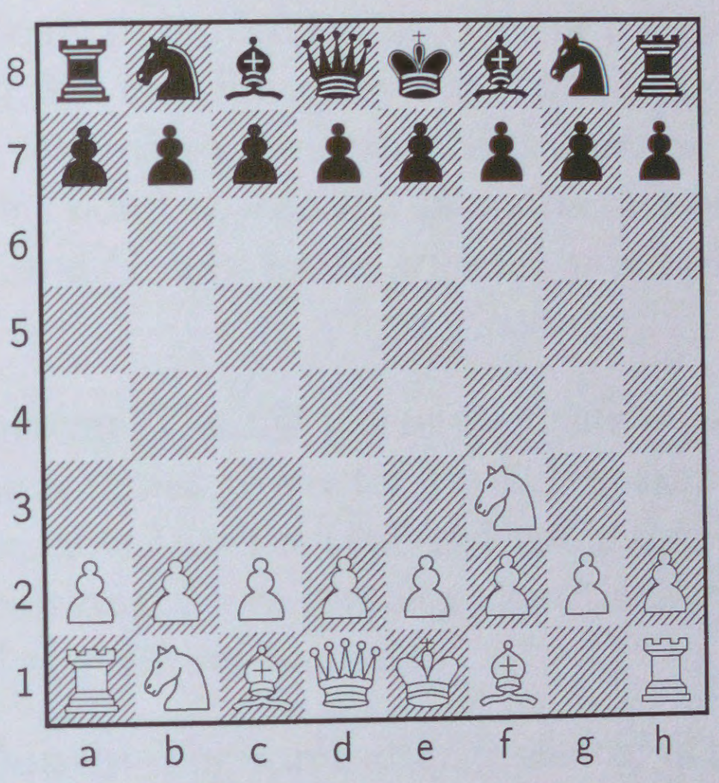


Figure 3.23: *State of play following White's opening move.*

3.7 Optimisations

The $\mathcal{CG}$ Chess application in its current form is too inefficient to do well in a timed tournament environment. Its purpose is only to provide a test bed for the WebCom system. However, some optimisations which could increase the efficiency of the application are listed here.

1. **Transposition Tables:** These are a required feature for modern $\alpha\beta$ based chess applications and are used in virtually all tournament quality chess programs. In practice, the chess search space tends to form a graph, not a simple tree. This is because many board positions may be reached by more than one sequence of moves. A lot of work may be saved by recording searched positions and various pertinent data such as score, depth, whether the score is exact or an upper or lower bound. This list is then consulted before a position in the search tree is searched and, if a match is found, the score may be returned without further searching that subtree. When using iterative deepening, transposition tables become especially useful. "Iterative Deepening $\alpha\beta$"[59] is an $\alpha\beta$ 2-ply search followed by a 3-ply search and so on until a D-ply (where D is the maximum search depth) search is reached. Although this seems very inefficient, results given by the previous iterations tighten bounds and improve move ordering to such an extent that it generally outperforms a single initial D-ply search. Transposi-

tion tables are vital since the same branches may be considered many times in an iterative search.

It is very difficult to implement an efficient transposition table mechanism on a parallel distributed system. Variations of the $\alpha\beta$ algorithm have been developed to use transposition driven scheduling[32]. It is not clear at this point how such a data structure would be implemented in the Condensed Graphs model without introducing WebCom specific mechanisms.

2. **Move Ordering:** The $\mathcal{CG}$ Chess application uses a relatively naive move ordering scheme. Moves for the highest ranking pieces are searched first and these moves are then sub-ordered in terms of capture or threat of capture positions. More complex algorithms exist such as the *killer* heuristic and the *history* heuristic.

3. **No Fixed Depth:** The application presented in this chapter is bounded by an overall depth parameter, as discussed in Section 3.4.2. A tournament application would be better served by searching till a given deadline and returning the best move found at that time. In this way, a part of the algorithm might look more plies ahead than usual and improve the accuracy of the next move evaluations.

4. **MTD(f):** An improved version of $\alpha\beta$, known as mtd(f), combines iterative deepening with aspiration search techniques to perform a number of zero window searches on the moves to be evaluated. Each search either fails "high" or fails "low", allowing a binary search of the Minimax tree to be performed. MTD(f) has been shown to be an extremely efficient sequential algorithm, outperforming all other algorithms it has been compared with[53].

3.7.1 When things go wrong

Since WebCom is a dynamic platform, and the accumulated risk of failure for individual components may be high, some mechanisms are required to allow the WebCom system to deal with various kinds of failures.

For example, a common problem with distributed systems of any kind is the temporary loss of a network link. The ConMan module addresses this problem by immediately attempting to re-establish broken links to connected units. However, if a network link cannot be salvaged, then the ConMan reports the broken connection to the *Fault Tolerance Module*. The Fault Tolerance Module contains logic for first *surviving* faults, and secondly *tolerating* them. This functionality is the focus of this thesis and is discussed in depth in Chapter 4.

Chapter 4

Managing the Unexpected

Due to the dynamic nature of WebCom, and the potential diversity of resources it may harness, it is reasonable to say that it will encounter unexpected behaviour and must react appropriately. A WebCom system may be a complex topology of interacting Units, each comprised of interacting Modules employing various hardware and technological subsystems (see Chapter 2). It is, for all practical purposes, impossible to accurately predict the exact behaviour of all these components. Hence, many factors may impede an application's execution, or may cause it to fail.

Failure may be caused by an incorrect graph construction or by a hardware or network failure. However, they are not limited to these. Since WebCom, in general, does not have full control over every machine participating in a computation, it may interpret various unexpected events as faults. For example, machines may be rebooted, other processes may fill a hard drive or overload a CPU, a local user may suddenly require all of a machine's resources or the machine may be running on a schedule and where it is only available to WebCom for certain time periods. These are *apparent faults*. That is, they may be interpreted as faults but are really just events that slow the progress of an execution.

Some of these events may be minor faults that can be immediately addressed by the appropriate WebCom Module. For example, the ConMan Module incorporates some local routines to attempt to re-establish a broken socket connection. Other apparent faults may have no simple remedy and will have a much larger impact on an application execution. These more severe faults are dealt with by the Fault Tolerance Module.

The following sections outline traditional fault tolerance schemes and how they are utilised in WebCom to address well known distributed system risks. Mechanisms and algorithms developed to minimize the impact of more unique risks added by WebCom's dynamic topology management are also discussed.

4.1 Inherent Mechanisms

The distributed execution of graphs optimises execution costs by exploiting parallelism, but also adds to the complexity of the system by introducing network technology and its associated fallibility. Breaks in physical wires, latency caused by variable network traffic loads plus the additional hardware failure risk associated with network equipment such as NICs and switches all greatly increase the chance of a fault in the distributed computation. These problems are not new to WebCom and have been addressed by many other systems and in the literature through such techniques as message acknowledgements, checkpointing with rollback (see Section 1.2) and redundant executions.

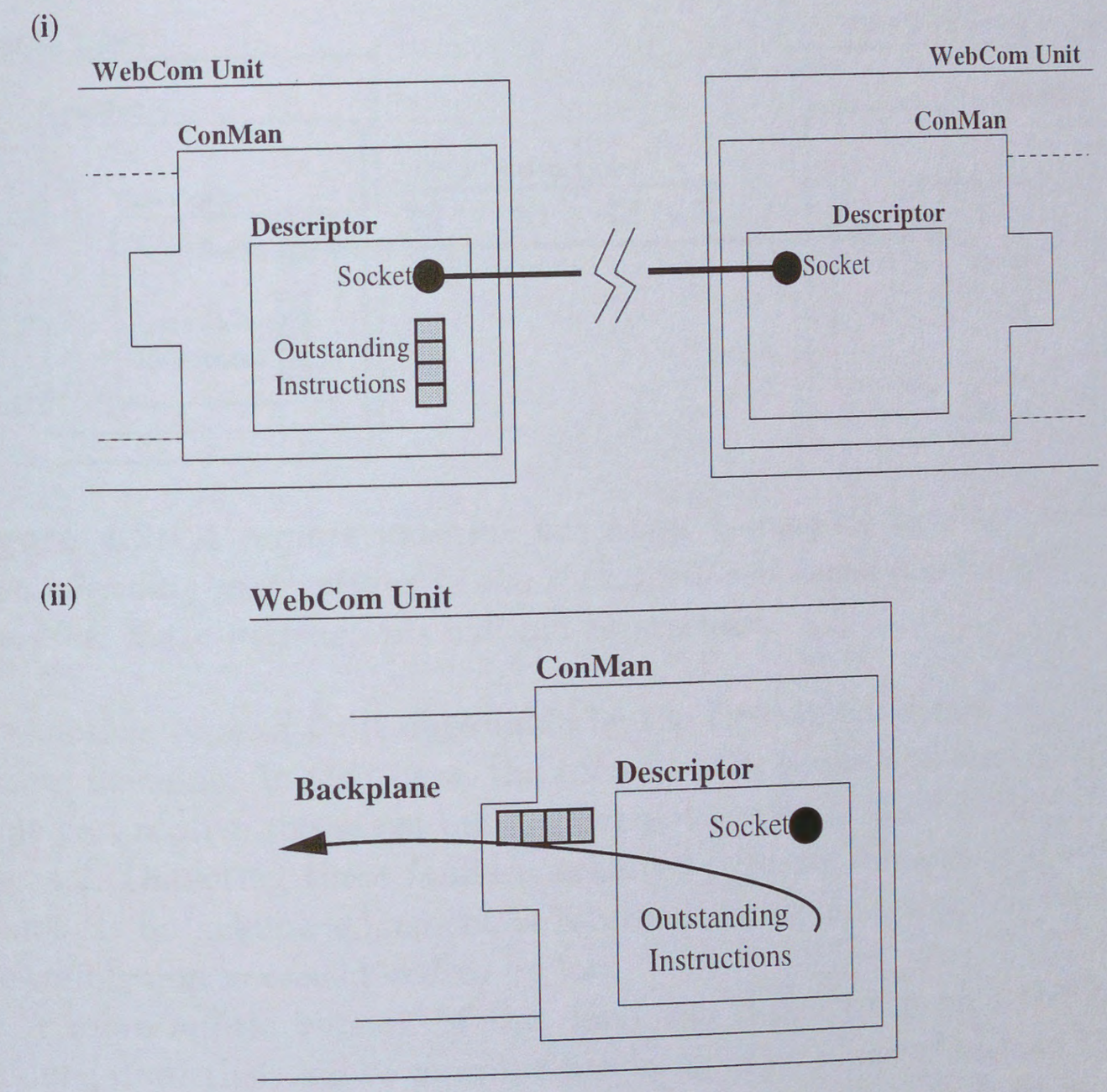


Figure 4.1: ***Fault Masking:****(i) A disconnect from a remote machine is detected by that connection's Descriptor on the local Unit. (ii) The Descriptor passes all outstanding instructions back to the Backplane for rescheduling. Any instructions the disconnected machine was executing must be started anew.*

Most faults in WebCom can be attributed to the breaking of a network link. This may be caused by some intermediary part of the network failing, such as a broken wire or failed router, or by the failure of the actual machine

on the other end of the connection. Fault survivability is obtained through localised fault management within the ConMan Module. Each Descriptor within that Module is responsible for the management of a single connection to some remote Unit and, by extension, is responsible for detecting problems with that connection. Faults such as a broken socket are easily detected as the Descriptor's "read" thread, described in Section 2.12 will catch a Java *IOException*. In this case, the descriptor attempts to reopen the socket. If the connection cannot be restored it is assumed to be lost and any outstanding instructions are passed back to the Load Balancing Module for rescheduling and the Descriptor is deallocated. This is illustrated in Fig. 4.1 and provides *Fault Masking* in WebCom, a process used by many other metacomputing systems.

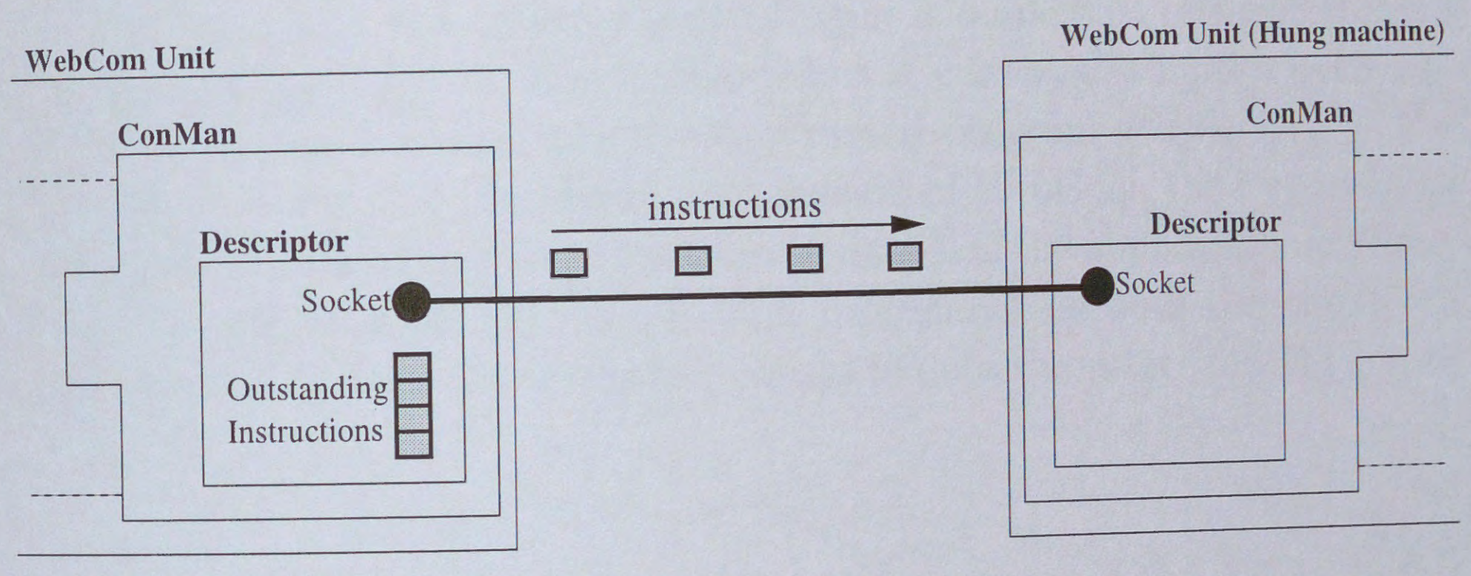


Figure 4.2: *A remote machine has hung, leaving its network connection open. Sending instructions to the socket will not cause a detectable failure. However, these instructions will not be executed.*

Another type of fault detectable by the Descriptor, is that of a "hung" remote machine. In that case, the connection is sound and the remote machine can receive messages but it does not respond. This is illustrated in Fig. 4.2. Detecting these faults is achieved by using message acknowledgements. If no acknowledgement is returned within an estimated time then the connection is considered to be lost.

A more subtle version of this fault can occur when only the Engine Module(s) on the remove machine has hung. The remote ConMan Module may then receive and acknowledge messages as normal, but instructions will not be processed by the Engine Module. The ConMan could probe for this fault by using instruction timeouts, i.e., by expecting a result to be generated for each instruction within a time window. This approach is not used in WebCom since instructions may be of an arbitrarily large (and dynamically determined) size and this time window is difficult to calculate. A preferable option is to directly detect an unresponsive Engine Module.

The definition of a failed Engine Module will change depending on implementations, and on the technology employed. The following heuristic is

employed by the Condensed Graphs Engine Module. The Engine Module is assumed to have failed when the following conditions are true:

1. It is not waiting on any partial results.
2. It has not generated either the result of its computation or further sub-task instructions within a reasonable time.

The time window referred to in Condition 2 is extremely small since once Condition 1 becomes true, the incorporation of a result value into the program graph simply involves setting reference pointers. This should almost immediately terminate the computation or render another node fireable making Condition 2 false.

This heuristic will detect a failed Engine Module, i.e., one whose execution thread has died for some reason, but it cannot distinguish between a failed Engine and an incorrectly constructed program graph, such as that illustrated in Fig. 4.3. In the current version of WebCom, the responsibility of ensuring correct graph construction is placed on the application developer. In the future, well formed graphs will be ensured by both the graph construction environment and runtime checks to detect non-terminating graphs.

