J. Functional Programming **4** (1): 19–45, January 1994 © 1994 Cambridge University Press

An overview of the Flagship system

JOHN A. KEANE

Centre for Novel Computing, Department of Computer Science, University of Manchester, Manchester, M13 9PL, UK (*e-mail*: jak@uk.ac.man.cs)

Abstract

The Flagship Project[1] was a research collaboration between the University of Manchester, Imperial College London and International Computers Ltd. The project was unusual in that it aimed to produce a complete computing system based on a declarative programming style. Three areas of a declarative system were addressed: (1) programming languages and programming environments; (2) the machine architecture and computational models; and (3) the software environment. This overview paper discusses each of these areas, the intention being to present the project as a coherent whole.

Capsule review

The Flagship project was a large declarative language research project involving the University of Manchester, Imperial College London and International Computers Ltd. The project was concerned with the design of the entire system, from language and programming environment down to the machine architecture and the system software. The language used in the project was an extension of Hope, called Hope$^+$. A prototype Flagship machine with 16 68020 processors has been built. The system software was designed on the assumption that a Flagship-like system would be commercially available in the early 1990s. This resulted in a complex system, and only a subset was implemented on the prototype. Experimental results are somewhat limited and run-time performance was less than had been anticipated from simulation studies. Nonetheless, many interesting results have come out of the Flagship project. The work is continuing as the European Declarative System, an Esprit II project. This paper is a survey of the overall Flagship project, with numerous pointers into the literature for readers desiring more information about particular aspects of the project.

1 The Flagship project

Flagship was a collaborative research project between the University of Manchester, Imperial College London and International Computers Ltd (ICL). Flagship was unusual in that it aimed to build a complete declarative system. In doing so, it addressed a wider area of activity than many other parallel machine projects. The project involved the following areas:

[1] Flagship was an Alvey project, IKBS 049. This work was partly supported by SERC grant GR/E 21070.

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

- Exploration of the relationship between different declarative language classes and their programming environments. This area involved Imperial College, and is discussed in section 2.
- The design both of a machine architecture for the execution of computations in parallel, and of low-level computational models to serve as targets for translations of higher-level language systems. This involved the University of Manchester. The work on the ALICE machine involved Imperial College. These areas are dealt with in section 3.
- The design of a software environment for the parallel machine. This area involved the University of Manchester, and is the subject of section 4.

Contributions to each of the areas were made by teams at ICL, in particular to the work on developing the prototype machine and to all aspects of the software environment.

The long-term aim of the project (and its successors) is to combine the ease and correctness of programming using declarative languages with the significant increases in performance expected from parallel computers (Broughton *et al.*, 1987). Although the project ended in funding terms in late 1989, work has continued in most of the areas. This paper draws together much of the work connected with the project as a coherent whole to the middle of 1992.

There are two major classes of declarative languages (Darlington, 1987): (1) *functional* languages that are based on the lambda-calculus; and (2) *logic* languages that are based on first-order predicate calculus; the fundamental unit being the relation. The primary example of a logic language is Prolog.

Because of the relatively short time-scale of the project it was decided to concentrate on one class of declarative language. The functional class was chosen for the following reasons:

- **Abstractness:** declarative languages allow an abstract description of a solution to be given. However, with 'pure' Prolog one must often be aware of the procedural interpretation of the language to write programs that will terminate. The declarative meaning is not sufficient, i.e. two programs with the *same* declarative interpretation may have such *different* procedural interpretations that one of them may fail to terminate. With functional languages, equivalent declarative meanings imply equivalent results will be obtained. Even with a knowledge of the procedural interpretation of Prolog, it is necessary to introduce features such as *cut*. Such additions do not have a semantics in first-order logic and constitute 'impure' Prolog. Functional languages do not require such additions.
- **Expressiveness:** first-order languages, by definition, are unable to express higher-order concepts. The higher-order facilities provided in Prolog are built-in, limited and cannot be given a first-order semantics. It is believed that a higher-order facility is necessary for complete expressiveness, and that its semantics should be expressed in the same way as the semantics of the rest of the language.
- **Strong typing:** although strong typing is orthogonal to the class of language, Prolog is an untyped language. Strong typing is viewed as being a necessary part of both a programming methodology and a declarative language. It is felt that

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

functional languages have typing systems in advance of any available in all other language classes.

- **Efficiency considerations:** it appears to be difficult to build efficient, extensible parallel machines that provide the low-level synchronisation and communication logic programs need. It appears to be easier to build machines that support functional languages (Greenberg and Woods, 1990).

2 Programming languages and programming environments

The work on the programming environment had two major aims: (1) the development of a more powerful and commercially viable functional language; and (2) the design and construction of a prototype program development and maintenance system based on the ideas of formal correctness-preserving program transformation. The primary references for this section are Darlington (1987) and Darlington *et al.* (1989*a*, *b*).