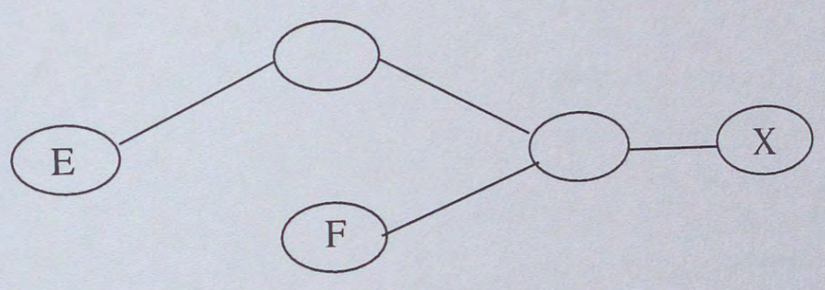


Figure 4.3: *An incorrectly constructed graph, Node F has no operand value and so will never fire, halting the critical path of the graph.*

The polling required to detect a failed Engine Module using these means is performed by the Fault Tolerance Module. When a failure is detected, this Module attempts to restart the Engine Module's execution thread. If the Engine does not restart, or starts but does not generate a result or instruction, then the Unit closes its network connections and shuts down. This allows any connected machines to use the fault masking mechanism described above to continue with the computation. During the shutdown procedure, WebCom logs various information about the circumstances at the point of failure, including a description of graph memory. One of these descriptions, a snapshot of a graph in memory, can be seen in Fig. 4.4.

The fault tolerance procedures discussed so far all implement a form of fault masking, ensuring that if a Unit fails any tasks on which it is working are rescheduled elsewhere. This means that the computation will complete as long as one Unit remains operational. However, there are two drawbacks to this approach. A single point of failure exists in the "root" Unit of the topology and the cost of rescheduling work may be large. In general, the

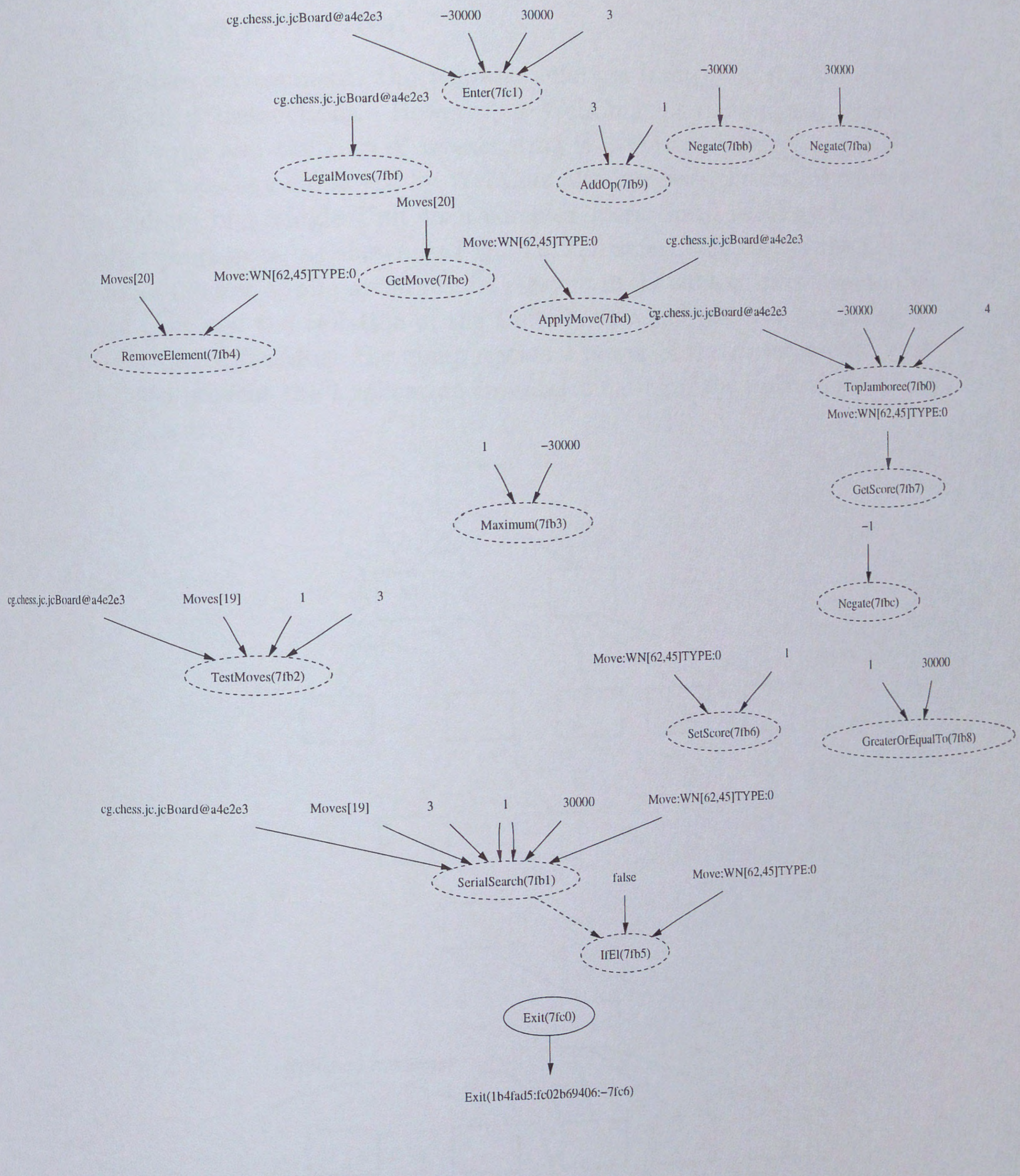


Figure 4.4: *A memory snapshot of a Jamboree graph. All nodes but the exit node have fired (indicated by a dashed node body).*

closer a failed unit is to the root of the topology tree, the larger the cost is likely to be.

4.1.1 Cost of Survival

In a 2-tier environment, the recovery effort is limited by the size of the rescheduled instructions. However, a WebCom instruction may be arbitrarily large and the cost of re-executing it is correspondingly large. The dynamic topology employed by WebCom also increases the cost of recovery. The failure of a single Unit in a complex hierarchical topology may also orphan processors, as shown in Fig. 4.5. Therefore, the cost of the failure includes the loss of all partial results resident in the subtree connected to the failed Unit and the isolation of the Units in that subtree. The orphaning of these Units will reduce the computational power of the metacomputer and may also prevent the application from terminating if the isolated resources were executed.

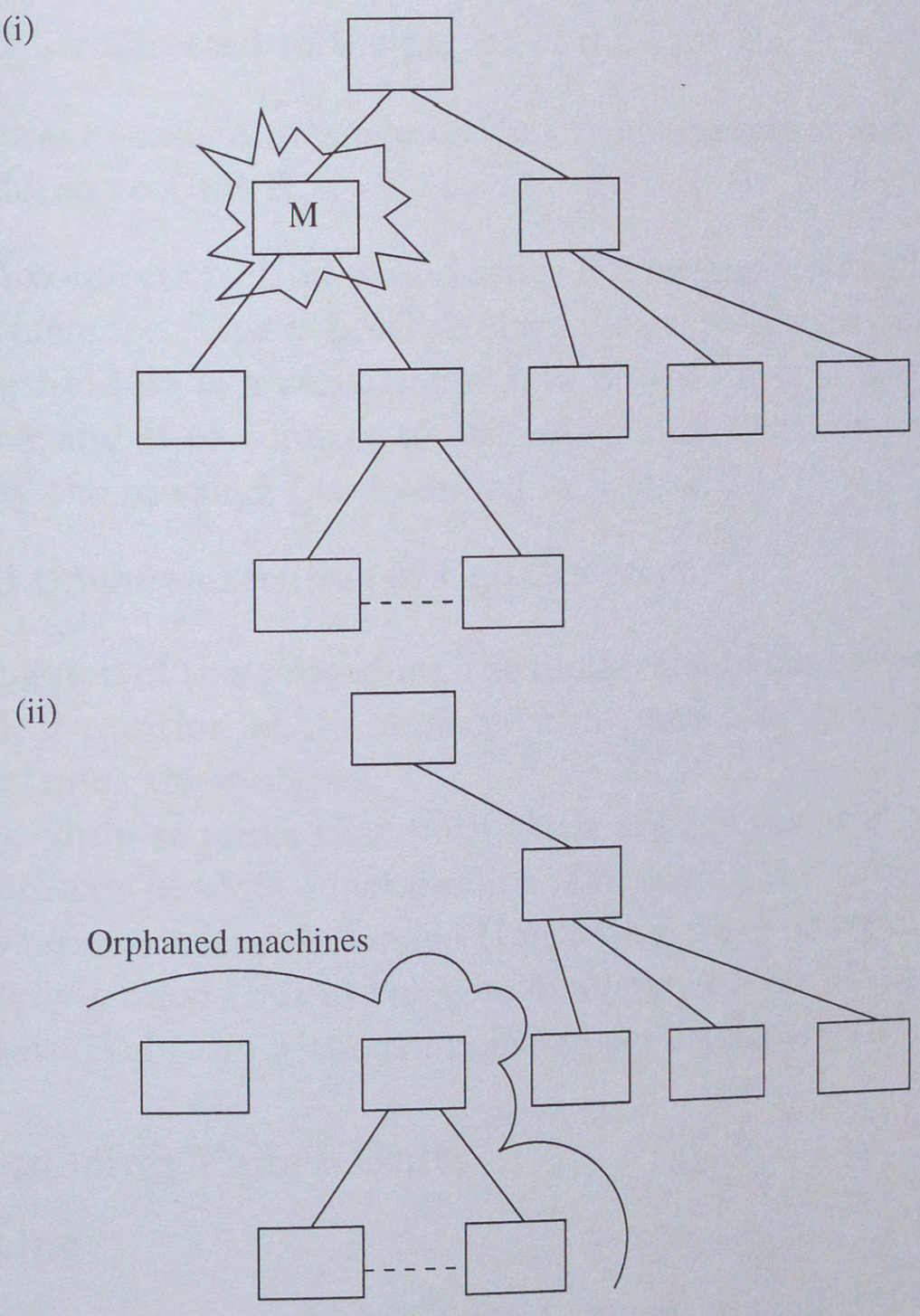


Figure 4.5: *The failure of a machine resulting in both the loss of work done by its descendants and the orphaning of those descendants.*

4.2 Fault Recovery Procedure

The previous section illustrates how a single fault can orphan subtrees of the execution and cause the loss of work. In the worst case, the "root" node fails. These problems can be addressed by the Fault Tolerance Module. This Module interfaces with the Backplane and sends messages to Fault Tolerance Modules on connected machines to maintain certain information about the Unit's position in the topology. It uses a modified version of the following *Unit Replacement Procedure* to manage fault impact.

4.2.1 Unit Replacement

The ability to redirect Units within the WebCom topology can be used to provide a general procedure for replacing Units regardless of their location in the topology. Any Unit A may be substituted by any Unit B through the following steps:

1. Unit B is redirected to the parent of the Unit A.
2. The parent sends a message to Unit A instructing it to redirect all of its children to Unit B.
3. Unit A connects to Unit B and sends it a message containing its entire graph memory. This is possible since the graph memory in the default Engine Module is a serializable Java object. If it is not possible for Units A and B to connect to each other directly, their parent is used to relay the message (as described in Section 2.18).
4. Unit B resumes execution of Unit A's graph.

On completion of this procedure, the replacement Unit has assumed both the topological position of the replaced Unit, and also its execution state. Fig. 4.6 illustrates these steps.

This procedure requires that both Units are equivalent in terms of Engine and ConMan Module functionality. Employing this procedure allows WebCom to replace a heavily loaded Unit with a more powerful Unit and it allows a heavily loaded Unit at the root of a large subtree of the topology to gracefully leave WebCom without major impact on the overall computation.

4.2.2 Replacing Failed Units

Housekeeping

The recovery procedure enables WebCom to replace a failed Unit anywhere in the topology while maintaining the structure of the topology and reincorporating partial results. To accomplish this, the following information is maintained by the Fault Tolerance Module on each Unit:

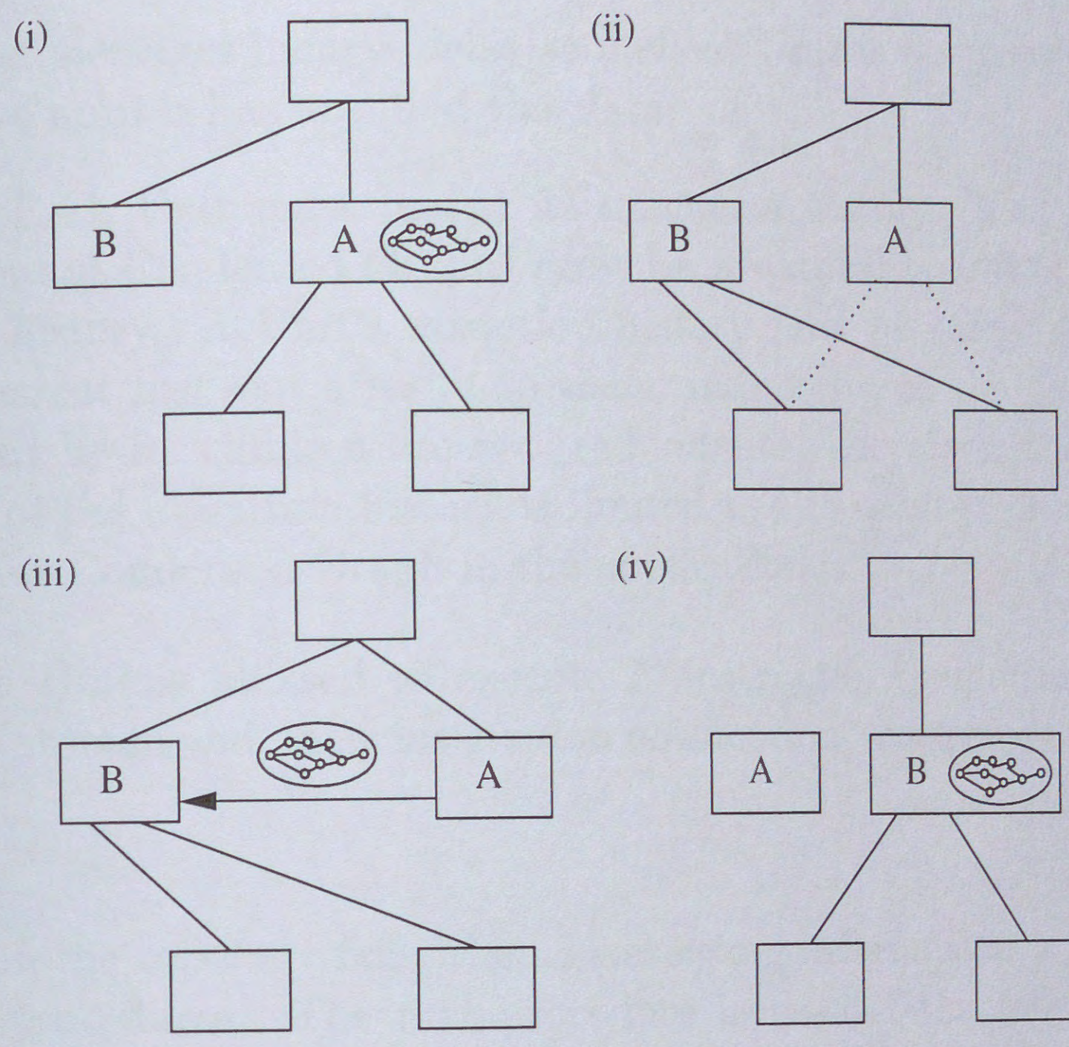


Figure 4.6: *Four stages of the Processor Replacement Routine, Unit B is replacing Unit A.*

1. A record of its grandparent in the topology. It receives this information when connecting to its parent.

2. Every result sent to its parent, along with that result's associated instruction. This information is stored in the Fault Tolerance Module and is referred to as the *execution history*.

3. The addresses of all its siblings (if no grandparent exists).

4. If its parent is the root node of the topology, a copy of that parent's current instruction.

Housekeeping Costs

There are a number of costs associated with the transmission and storage of this extra information at runtime. These costs are analysed later in this chapter.

Communications: A message must be sent whenever a connection is established to send grandparent information to new children. N+2 (Where N = the number of existing children) messages must be sent when a new child connects to the root node, one for each existing child and one to the new Unit, listing those children. An additional message informs the new Unit of what instruction the root node is executing.

These messages incur a delay as a client Unit is not considered to be active until it has acquired this data.

Storage: Each Unit must record its execution history. The hierarchical nature of Condensed Graphs provides a natural bound on the size of this history. A Unit's execution history may be safely deleted once its parent has sent a result to the grandparent, as the partial results stored by its children are now redundant. Therefore, the maximum size of the execution history is limited to the number of nodes in the largest Condensed Graph in the application.

If each Unit is allowed to execute N instructions simultaneously, the maximum storage and "new instruction notification" costs are a factor of N.

Action

If a Unit in the topology fails, the housekeeping information is used by the following procedures. The first procedure is used if the failed Unit had a parent and the second is used if the failed Unit was at the root of the topology tree.

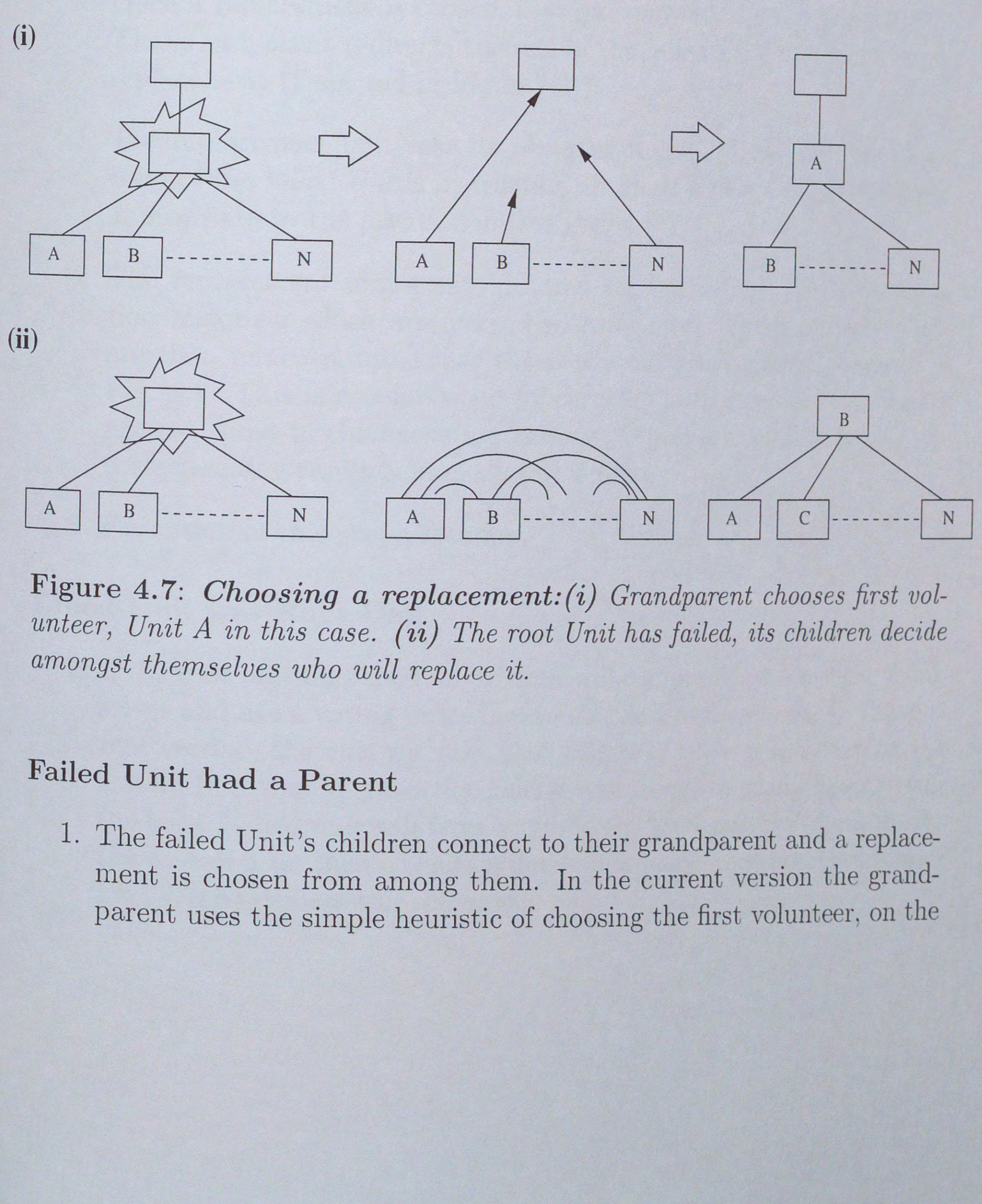


Figure 4.7: ***Choosing a replacement:*** *(i) Grandparent chooses first volunteer, Unit A in this case.* ***(ii)*** *The root Unit has failed, its children decide amongst themselves who will replace it.*

Failed Unit had a Parent

1. The failed Unit's children connect to their grandparent and a replacement is chosen from among them. In the current version the grandparent uses the simple heuristic of choosing the first volunteer, on the

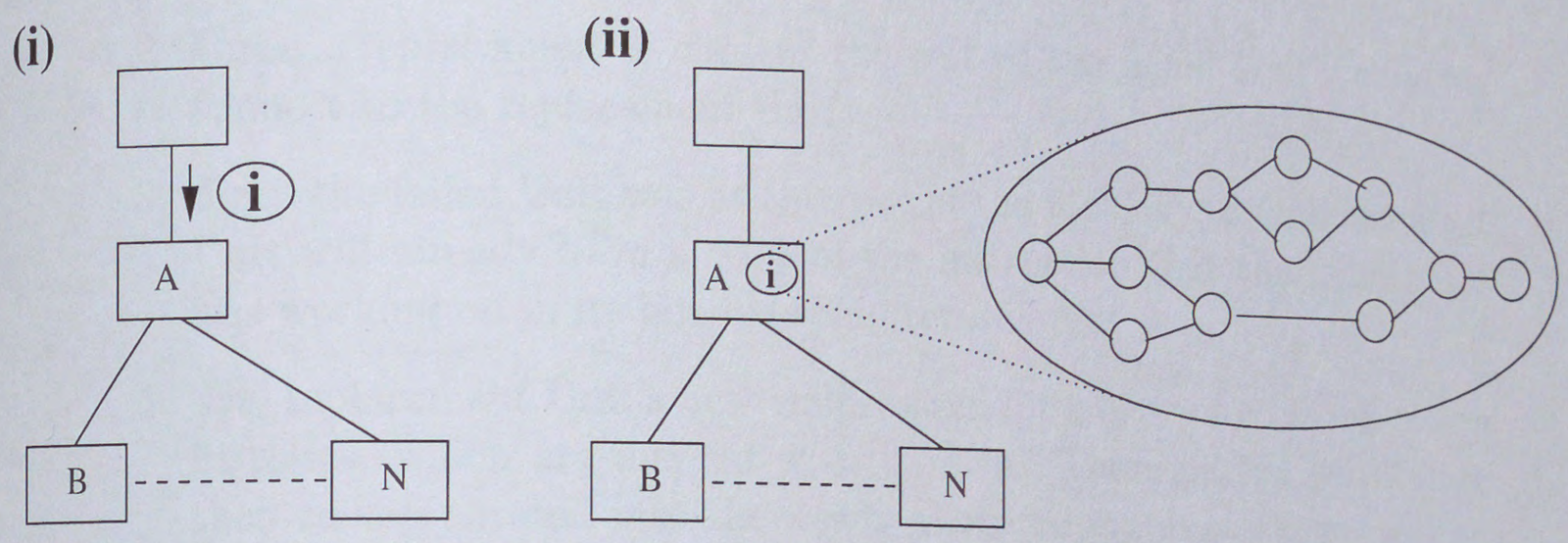


Figure 4.8: *The replacement Unit requests a new copy of the failed unit's instruction. This instruction is evaporated to a new graph.*

assumption that that Unit will be either the least loaded of the volunteers or it will have the fastest network link to the grandparent. An advantage of this heuristic is that the grandparent does not have to wait for *all* of the failed Unit's children to volunteer before choosing its replacement. Both of these scenarios are depicted in Fig. 4.7.

2. Once a replacement is chosen, it stays connected to the grandparent. The grandparent redirects the rest of the failed Unit's children to the replacement (Depicted in Fig. 4.7(i)).

3. The replacement unit asks its new parent (its old grandparent) for a copy of the failed Unit's instruction. It loads a new CondensedGraph appropriate to the instruction (see Fig. 4.8).

4. The replacement unit's new children send it a copy of their execution histories, which are merged with its own. These partial results are then reincorporated into the newly allocated graph as shown in Fig. 4.9. This is accomplished by checking each outgoing instruction against those in the execution history. Where a match is found, the corresponding result is immediately reused.

5. Execution of this graph resumes.

Failed Unit was Root Unit

1. If the root unit fails then its children will communicate amongst themselves and use a voting procedure to choose a replacement. In the current version, the unit with the least clients of its own is chosen on the assumption that if a heavily loaded unit acquires many clients then the least loaded unit will have acquired the least number of clients. In the case of a tie then a random decision is made based on IP address. Fig. 4.7(**ii**) depicts this scenario.

2. Once a replacement is chosen, the rest of the failed Unit's children connect to the replacement Unit.

3. Since the failed Unit was at the root of the topology, the replacement Unit will already have a copy of the instruction that the failed Unit was working on in its housekeeping records.

4. The replacement Unit's new children send it a copy of their execution histories, which are merged with its own. These partial results are then reincorporated into the newly allocated graph as shown in Fig. 4.9.

5. Execution of this graph resumes.

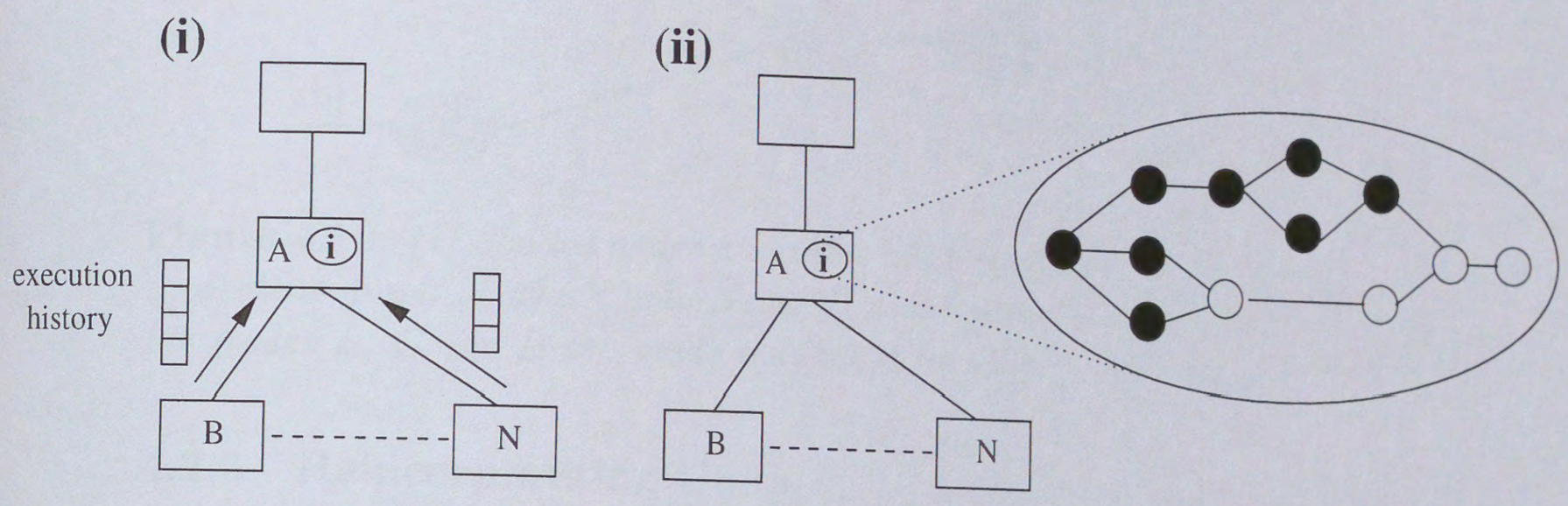


Figure 4.9: *The replacement Unit's new clients send it their execution histories. These results are then reused in the new graph, bringing it to a state similar to that of the graph on the failed Unit.*

Rolling a Graph Forward If the failed Unit had not executed any instructions locally then the new graph will resume from whatever point that Unit stopped at. However, if the failed Unit had executed instructions locally then these instructions may have to be re-executed. The following routine is used to prune unnecessary nodes from the graph.

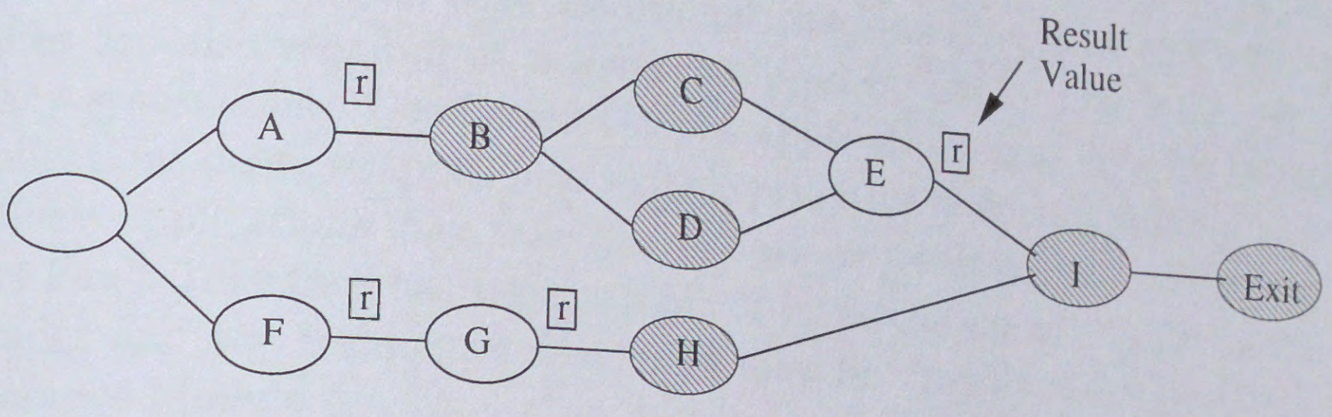
1. A recursive routine parses the graph, beginning at the Exit node and working backwards.

2. If no result value is present for the current node, it is added to a new graph and the routine is called again on all nodes for which it is a destination.

3. If a result is present for that node then the result value is placed in the graph as a constant value.

Fig. 4.10 depicts a graph with known results for some nodes. The algorithm produces a smaller equivalent graph containing only the nodes from the original graph that must be re-executed.

(i)



(ii)

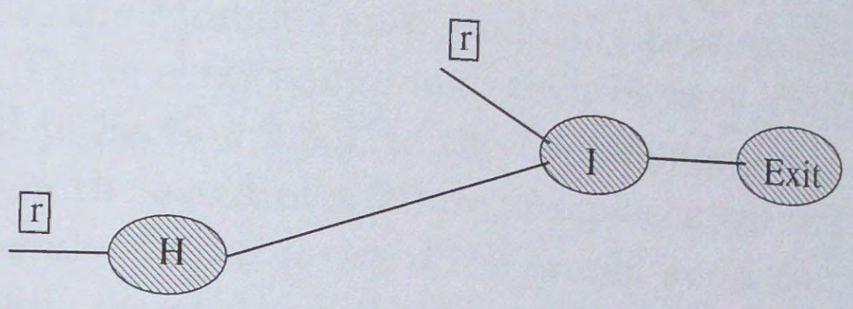


Figure 4.10: *(i) Shaded nodes are nodes whose result is not known. (ii) An equivalent graph in which only Nodes H and I must be re-executed. Results for Nodes B, C and D are made redundant by other known results.*

4.2.3 Reincorporating Units

A failed WebCom Unit can be reincorporated into the network topology by attaching to any other Unit. To facilitate this, each ConMan Module records a list of N machines to which it has connected. When rejoining WebCom it attempts to connect to the Unit at the start of the list and works through the list until a contact is made. The Load Balancing Modules on other Units will redirect it to where it is most needed.

This feature allows Units which had failed and then recovered to rejoin the computation. This, coupled with the fault tolerance procedure described above, reduces the impact of temporary apparent faults (such as a user rebooting a machine or the temporary loss of a network link).

4.2.4 Incorporating Traditional Checkpointing and Rollback

The adaptive lineage routine presented above cannot be directly likened to traditional checkpointing and rollback techniques but can utilise them appropriately. It uses the natural distribution of partial results from graphs as a continually evolving checkpoint of the states of computational subtasks. Should a fault occur, the routine does not roll back but rather begins with a copy of the original graph, using partial results to quickly *roll forward* to the point where the computation failed. Condensed Graphs can be seen as a set of sequencing constraints on atomic tasks and, therefore, the adaptive lineage routine is fault tolerance at a higher, *job control*, level.