2.1 Language development

The basis of this work was the functional language *Hope* (Burstall *et al.*, 1980). Hope includes most features considered necessary in a modern functional language, but was chosen primarily for its familiarity, having been used extensively at Imperial College.

Hope has been extended in various directions to produce more generally applicable languages. This has led to a number of experimental language designs:

- **Hope⁺** (Perry, 1987*a*) is based on Hope with minor extensions for large-scale programming. The aim was to extend Hope so that it could be used to implement the Flagship software environment. *Hope⁺* is the example language used in Field and Harrison (1988).
- **Hope⁺ with Unification:** logic languages have two features that functional ones do not: (1) a logic program makes no commitment to which variables in a relation are considered as inputs and which as outputs; and (2) the results produced by a logic program need not be in ground form, i.e. they may involve variables. In Flagship, logic programming extensions to functional languages have been provided by introducing a programming structure, termed *absolute set abstraction*, which allows sets of values to be specified by the conditions that the members of the set must satisfy. The conditions are expressed as equalities between functional expressions. A full semantics and details of the implementation of *Hope⁺ with Unification* are given in Darlington and Guo (1989). Other work on absolute set abstraction and its functional interpretation is discussed by Liu (1990).
- **SIAN:** there exist classes of problems, usually involving communication with entities outside the computer system, which require an explicit order of evaluation to ensure correct behaviour. An advantage of functional languages is that there is no need to give a precise evaluation order. For functional languages to be used for problems that require an explicit evaluation order there must be some way of controlling the order in which the reductions of the graph representing the program occur. *SIAN*, a language based on temporal logic, allows temporal constraints on the execution of the program to be expressed. A pure functional program together

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

with the temporal constraints is then automatically transformed to produce a simple, augmented functional program that satisfies the constraints. This work is described by Darlington and While (1987).

- **Hope$^+$C:** the absence of side effects in functional languages makes it difficult to express input-output activity. *Continuations* provide a means of systematically expressing such behaviour by putting the non-referentially transparent parts into the underlying system, while the programs remain determinate and referentially transparent. Continuations are essentially functions representing the work remaining to be done in a partially evaluated function application. A continuation may be viewed as a mapping that, when applied to an object representing some 'current state', will yield some 'final state' from which a desired result can be extracted (Field and Harrison, 1988). The language *Hope$^+$C* (Perry, 1987*b*) has extended Hope$^+$ with continuations. Other work, using Hope$^+$ extended with continuations, is discussed by McLoughlin and Hayes (1990).
- **X/Haskell/Hope$^+$ with annotations:** in the early stages of the Flagship project a new language, *X*, was planned that would specifically exploit the machine architecture. Much of this work evolved to be contributions, from members of Flagship at Imperial College and ICL, to the design of Haskell (Hudak *et al.*, 1992). Work on annotations to Hope$^+$ to fully exploit the parallelism of the Flagship machine is discussed by Kewley and Glynn (1990). This work involves experimenting with different evaluation strategies for the language: Hope$^+$ has a mixed evaluation strategy, with strict function application combined with lazy data constructors. This strategy can result both in redundant evaluation and loss of parallelism. The intention of the work on annotations is that such problems can be avoided by allowing user control of the evaluation strategies.

2.2 *Transformation schemes*

Program transformation is the process of converting a program into an alternative, semantically equivalent form that differs from the original in some quantitative aspect, e.g. execution time or space utilization. The transformation capability in the Flagship programming environment seeks to give formal, mechanized interactive support. The *algebraic* approach to transformation has been concentrated on because it is considered to have a number of advantages over the *unfold/fold* transformation methodology. In particular, it is felt that the algebraic approach allows greater scope for automation. There are several transformation schemes that have been developed in the project based on the algebraic approach, the following describes two of these.

Memoisation is a method to assist in the efficient implementation of non-linear functions. A *memo-function* (Michie, 1968) is a function that retains details about all the arguments to which it has been applied, and the corresponding results. When a memo-function is re-applied to an argument, the argument is used to look up a table for the corresponding result. The saving in computation from using memo-functions is obvious. The major difficulty is knowing when an entry can be deleted from the table, i.e. when the memo-function will no longer be applied to a particular argument. The work reported by Khoshnevisan (1988) has been concerned with developing a

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

space-efficient implementation of memo-functions. This technique is claimed to achieve improvements comparable to existing program transformation schemes.

As discussed earlier, functional languages lack the facility of logic languages to use relations in different modes, i.e. variables in functions must be defined as either input or output. One approach is to augment functional languages with such facilities, as for example in *Hope$^+$ with Unification*. However, supporting unification is less efficient than supporting the reduction system used for functional languages. An alternative approach is to use *function inversion*. A method for automatically generating the inverses of many first-order functions is described by Harrison (1987). This method requires extensions to functional languages to include logical variables and function-level unification to express inverses. The logical variables are removed from the program at transformation-time and thus the execution mechanism remains conventional reduction. *InvX*, a mechanized system for function inversion, is described by Khoshnevisan and Sephton (1989).