It naturally follows that traditional checkpointing and rollback techniques are also required at a lower, or *process*, level. The integration of legacy systems into Condensed Graphs applications may involve complete applications being wrapped into a single primitive node in a graph and some of these applications may take hours, days or weeks to run. This process level Fault Tolerance has been a topic of investigation in the literature for decades and the techniques developed may be directly used by the Fault Tolerance Module.

For example, the Fault Tolerance Module on a WebCom Unit can employ utilities such as Condor's checkpointing mechanism for Grid computing. With this, the Module can periodically capture the state of atomic computations. These checkpoints may then be sent via the messaging system, as discussed in Section 2.3.2, to the Unit's parent or to one or more clients for storage. In the event of a failure, a replacement machine may then use both partial results to *roll forward* the graph, and checkpoint information to *roll back* atomic instructions that the failed machine was working on.

4.3 Adaptive Lineage Management

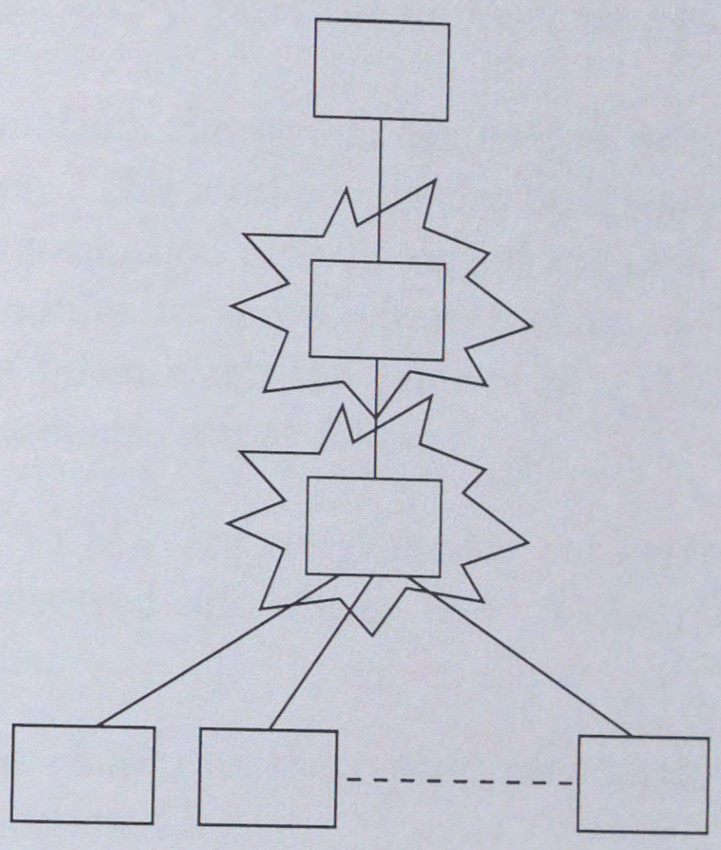


Figure 4.11: *A chain of faults occur simultaneously in the lineage.*

It is unrealistic to assume that faults will occur in single, isolated instances on a large network of heterogeneous machines. If faults occur on two units next to each other in their lineage, as shown in Fig. 4.11 for example, then orphaning of topology branches and loss of partial results will occur regardless of the basic Fault Tolerance scheme presented above. This scenario of chained and simultaneous faults is referred to as a *group fault*. A group fault occurs when a chain of units fail simultaneously or if a Unit

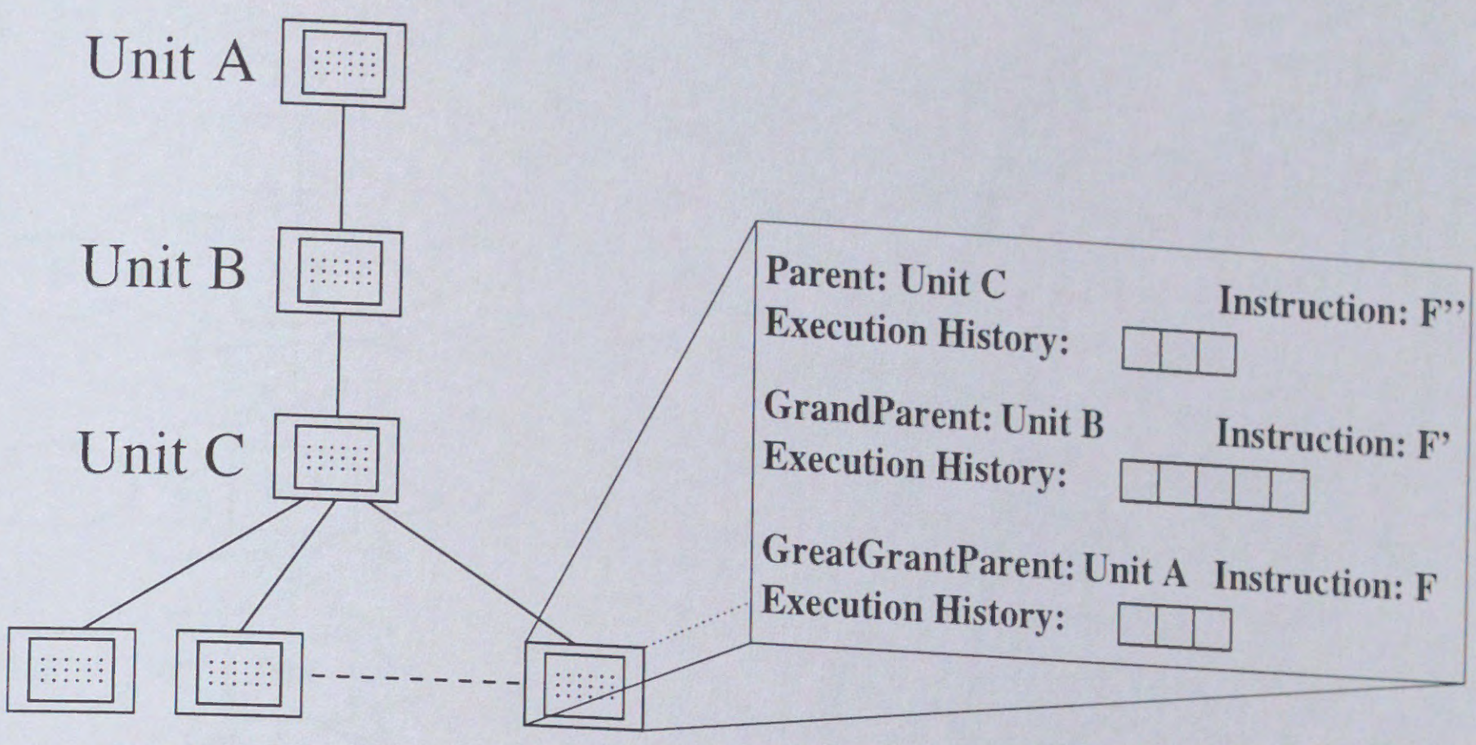


Figure 4.12: *Lineage information recorded by a client. This is the information required to recover from a chain of two faults.*

fails while its parent is failed.

To deal with a group fault, the Fault Tolerance Module's functionality is extended to record an extended lineage, i.e., parent, grandparent and great-grandparent. In addition, each unit records the partial results generated by its "ancestors". Fig. 4.12 depicts the information recorded by units in the sample lineage.

With this information, the system can recover from a chain of two units failing simultaneously. This method can also be extended to record ancestor and partial result information for a lineage of length N which will allow the Fault Tolerance Modules to recover from a chain of N faults. Fig. 4.13 illustrates the steps taken when the value of N is three and the algorithm steps taken in this scenario are as follows:

1. Machines M_1 to M_N are collaborating on a computation where the topology has evolved into a deep tree. A chain of three machines in the lineage fails.

2. Machine M_5 is chosen as the replacement using the same means as described in Section 4.2.2.

3. Machines M_6 to M_N connect to M_5 as clients and send it their execution histories. M_5 also has the partial results generated by Machines M_2, M_3 and M_4.

4. M_5 loads a new graph F, which M_2 was working on. Any already existing results that were generated by M_3 are plugged into it.

5. M_5 will eventually uncover subgraphs F' and F". As they do so, pre-existing results are reincorporated into them accordingly. It is not required that M_5 execute all condensed nodes locally, only those nodes

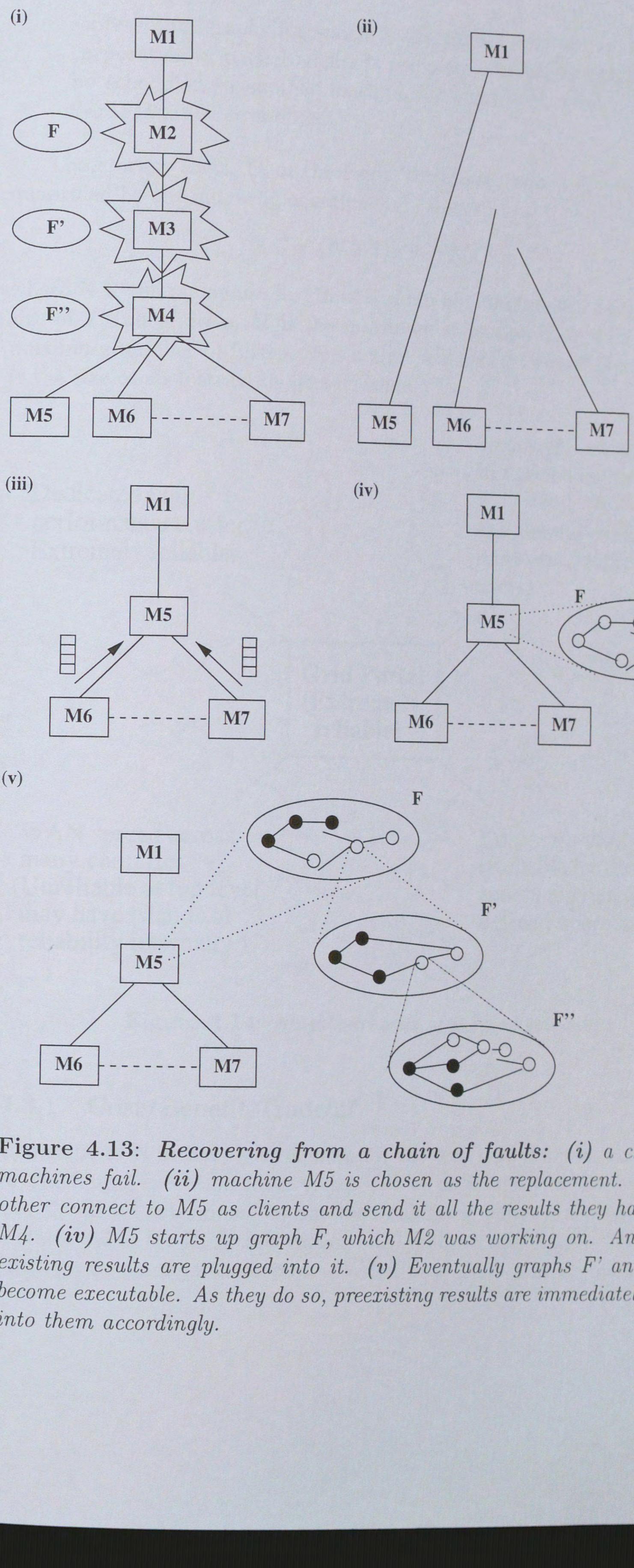


Figure 4.13: ***Recovering from a chain of faults:*** *(i) a chain of 3 machines fail. (ii) machine M5 is chosen as the replacement. (iii) the other connect to M5 as clients and send it all the results they had sent to M4. (iv) M5 starts up graph F, which M2 was working on. Any already existing results are plugged into it. (v) Eventually graphs F' and F" will become executable. As they do so, preexisting results are immediately plugged into them accordingly.*

representing graphs for which is has partial results. Once the reincorporation of partial results is complete, the whole graph may then be scheduled to another machine for execution, since a graph is a `Serializable` Java object.

The storage costs, C, of the Fault Tolerance routine for each unit are quantified by the following equation:

$$C = (N+1)a + NMIr$$

where N is the maximum length of a chain of simultaneous faults, a is the size of a Unit address, M is the maximum execution history size, I is the maximum number of Instructions a Unit will execute simultaneously and r is the size of an Instruction/Result tuple.

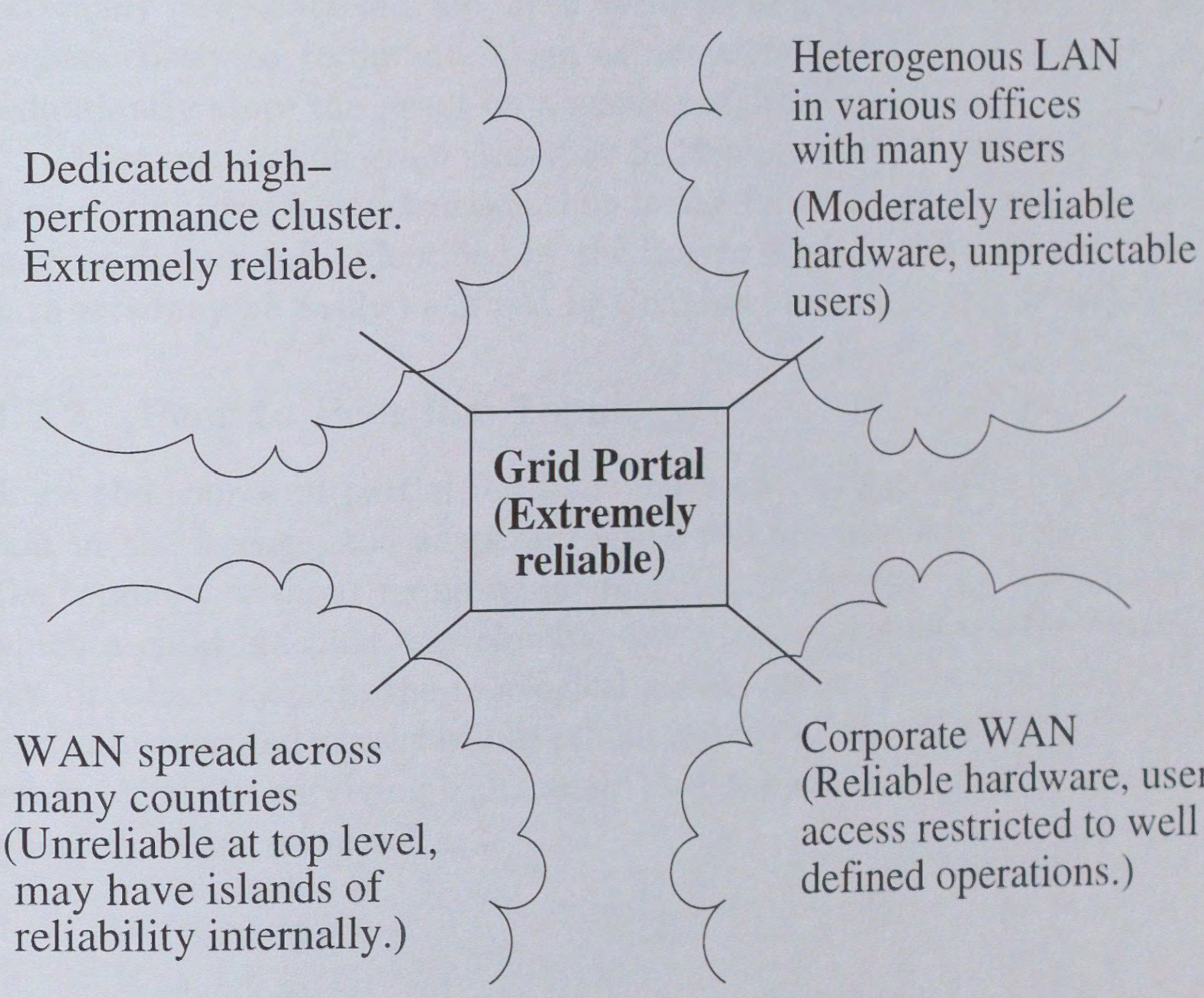


Figure 4.14: *Neighbourhoods of varying reliability.*

4.3.1 Cost/Benefit Tradeoff

The value of N in the above equation is the only real variable. Factors such as maximum execution history size will be determined by the application and I will be determined by the load balancing policy. Instead of setting a static value for N, future versions of WebCom may employ an adaptive strategy to manage these costs. When new Units join the computation,

they may be assumed to be unreliable and a high value of N may be used in their neighbourhood. Over time, these machines and associated network links may prove their reliability and so the value of N can be lowered as the perceived risk of group faults occurring, and the size of those groups, diminishes. In this manner, the system may *learn* how much housekeeping to employ in various areas of the topology, dividing it into *neighbourhoods* of varying reliability. Fig. 4.14 illustrates a scenario of different neighbourhoods working together.

The amount of housekeeping done by WebCom may also be increased, regardless of reliability, where instructions are of a large grainsize and the cost of recording partial results is small relative to the cost of recomputing those results. In this case it is worth saving the partial results since even if a fault is unlikely, its potential cost would be very large. Conversely, where extremely large data sets are taken as inputs or generated as results, it may be less costly to recompute them as necessary rather than transmit and redundantly store the result on a number of other machines.

These instruction grain sizes may be identified by the application developer during graph construction, but in the future large grain instructions may be dynamically identified by the Engine Module at runtime and large data sets may be easily identified by ConMan modules during transmission.

4.3.2 Peer to Peer like Topology

Since the source of partial results is irrelevant with respect to their location in the lineage, the adaptive routine will function in a "peer to peer" like topology without requiring modification. This topology type is one in which a child machine may simultaneously act as a parent to its own parent, or where loops in the topological lineage occur, as shown in Fig. 4.15. Partially executed graphs will be rebuilt as before, reincorporating whatever information the surviving Units retain until the graph is rebuilt to the point where the fault occurred.

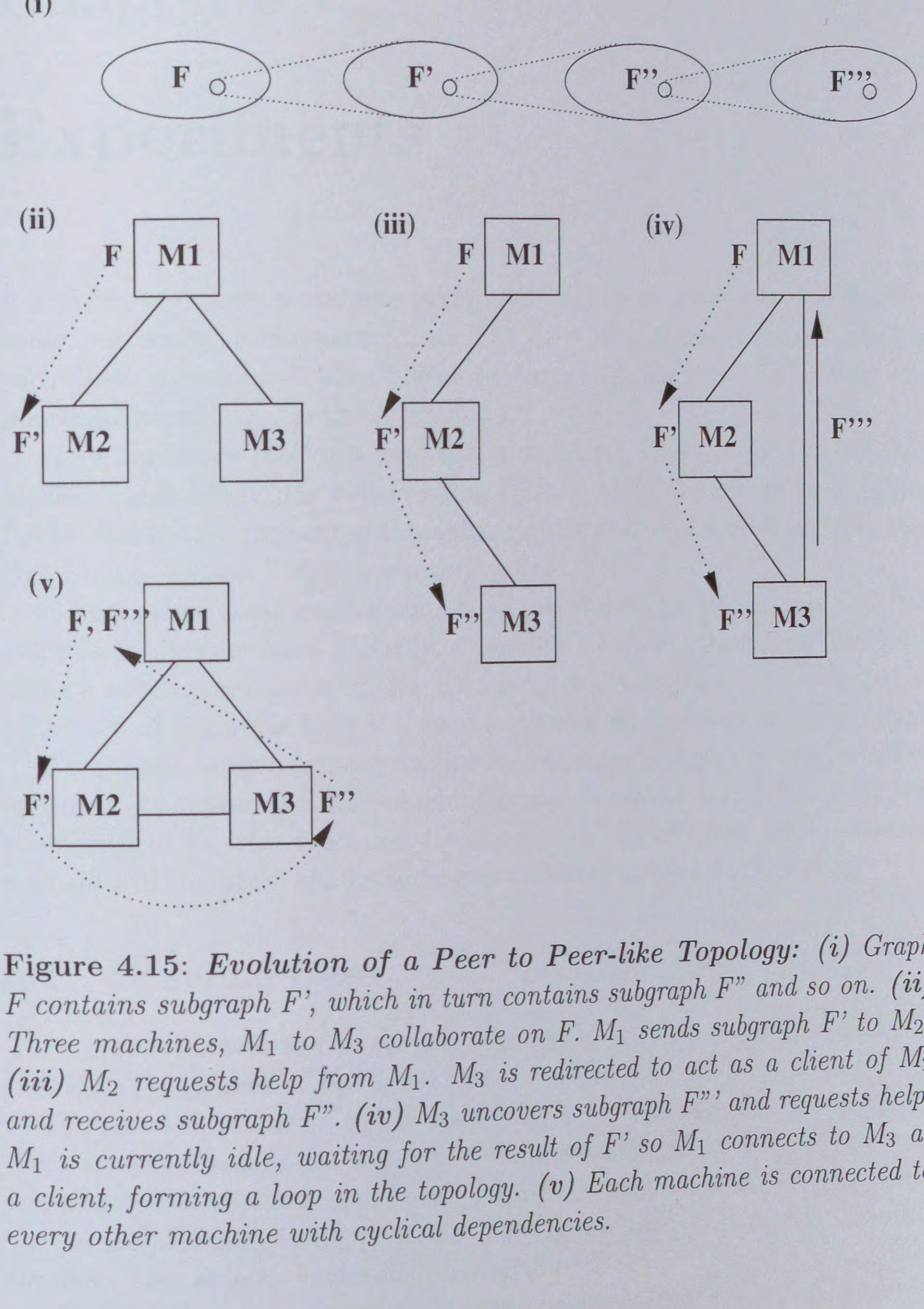


Figure 4.15: ***Evolution of a Peer to Peer-like Topology:*** *(i) Graph F contains subgraph F', which in turn contains subgraph F" and so on. (ii) Three machines, M_1 to M_3 collaborate on F. M_1 sends subgraph F' to M_2. (iii) M_2 requests help from M_1. M_3 is redirected to act as a client of M_2 and receives subgraph F". (iv) M_3 uncovers subgraph F"' and requests help. M_1 is currently idle, waiting for the result of F' so M_1 connects to M_3 as a client, forming a loop in the topology. (v) Each machine is connected to every other machine with cyclical dependencies.*

Chapter 5

Experiments

A number of experiments were performed to exercise WebCom and measure some performance characteristics. The $\mathcal{CG}$ Chess application was used in each test, searching 4 plies ahead. Chapter 3, Section 3.6 discusses the potential parallelism in the application.

The hardware platform used was a dedicated cluster comprised of fifty *blades*. Each blade has two Pentium III 1.26 GHz processors and 1GB of RAM. Blades are connected through a gigabit switch and each runs the Red Hat Linux, version 7.3[38] operating system.

The goal of these experiments is to investigate the performance of WebCom's fault tolerance. Initially, a baseline execution profile for WebCom using a varying number of nodes in a fault free execution, is established. The efficiency of fault masking is then investigated when faults are introduced. The complete fault tolerance routine is then employed, firstly in a fault free execution to investigate housekeeping costs. Faults are then introduced and their impact on the execution time assessed. Ideally, the fault tolerance routine will maintain the baseline execution time, even if faults occur.

5.1 Baseline Case

The application was run on varying numbers of units to establish WebCom's best case performance. Each unit was allowed to send messages to its parent to request helper units. This allows the evolution of an n-tier topology tree, as described in Section 2.4.2. In these experiments, WebCom only transmitted condensed nodes – all primitives were executed locally. Since the condensed nodes are generally large grain instructions, the dynamic topology was able to better utilise available units.

Fig. 5.1 shows the execution speedup using this set of features. Average execution times were used in the construction of this graph.

Fig. 5.2 shows the number of messages sent during this execution. Most messages during the execution were requests for helper clients. As this graph

shows, the number of requests for clients is not directly affected by the number of machines participating. A number of factors affect the messages sent. When a unit asks its parent for clients, the parent may pass on that request to its own parent. If fewer units are involved then the topology tree will generally be less deep and each request for clients will be passed on, and therefore duplicated, fewer times. However, they will tend to ask for clients more frequently since the supply of clients is more limited.

Another factor in the number of messages sent is the number of instructions and results generated and transmitted. For the *CG* Chess application, the number of instructions generated will remain constant as, in its current form, it does not employ any speculation. However, an increased number of participating units will lead to units transmitting more instructions than are executed locally.

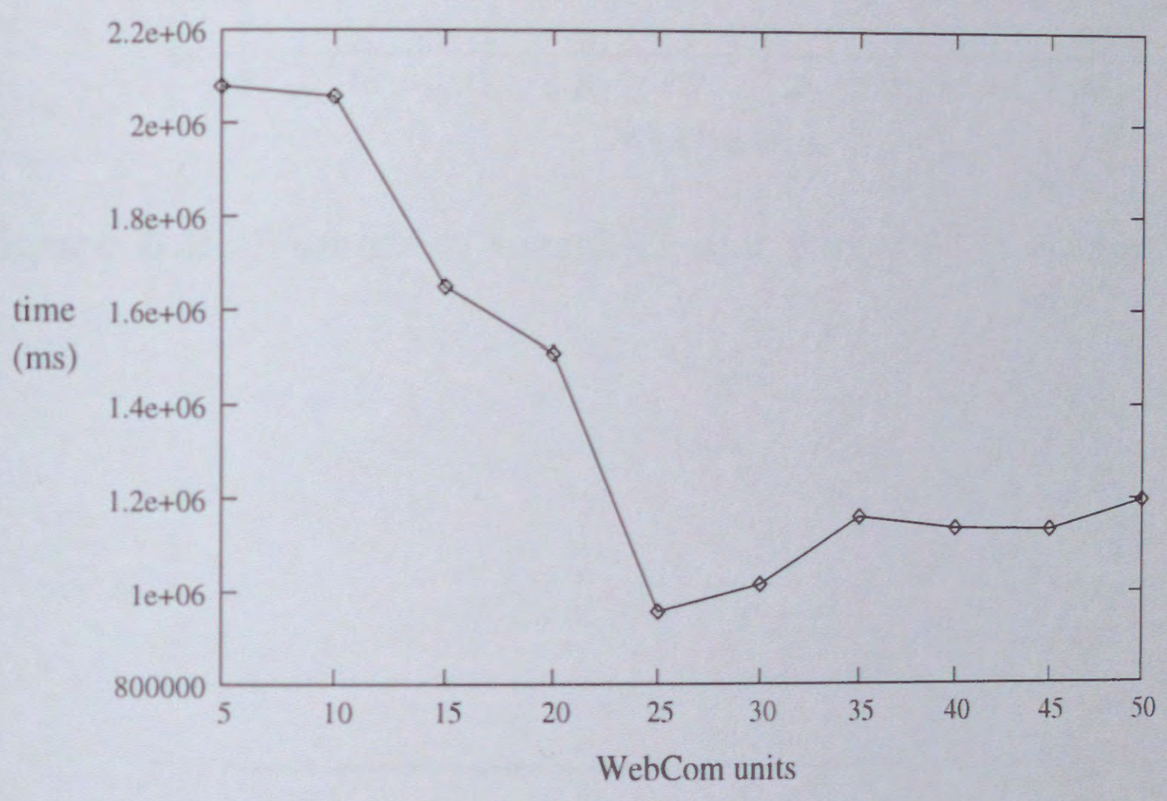


Figure 5.1: *N-Tier topology speedup as units are added to WebCom.*

These results demonstrate how WebCom takes advantage of implicit parallelism in condensed graphs. In a fault free environment, speedup increases as the number of participating units increases. The executions are particularly fast with 25 and 30 participating units. There is a small decrease in the number of messages sent(Fig. 5.2) and number of redirections performed (Fig. 5.3) during these executions. From this it can be concluded that the configuration was more effective with this many units. As the number of units increases from thirty, the redirection of units becomes less effective.

The ratio of machines to speedup in general is not optimal for a high performance system but it is adequate for the purposes of these experiments. WebCom is limited by its naïve Load Balancing Module and better speedup could be achieved through more advanced load balancing techniques, some of which are being investigated for WebCom in[55].

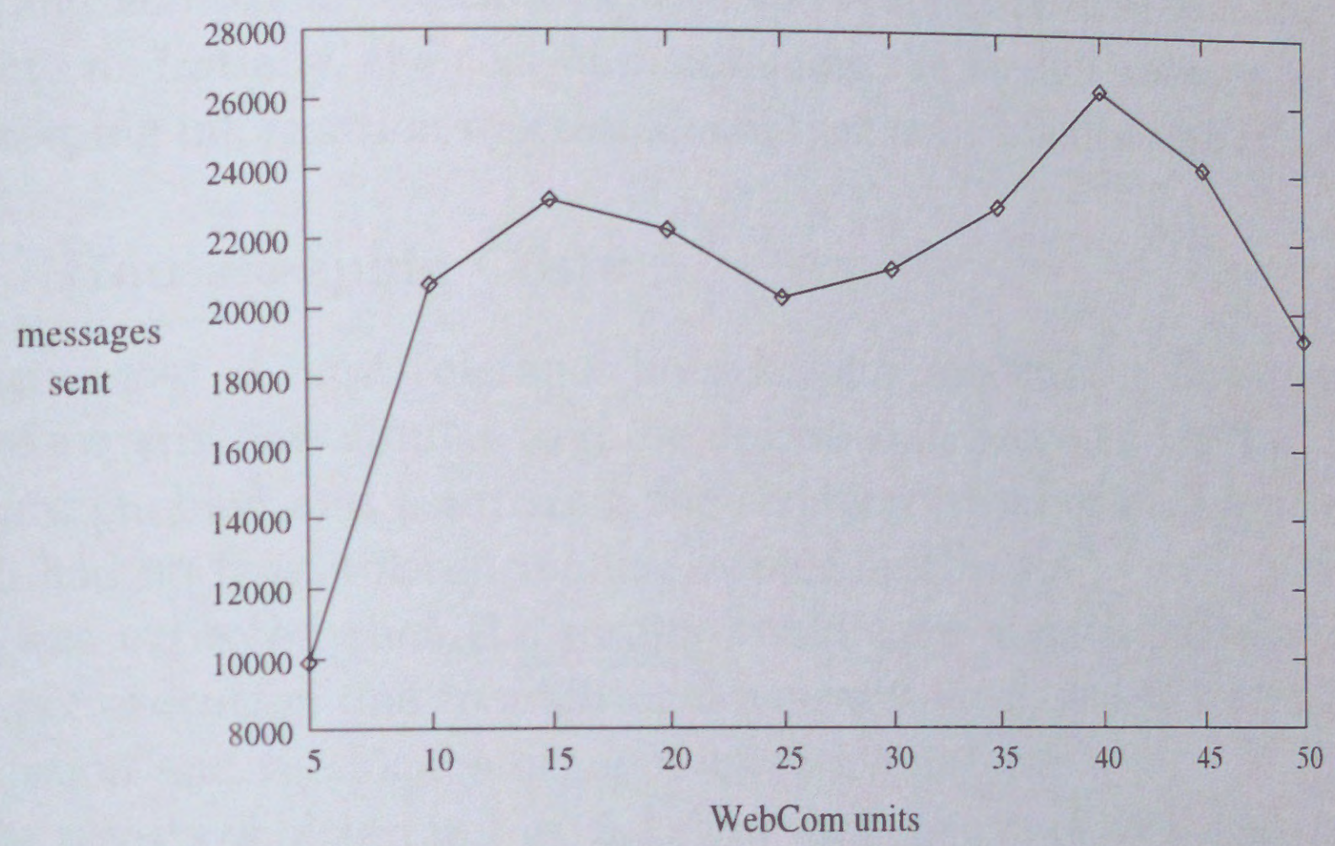


Figure 5.2: *Number of messages sent during N-Tier execution*

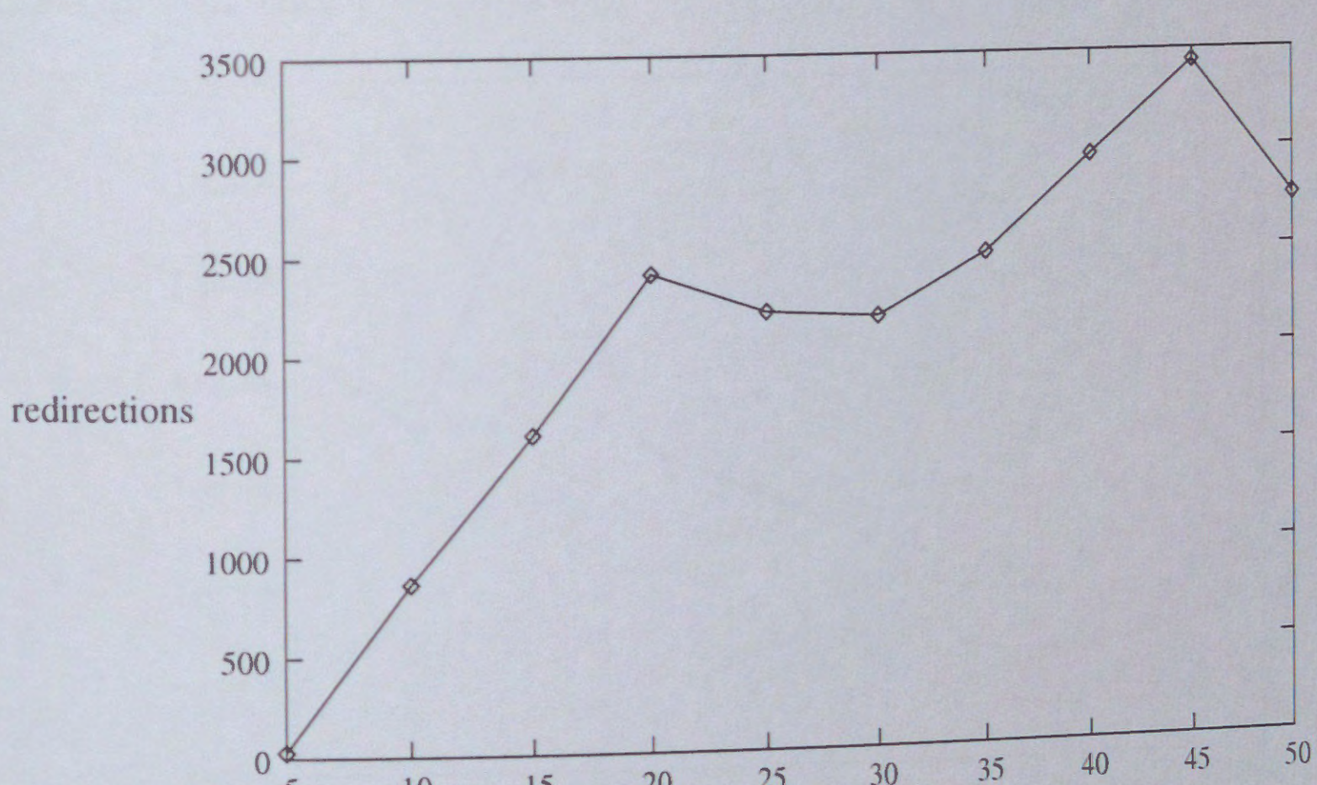


Figure 5.3: *Number of redirections for each N-Tier execution.*

5.2 Faults Tolerance Results

The effectiveness of the Fault Tolerance Module was tested by introducing faults and comparing executions with the optimal results from the previous section. Initially, the cost of maintaining the Fault Tolerance Module's housekeeping information was tested and then actual faults were introduced.

5.2.1 Housekeeping Costs

The extra cost of fault tolerance housekeeping was measured by running a suite of experiments similar to those described in Section 5.1 but with fault tolerance enabled and comparing the results to those of the baseline cases (which had no fault tolerance housekeeping overhead).

It was expected that the results would show a small increase in time taken per execution due to additional messages being sent to record lineage information and the local storing of execution histories.

The results depicted in Fig. 5.4 show that the cost of the housekeeping is generally small and in some cases actually becomes negative. Investigation showed that this is due to the recording of partial results. The execution history is consulted before each instruction is executed and if a result exists for a node with the same operator and operands it is used immediately. As discussed in Chapter 3, Section 3.7, it is not uncommon for the same board position to be evaluated many times and so the local execution history on each processor acts as a local transposition table. This optimisation should provide speedup for classes of applications which may expose the same instruction more than once.

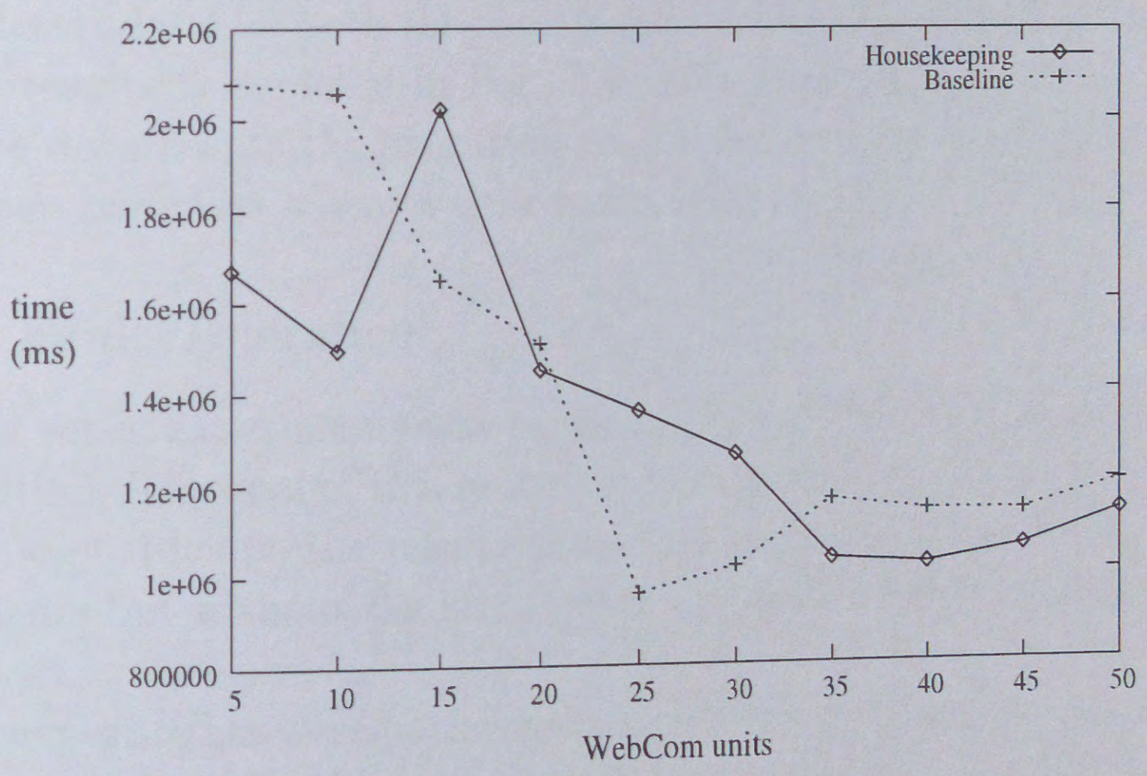


Figure 5.4: *N-Tier topology speedup as units are added to Fault Tolerant WebCom in the absence of faults.*

The number of messages sent also increased as expected due to the house-

keeping reliance on a number of messages being passed before a connection is considered active.

5.2.2 Fault Masking

To test the effects of faults on the execution, WebCom was again exercised with the $\mathcal{CG}$ Chess application as above. However, in these experiments a number of failures were simulated. An independent application, called *randomFaults* was run on the cluster. This application was implemented as a shell script and took as a parameter a failure percentage. This value determines what percentage of machines will "fail" during the execution and was set to 10%. As an execution progresses, the randomFaults script (see Code 5.1) sleeps for a period of time between one hundred and two hundred seconds, randomly chooses a machine to fail and terminates the WebCom application on that machine using the unix `kill` command.

Fault masking was tested by running the application with fault tolerance disabled. This leaves WebCom in fault masking mode, i.e., if a unit detects that one of its clients has failed, it reschedules any work that had been sent to that client. This approach provides fault survivability but risks large amounts of work being lost and units being orphaned.

The timing results for these tests are depicted in Fig. 5.5. Large peaks appear in the graph where one or more of the failures caused the loss of work and machines. Other points in the graph remain reasonably close to the fault free execution times. It can be concluded from these results that while the fault masking approach has a reasonable best case, its worst case scenario can be extremely costly. The variation is this cost extremely unpredictable as it is determined by both the timing and the location of faults. An average of these results is depicted in Fig. 5.6 and compared against the baseline. For some data points the execution time is close to the baseline but others show steep increases where a fault had a large impact.

5.2.3 Fault Tolerance

The final set of experiments was performed using WebCom's full fault tolerance routine. The goal of this routine is to minimise the impact of a fault on the topology. Ideally, the results should be similar to those of the baseline experiments but without the large spikes - all faults should have a relatively low impact.

The experiments were performed under the same conditions as the fault masking experiments using n-tier topology management (the fault tolerance routine will have no effect in a 2-tier topology and degenerates into fault masking). The timing results are shown in Fig. 5.7 and a graph of the average times is shown in Fig. 5.8.

```
#!/bin/bash

#kill command
command='killall java'

#Time to wait between faults
SLEEP_MIN=100
SLEEP_MAX=200
SLEEP_RANGE=$(($SLEEP_MAX - $SLEEP_MIN))

#Number of machines to fail
NUM_FAILING_MACHINES=$1

#Initialise array of machine names
for ((arg=1 ; arg<50 ; arg++))
do
    machines[$arg]=$((100 + $arg * 2))
done

# Main loop - sleep then cause a fault
for ((counter=0 ; counter<NUM_FAILING_MACHINES ; counter++))
do
    myTime=$(($SLEEP_MIN + $(($RANDOM % $SLEEP_RANGE))))
    echo "Sleeping for $myTime"
    sleep $myTime

        index=$(($(($RANDOM % $((${#machines[*]} - 1)))) + 1))
        node="cuc${machines[$index]}"
        unset machines[$index]
        tempArray=( ${machines[*]} )
        unset machines
        machines=( ${tempArray[*]} )
        unset tempArray
        echo "Fault $argFault is on $node"
        rsh "$node" "$command" >& /dev/null
done
```

Code 5.1: *Script to cause random faults during the WebCom execution*

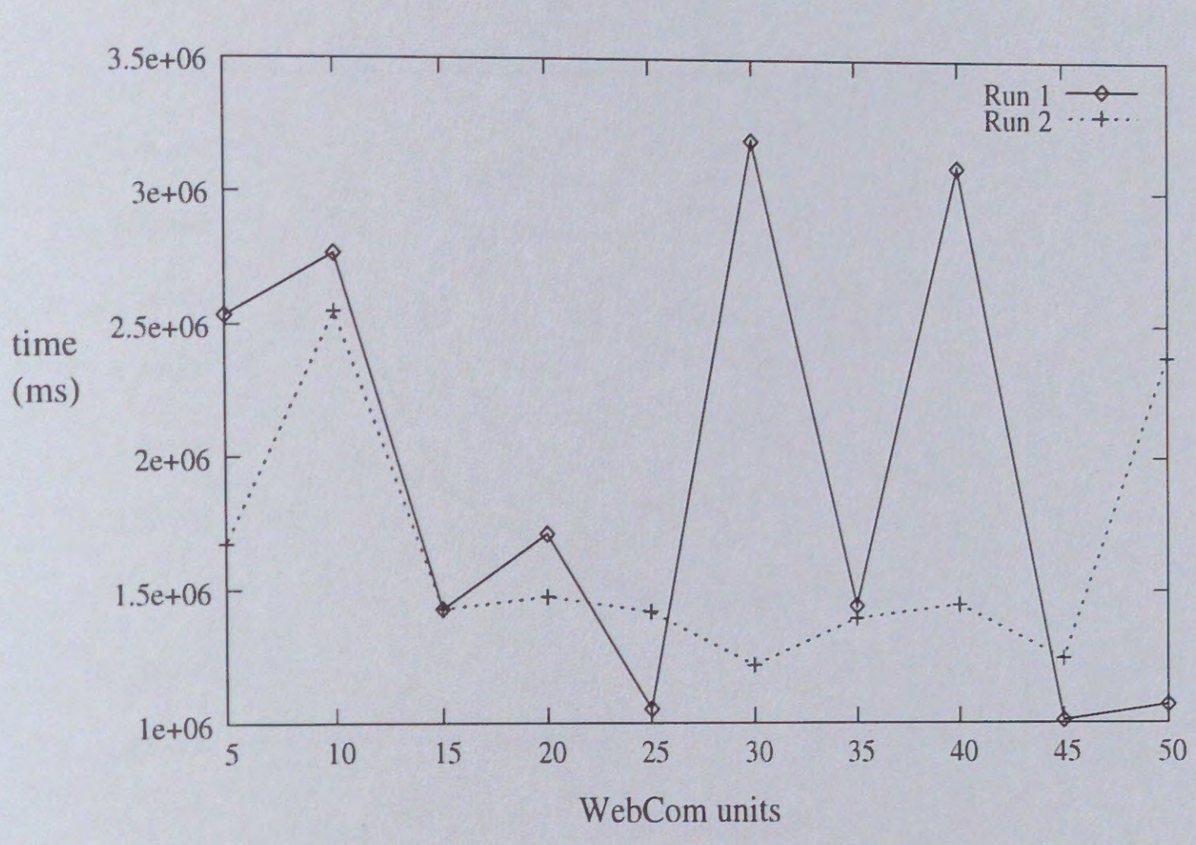


Figure 5.5: *Timing results from two experiments performed where 10% of machines failed during the computation and Fault Masking was employed.*

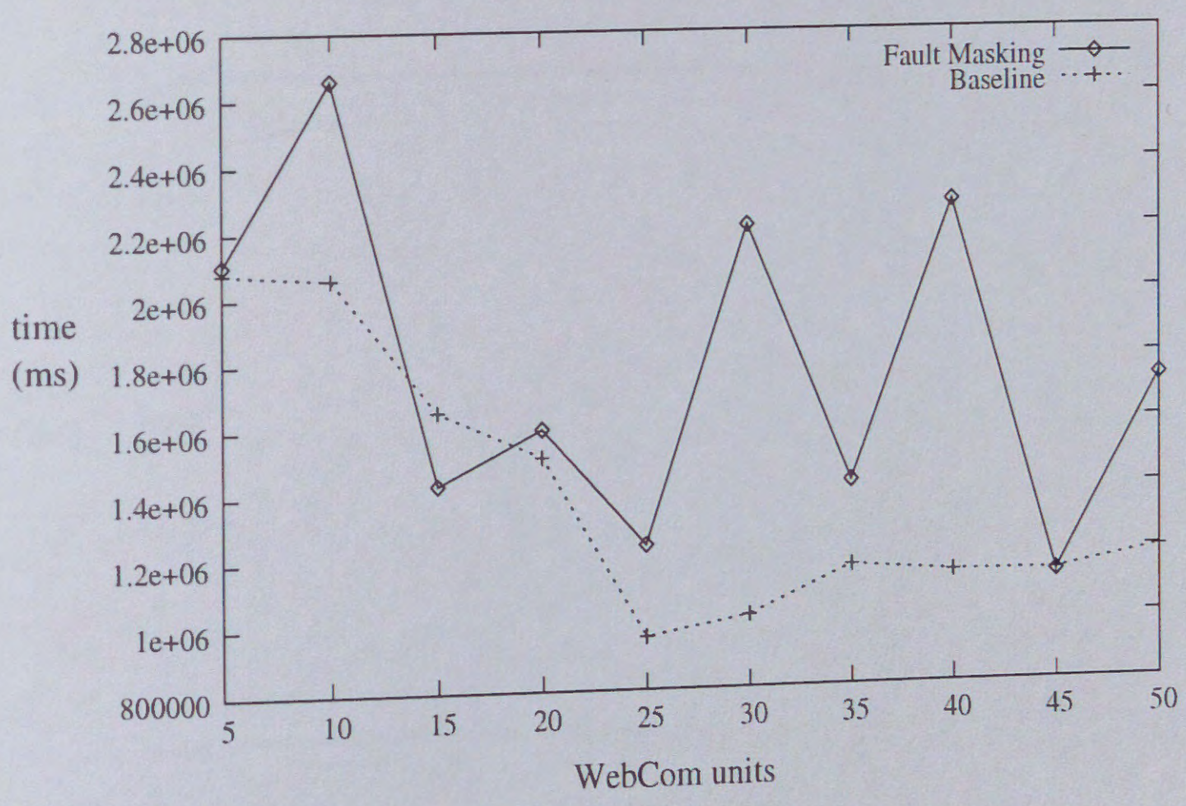


Figure 5.6: *Average results from fault masking experiments compared to baseline.*

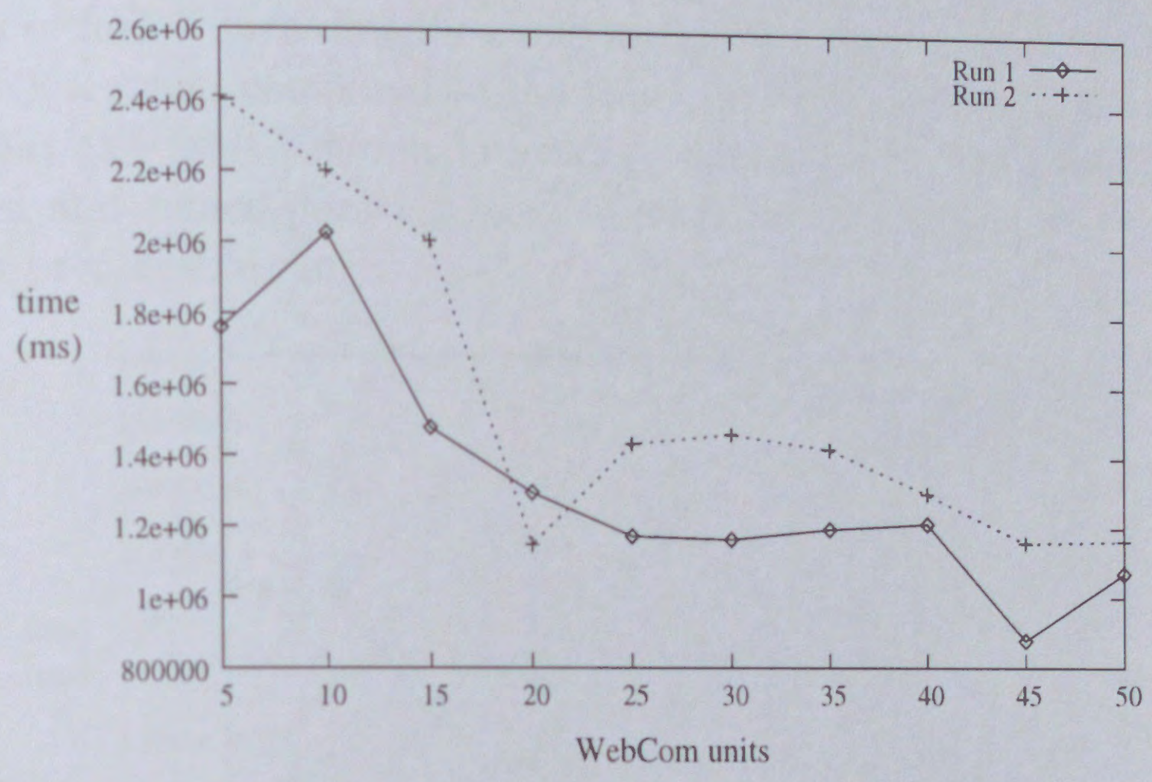


Figure 5.7: *Timing results from 2 runs where 10% of machines failed during the computation and the fault recovery routine was used.*

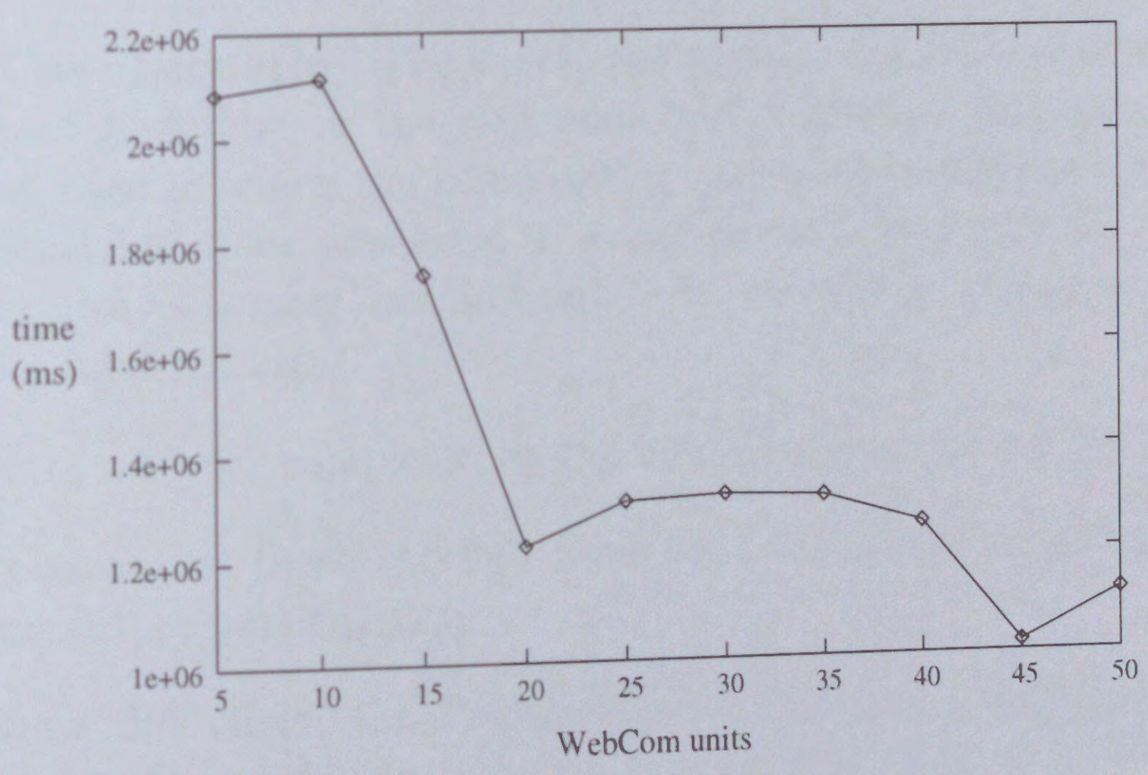


Figure 5.8: *Average of fault tolerance results depicted in Fig. 5.7*

Comparison Fig. 5.9 compares the results from the fault masking experiments to those of the fault tolerance. The fault tolerance trend maintains a curve similar to that of the speedup graphs in fault free executions. The addition of faults does slow the computation but their impact remains consistently low when compared to the trend for fault masking. These results prove that the fault tolerance routine is successful in replacing the failed machines and reincorporating existing work and so keeping the impact of within a predictable range.

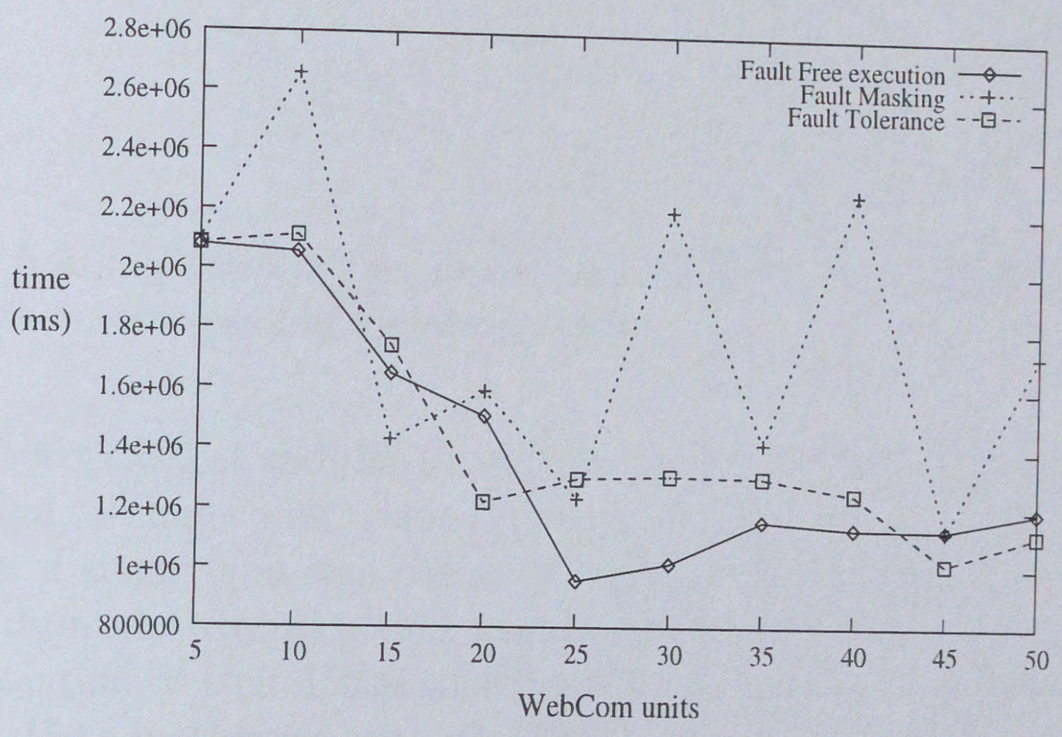


Figure 5.9: *Comparison of average timings from fault masking, fault tolerance and baseline experiments.*

Worst Case Scenario The worst case scenario for fault masking is the single point of failure in the root node which destroys the computation. The worst case in which the computation *survives* is when the bulk of the computation has been scheduled to a remote unit, that unit has acquired all other units as helpers and that unit fails just before returning its result. The losses incurred are:

The loss of all work completed by the failing unit and its children.

The orphaning of those children from the computation, thus preventing their future participation.

To show this effect, the *PowersOfTwo* graph was executed with n = 1000. Twenty-five of the cluster's blades were used. A fault free execution required 18830ms to execute the graph. Fig. 5.10 depicts the state of the topology during this execution just before Unit 156 returns the result of the first condensed node transmitted, i.e., the one that through recursion becomes the instruction with the largest grain size. If Unit 156 fails at this

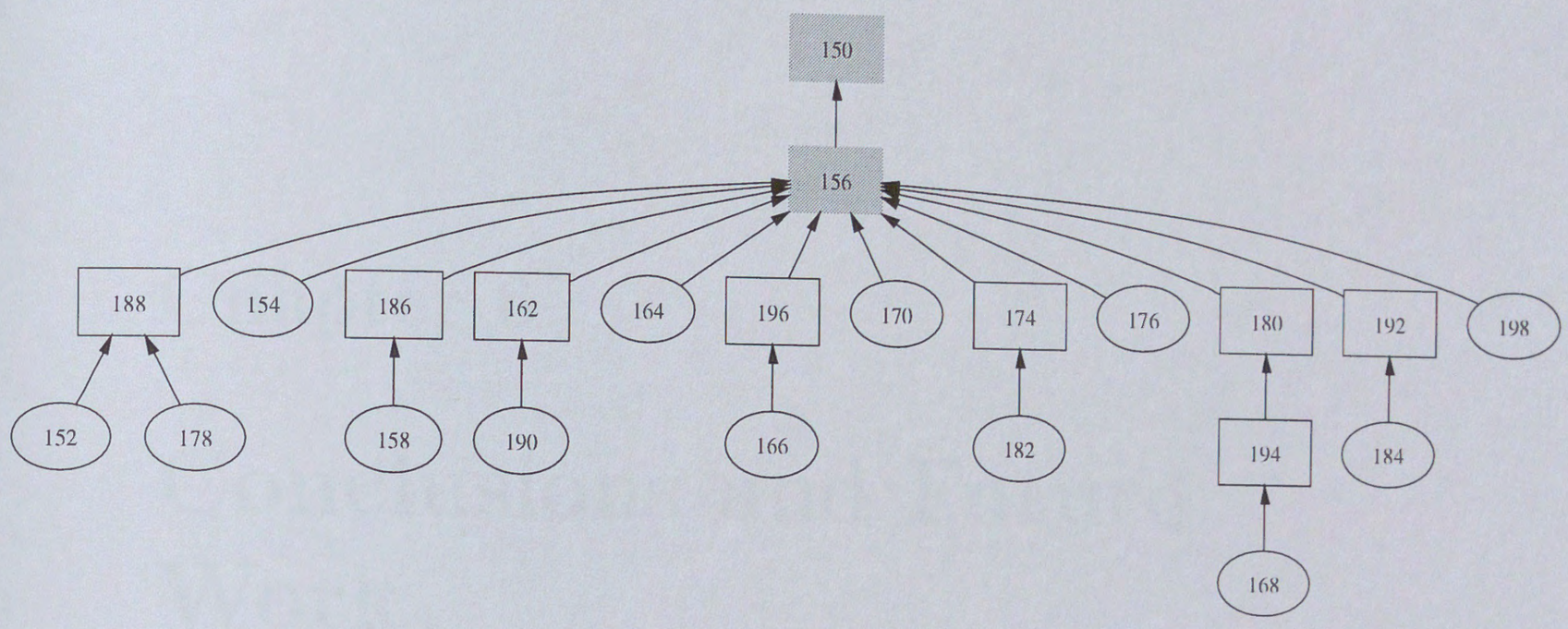


Figure 5.10: *PowersOfTwo execution topology just before Machine 156 returns the main result of the computation*

point, all work is lost and this instruction must be rescheduled by Unit 150. In addition, all units which were clients of Unit 156 will be orphaned.

When a single fault was manually triggered to occur in approximately this situation, the execution time increased to 96182ms. This increase is due to the isolation of Unit 150 from the rest of the units – it must re-execute the Unit 156's instruction without help. If all orphaned units returned to the root node, Unit 150, they could participate in the re-execution of Unit 156's instruction but the execution time would still be almost double that of a fault free execution.

The experiment was performed again with orphaned clients returning to the root node when a fault masking operation takes place. This reduced the execution time to 35488ms, almost double the value for the fault free execution time. This is expected as almost the same number of machines have to repeat almost the same amount of work.

Finally, the experiment was performed with fault tolerance enabled. In this case, the execution time was reduced to 28520ms. With the PowersOfTwo application, which is smaller in scale than the $\mathcal{CG}$ Chess application, the time taken to replace the failure and redirect volunteers to the replacement meant that the computation had completed before all machines could connect to the replacement and contribute their partial result lists.

Chapter 6

Conclusions and Future Work

WebCom is a highly configurable and adaptable machine which dynamically changes to match the environment in which it is executing. That adaption may be precipitated by static knowledge of the ideal execution characteristics of the application. The execution will then proceed on a best effort basis given the characteristics of the implementation and the current environment. WebCom allows developers to effectively harness heterogenous resources and assemble blocks of functionality into complex applications according to a powerful graph based model of computing. Runtime issues such as load balancing, fault tolerance and security are dealt with dynamically by WebCom, thereby isolating the application, and the developer, from such concerns. To accomplish this, WebCom takes advantage of features of the Condensed Graphs model to exploit parallelism in program graphs, to manage its network topology, and to recover in the event of failure.

The architecture and features of WebCom's implementation are presented in Chapter 2. This work represents a complete rewrite from a prototype WebCom developed a number of years ago. Originally, WebCom was based around a single server with a number of helper clients implemented as Java applets. A Condensed Graphs implementation written in C was used on the server to manage program graphs[47]. WebCom used the Java Native Interface library to interact with this early compute engine. Some attempts were made to develope this system into an n-tier system but, given the simple architecture of the prototype, it became too cumbersome to develope in an organised manner. The modular architecture presented here provides many new features, as discussed in Chapter 2 facilitates the introduction of new functionality in an organised manner.

A WebCom system is comprised of one or more units connected to each other via network links. These units are composed of various modules. Metacomputer functionality, such as instruction execution, load balancing

and security, is implemented in appropriate module types. This encapsulation allows the independent development of subsets of the functionality with little or no overlap between modules. A core Backplane entity defines a minimal WebCom system that may dynamically load or unload modules at runtime. Once loaded, modules communicate with each other via a message passing system provided by the Backplane. Any Java object, including actual module implementations, may be sent as a WebCom message. The Backplane's ability to load and unload modules means that they can be deployed at runtime to provide alternate or updated functionality to running WebCom units. This feature, combined with the message passing system, means deployment of a module implementation on one machine can be automatically propagated throughout the entire network topology. The Backplane also acts as the interface between WebCom and the application that instantiates WebCom.

The rest of WebCom's feature set is provided within the different module subclasses. For example, computational tasks represented by WebCom instruction objects are executed by Engine Modules. Various Engine Module implementations may be loaded simultaneously to act as technology bridges and to allow WebCom to utilise different underlying technologies for task execution. A Communications Manager Module allows WebCom units linked through a network to communicate with each other and collaborate on tasks. This collaboration facilitates the effective execution of parallel tasks that have been automatically uncovered by an Engine Module. The allocation of resources throughout the distributed computation is the domain of Load Balancing Modules. A Load Balancing Module on each unit allocates subtasks to connected helper units. Client units may request clients of their own to aid them in their work. In response, the Load Balancing Module can utilise a redirection mechanism to pass on some of its own unit's idle clients. If none are available it can pass the request to its parent. This procedure allows WebCom to automatically evolve the interconnection topology according to characteristics of the program graph.

Security concerns are addressed by a Security Module that employs a Keynote based Trust Management system to ensure that program subtasks (and their associated data and results) are only exposed to authorised machines and users. Assembling these modules together results in a powerful, dynamic and self-regulating (with respect to load balancing and fault tolerance) metacomputing system.

Chapter 3 discusses application development for WebCom's Condensed Graphs Engine Module. The implementation of the multithreaded $\mathcal{CG}$ engine and the code structure of $\mathcal{CG}$ components is presented. A number of sample program graphs are then discussed, including simple numerical computations and a more abstract workflow application. The chapter then focusses on the $\mathcal{CG}$ Chess application that was developed to stress test WebCom. Traditional chess algorithms and their evolution into the parallel Jamboree algorithm

are discussed and the program graphs of the $\mathcal{CG}$ implementation is presented. Some additional $\mathcal{CG}$ constructs are also presented, which were required to build the application and to increase its efficiency. Experimental data is then presented to investigate the parallelism exhibited by the application. The chapter closes with a number of future optimisations to the $\mathcal{CG}$ Chess algorithm. These optimisations would be required to bring it to tournament standard.

Chapter 4 details the functionality of the Fault Tolerance Module. A novel fault tolerance routine, based on a unit replacement routine, is described and Chapter 5 shows the results of experiments performed on the WebCom implementation. To achieve fault tolerance, each unit records some local housekeeping information. In the event of a fault, those machines that are directly connected to the failing machine coordinate amongst themselves to repair the damage in the topology and to recover partial results.

Experimental results, described in Chapter 5, compare the performance of WebCom's fault tolerance to results obtained from fault free executions. An initial baseline speedup, with respect to the number of WebCom units involved, is first obtained. When faults are introduced and a fault masking policy is employed, experiments show that while sometimes the impact of the faults may be small, at other times they may be very large. Further experiments provide empirical evidence that using the fault tolerance routine allows WebCom to maintain consistent execution times, even when random faults occur. This in turn allows execution performance to be predicted based on knowledge of available resources and the general failure rate of those resources. Predictable performance facilitates efficient job scheduling and in applications such as safety critical systems, this quality of service is often of higher importance than simple best case performance. Quality of service is also expected to be an important requirement in the field of Grid computing as companies and virtual organisations provide services to each other within an online economy.

WebCom can be described as an *autonomic* distributed system, i.e., each unit only interacts with its immediate neighbours. Each unit is responsible for its own tasks and performs services, such as instruction execution, for other units. There is no master unit responsible for the whole execution. In such an environment, a centralised fault tolerance scheme is not scalable and the desire to provide autonomic fault tolerance has been documented in the literature[67]. This thesis describes a fault tolerance routine that fulfils this desire and extends WebCom's dynamic and decentralised nature into the domain of fault tolerance.

6.1 Future Work

There are a number of areas in which WebCom may be immediately improved, the first of which became apparent during the development of the Fault Tolerance Module.

Event Listeners The modular architecture employed by WebCom can be enhanced by the addition of event listeners. Currently, there is a small degree of hardcoded overlap between classes in different modules, e.g., Descriptor objects make certain calls to the Fault Tolerance Module. If each module defined a set of events it provides, then other modules that are interested in those events can register as listeners. This system is used quite successfully in Java's *Swing* API[42]. In Swing, objects may listen for events from graphical user interface components, such as buttons, and react accordingly.

Management of events will be incorporated into the Security Module to restrict access to events, this will be required for version control. A particular module may require that listeners to its events be of a certain version to ensure they know how to correctly interpret the event. Module dependencies will form which must be managed by the Backplane and Security Module to ensure modules know how to interact with each other. The Unix RPM (originally called Redhat Package Manager) is an analagous system - installation of a new package may require updating of other installed packages. In the same way, the loading of a new module by the Backplane may require the acquisition of updated versions of existing modules.

WebCom's event listener system can be extended to be useful remotely. This will be a particularly useful tool for Fault Tolerance and Load Balancing Modules. For instance, a unit may register with its parent to be notified when the parent has an idle client (if that unit is authorised to register as a listener for this event).

Secure Naming The Secure Naming system will eventually be used at every level in WebCom. The ability to refer to a particular context as opposed to a direct reference will be useful for almost all interactions between units and modules. Initially, it will isolate most WebCom modules from the underlying network protocol. Names will be translated to network addresses where required by the ConMan. The ability to refer to application graph contexts will simplify reincorporation of results into newly allocated graphs. As discussed in Section 4.2.2, results will simply be "plugged" into the newly loaded graph before execution resumes. This is only possible if the context of a result can be specified, e.g., a result value might be named as "the result of the *SerialSearch* node that takes six of its operands from the *Enter* node in a *SerialSearch* graph with the following operands...".

Reliability Ratings As mentioned in Section 4.3.1, sections of the overall WebCom network may be partitioned according to reliability. These sections are given a rating and may gradually earn or lose trust in terms of likelihood of a fault free execution using their resources. The actual quantification of this likelihood, and its modification over time, must be investigated in order for the Adaptive Lineage Management scheme presented in Section 4.3 to be efficient. The automated judgement of this rating and the associated housekeeping required will be important to WebCom maintaining its quality of service reliability.

Load Balancing The issue of load balancing is of great importance to any distributed system. The load balancing used in this thesis is extremely naïve. Work in a parallel project[55] proposes that WebCom incorporate a framework for load balancing. In this project, various issues such as task scheduling and mapping are addressed and Load Balancing Module subcomponents are specified. Using this framework, WebCom's load balancing will be extended to allow it to efficiently manage computations across loosely connected and geographically divergent resources.

6.1.1 WebCom for Grid Computing

WebCom has been an ongoing project for a number of years now and has shown great potential as a distributed computing platform. WebCom's future direction is moving towards Grid Computing and it has a number of features which will lend themselves well to this field. A Connection Manager Module is under development which will be aware of Grid structures such as *Virtual Organisations*, as well as raw network connections.

Tools such as micropayments and WebCom's advanced security systems will help to create a viable platform for companies to do business on the Grid. It is envisaged that the Condensed Graphs model will prove to be an extremely simple, yet powerful, means to combine both legacy system components and new web service components into complex applications. These applications can then be run on the Grid by WebCom, automatically spreading load, acquiring the necessary resources and managing the computation. In this manner, it is hoped that WebCom will bring the benefits of Grid computing to a much wider audience than currently exists.

Bibliography

[1] Giovanni Aloisio and Domenico Talia. Grid computing: Towards a new computing infrastructure. *Future Generation Computer Systems*, 18(8):v–vi, October 2002.

[2] Dorian C. Arnold, Henri Casanova, and Jack Dongarra. Innovations of the NetSolve Grid Computing System. *Concurrency and Computation: Practice and Experience*, 14(13–15):1457–1479, November/December 2002.

[3] Arvind and Kim P. Gostelow. A Computer Capable of Exchanging Processors for Time. Information Processing 77 Proceedings of IFIP Congress 77 Pages 849-853, Toronto, Canada, August 1977.

[4] Ozalp Babaoglu and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

[5] R. Baraglia, G. Faieta, M. Formica, D. Laforenza, and M. Stiavelli. Metacomputing to solve complex computational problems. A case study: Evolution of galaxies. In H. Liddell, A. Colbrook, B. Hertzberge, and P. Sloot, editors, *High-Performance Computing and Networking (HPCN'96 Europe)*, volume 1067 of *Lecture Notes in Computer Science*, pages 429–440. Springer-Verlag, New York, 1996.

[6] Arash Baratloo, Mehmet Karul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the Web. 9th International Conference on Parallel and Distributed Computing Systems, 1996.

[7] K. E. Batcher. The massively parallel processor (MPP). In *Digest of Papers, Compcon85 Thirtieth IEEE Computer society International Conference*, pages 21–24, San Francisco, CA, February 1985. IEEE Computer Society.

[8] Lubomir F. Bic, Munehiro Fukuda, and Michael B. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, 29(8):55–61, August 1996.

[9] Ken Birman. IEEE Internet computing: Peer to peer: The League of SuperNets. *IEEE Distributed Systems Online*, 4(10), 2003.

[10] Matt Blaze. http://www.crypto.com/trustmgt/kn.html.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. C. E. Zhou, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. 30(8):207–216, August 1995.

[12] Robert D. Blumofe and David S. Park. Scheduling Large-Scale Parallel Computations on Networks of Workstations. In *Proceedings of the Third International Symposium on High-Performance Distributed Computing*, pages 96–105, San Francisco, California, August 1994.

[13] Anthony J. Bonner and Michael Kifer. Transaction logic programming. In *International Conference on Logic Programming*, pages 257–279, 1993.

[14] B. Braunleder, P. Götz, and G. Tanner. Parallel processing with 128 microprocessors. *SIEMENS FORSCH. ENTW. BER.*, 9, 6:330–333, 1980.

[15] P. Cappello, B. O. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. In Geoffrey C. Fox and Wei Li, editors, *ACM Workshop on Java for Science and Engineering Computation*, June 1997.

[16] E.N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Proceedings of the TwentyFourth International Symposium on Fault Tolerant Computing (FTCS-24)*, pages 298–307, June 1994.

[17] R. A. Finkel and J. P. Fishburn. Parallelism in alpha-beta search. *Artificial Intelligence*, 19:89–106, 1982.

[18] Lloyd D. Fosdick, Elizabeth R. Jessup, Carolyn J. C. Schauble, and Gitta Domik. *Introduction to High Performance Scientific Computing.* Scientific and Engineering Computation Series. MIT Press, Cambridge, MA, 1996.

[19] Paul Francis. IEEE Internet computing: Peer to peer: Is the Internet going NUTSS? *IEEE Distributed Systems Online*, 4(11), 2003.

[20] Robert Norman Frank Harary and Dorwin Cartwright. Structural Models: An Introduction to the Theory of Directed Graphs. John Wiley and Sons,1969.

[21] Paul A. Gray and Vaidy S. Sunderam. Metacomputing with the IceT System. *The International Journal of High Performance Computing Applications*, 13(3):241–252, Fall 1999.

[22] W. Gropp, E. Lusk, and A. skjellum. Using MPI: Portable Parallel Programming with the Message-Passing-Interface. MIT Press, 1994.

[23] Gartner Group. The Emergence of Distributed Content Management and Peer to peer content networks. , 2001.

[24] J.R. Gurd, C. C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[25] B. Janssens and W. K. Fuchs. Ensuring correct rollback recovery in distributed shared memory systems. *Journal of Parallel and Distributed Computing*, 29(2):211–218, September 1995.

[26] Christopher Joerg and Bradley C. Kuszmaul. Massively parallel chess. February 12 1999.

[27] **James J. Kennedy** John P. Morrison and David A. Power. Extending WebCom: A Proposed Framework for Web Based Distributed Computing. Workshop on Metacomputing Systems and Applications, ICPP2000.

[28] Simon Peyton Jones and Philip Wadler. Imperative Functional Programming. 20th ACM Symposium on Principles of Programming Languages (POPL'93), Charleston, Jan 1993, pp71-84.

[29] Gerti Kappel, Peter Lang, S. Rausch-Schott, and Werner Retschitzegger. Workflow management based on objects, rules, and roles. *Data Engineering Bulletin*, 18(1):11–18, 1995.

[30] Mehmet Karul. Metacomputing and Resource Allocation on the World Wide Web. Phd thesis, New York University, May 1998.

[31] Michael S. Keller. The cutting edge: Moving to SMP. *Linux Journal*, 71, March 2000.

[32] Akihiro Kishimoto. Transposition table driven scheduling for two-player games. Ms thesis, University of Alberta, Edmonton, Canada, 2002.

[33] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[34] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[35] Santhosh Kumaran and Robert N. Miller. A comparison of parallelization techniques for a finite element quasigeostrophic model of regional ocean circulation. *The International Journal of Supercomputer Applications and High Performance Computing*, 9(4):256–279, Winter 1995.

[36] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[37] Scott T. Leutenegger and Xian-He Sun. Limitations of cycle stealing for parallel processing on a network of homogeneous workstations. *Journal of Parallel and Distributed Computing*, 43(2):169–178, 15 June 1997.

[38] Red Hat Linux. http://www.redhat.com/.

[39] Satoshi Hirano Luis F. G. Sarmenta and Stephen A. Ward. Towards Bayanihan: Building an Extensible Framework for Volunteer Computing Using Java. ACM 1998 Workshop on Java for High-Performance Network Computing, Palo Alto, California, Feb. 28 - Mar. 1, 1998.

[40] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *The International Journal of Supercomputer Applications and High-Performance Computing*, 8, 1994.

[41] Sun Microsystems. Java Programming Language. http://www.java.sun.com.

[42] Sun Microsystems. The swing tutorial, programming graphical user interfaces.

[43] Ronald G. Minnich and David J. Farber. The mether system: Distributed shared memory for SunOS 4.0. In *Proceedings of the Summer 1989 USENIX Conference*, pages 51–60, Berkeley, CA, June 12–16 1989. USENIX Association.

[44] Eugenio Moggi. Computational Lambda-calculus and Monads. Tech. Report ECS-LFCS-88-66, Edinburgh Univ., 1988.

[45] John P. Morrison, David A. Power, and **James J. Kennedy**. Load Balancing and Fault Tolerance in a Condensed Graphs Based Metacomputer. *The Journal of Internet Technologies. Special Issue on Web Based Computing*, 3(4):221–234, December 2002.

[46] John P. Morrison, David A. Power, and **James J. Kennedy**. An Evolution of the WebCom Metacomputer. *The Journal of Mathematical Modelling and Algorithms, Special Issue on Computations Science and Applications*, 2003.

[47] John P. Morrison, **James J. Kennedy**, and David A. Power. A Condensed Graphs Engine to Drive Metacomputing. Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA '99), Las Vagas, Nevada, June 28 - July1, 1999.

[48] John P. Morrison, **James J. Kennedy**, and David A. Power. Fault Tolerance in the WebCom Metacomputer. International Workshop on Metacomputing Systems and Applications. Valencia, Spain, September 03 - 07, 2001.

[49] John P. Morrison, **James J. Kennedy**, and David A. Power. WebCom: A Web-Based Distributed Computation Platform. Proceedings of Distributed computing on the Web, Rostock, Germany, June 21 - 23, 1999.

[50] John P. Morrison, **James J. Kennedy**, and David A. Power. WebCom: A Volunteer-Based Metacomputer. *Journal of Supercomputing*, 18(1):47–61, January 2001.

[51] T. Park, I. Lee, and H. Y. Yeom. An efficient causal logging scheme for recoverable distributed shared memory systems. *Parallel Computing*, 28(11):1549–72, November 2002.

[52] Judea Pearl. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence*, 14(2):113–138, September 1980.

[53] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Best-first fixed-depth game-tree search in practice. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 1, pages 273–279, Montreal, Canada, 1995.

[54] Rinus Plasmeijer and Marko van Eekelen. Functional Programming and Parallel Graph Reduction. ISBN: 0-201-41663-8 Addison-Wesley Publishers Ltd.

[55] David Power. A framework for heterogenous computing, load balancing and programming in webcom. *PhD thesis, National University of Ireland, Cork*, 2004.

[56] D. L. Russel. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[57] Kyung D. Ryu and Jeffrey K. Hollingsworth. Fine-grain cycle stealing for networks of workstations. In *Proceedings of Supercomputing'98*

(CD-ROM), Orlando, FL, November 1998. ACM SIGARCH and IEEE. University of Maryland.

[58] Luis F. G. Sarmenta. Bayanihan: Web-Based Volunteer Computing Using Java. 2nd International Conference on World-Wide Computing and its Applications (WWCA'98), Tsukuba, Japan, March 3-4, 1998.

[59] Jonathan Schaeffer and Aske Plaat. New advances in alpha-beta searching. In *Proceedings of the 1996 ACM 24th annual conference on Computer science*, pages 124–130. ACM Press, 1996.

[60] Jennifier M. Schopf and Bill Nitzberg. Grids: The top ten questions. *Scientific Programming*, 10(2):103–111, 2002.

[61] Larry Smarr and Charles E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[62] J.P. Morrison D.A. Power S.N. Foley, T.B. Quillinan and **J.J. Kennedy**. Exploiting keynote in webcom: Architecture neutral glue for trust management. pages 101–119.

[63] Pedro A. Souto and Eugene W. Stark. A distributed shared memory facility for FreeBSD. In *Proceedings of the USENIX Annual Technical Conference (USENIX-97)*, pages 149–162, Berkeley, January 6–10 1997. Usenix Association.

[64] John Stankovic. Parallel and distributed computing. *IEEETC: IEEE Transactions on Computers*, 36, 1987.

[65] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[66] Dwight Sunada, David Glasco, and Michael Flynn. Fault tolerance: Methods of rollback recovery. Technical Report CSL-TR-97-718, Stanford University, Computer Systems Laboratory, March 1997.

[67] Yoshihiro Tohma. Incorporating fault tolerance into an autonimic-computing environment. *IEEE Distributed Systems Online*, 5(2), Feb 2004.

[68] T. Park Y. Yi and H. Y. Yeom. A causal logging scheme for lazy release consistent distributed shared memory systems. In *Proc. of the 1998 Int'l Conf. on Parallel and Distributed Systems (ICPADS'98)*, December 1998.