The Flagship programming environment provides support for the development, implementation and modification of programs written in Hope$^+$, and various program analysers and transformation tools. It is believed that this represents one of the first attempts to construct a formally based, but practical, transformation system and embed it inside an execution environment.

The environment was completed in two years, and became available as an integrated whole in June 1988. The environment is almost totally written in Hope$^+$. This development, in such a relatively short time-scale using limited resources, appears to support the claims made about the advantages of using declarative (specifically functional) languages for software development.

More recent work in the general area is discussed by Darlington *et al.* (1991).

3 Machine architecture and computational models

The operational semantics of functional languages is to regard equations as rewrite rules that permit elements of an expression that match the left hand side of an equation to be replaced with the corresponding right hand side. Each such rewrite represents a change of *form* in keeping with the equations, i.e. each change preserves the meaning (Kennaway and Sleep, 1984).

The reduction model of computation was considered to be the most appropriate one for parallel execution. The model can be summarized as followed (Peyton Jones, 1987):

1. Executing a functional program consists of evaluating an expression.
2. A functional program has a natural representation as a tree (or more generally a graph).
3. Evaluation proceeds by means of a sequence of simple steps called reductions. Each reduction performs a local transformation of the tree/graph.
4. Reductions of redexes at different places in the expression (tree or graph), that exist at the same time, may safely take place in a variety of orders, or in parallel, as they cannot interfere with each other.
5. Evaluation is complete when there are no further reducible expressions.

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

There are two distinct types of reduction model (Kennaway and Sleep, 1984). With *string* (*tree*) *reduction*, when a function is applied to an argument, a copy is made of the function body with a copy of the argument substituted for each occurrence of the formal parameter. FFP (Mago and Stanat, 1989) is a parallel string reduction machine. In such a machine the distribution of reducible expressions is relatively easy because they are all self-contained and, as a result, there is no need for a globally addressable memory space. String reduction systems generally evaluate expressions eagerly, reducing arguments to normal form before supplying them to functions. When infinite data structures, and thus lazy evaluation, are considered string reduction becomes unacceptable because of the re-evaluation of expressions that is necessary.

With *graph reduction* pointers to sub-expressions are copied rather than the full text. The expression being evaluated is stored as a graph. On a parallel machine the sharing of sub-expressions means that a globally addressable memory space is required for full flexibility in distribution of reducible expressions. The alternative is to only distribute self-contained sub-graphs as in the ZAPP machine (Sleep and Kennaway, 1984). *Combinator reduction* is a special case of graph reduction in which all functions are combinators. A *combinator* is defined as a lambda expression that contains no occurrences of a free variable (Peyton Jones, 1987). A transformation using combinators by which all variables are removed from a functional program is described by Turner (1979). Each combinator has a fixed rewrite rule associated with it. Because combinators are higher-order functions having no free variables in their bodies there is no need for an environment holding values associated with free variables. Furthermore, combinators can be reduced in parallel. A problem with these types of fixed combinators is that the granularity of each reduction is extremely small. *Supercombinators* generalize the class of combinators (Hughes, 1982). They are not fixed but are compiled from user-defined functions. The granularity of the reduction of a supercombinator can be much larger than a fixed combinator thus increasing efficiency. In addition, a supercombinator is 'pure code' so its internal structure is of no importance, and any suitable, efficient representation may be used.

In the Flagship system a functional program is represented as a graph of supercombinators.

3.1 Major influences on the Flagship machine

The Flagship machine was developed as a synthesis of dataflow and reduction principles, allied to techniques used to gain cost-performance in conventional approaches. In this section, dataflow principles are summarized and the ALICE reduction machine is discussed.

The *dataflow* model of computation is based on dataflow graphs. Nodes of the graph contain operations, and the arcs of the graph allow tokens containing values to flow from the operations that produce them to the operations that consume them. Operations can execute when they have the values they require, and can do so in parallel with other operations without limit (Kirkham, 1990). Dataflow languages can be regarded as first-order functional languages.

Because dataflow is entirely data-driven, machines built to accommodate the model

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

only allocate resources when computation is required. Thus a processor allocated to a task can proceed to perform the task without having to pause while required input becomes available: all input to a task is available before allocation and when the task is performed an output is produced. Therefore, the machine structures require only uni-directional communication. As a result such machines are highly extensible with linear speed-up characteristics. However, the dataflow model is regarded as having two limitations (Watson *et al.*, 1986):

- **Expressiveness:** pure dataflow is considered to be an inadequate general-purpose computational model, for operating systems for example, because it has no concept of storage. Consequently, to express such systems there is a need to introduce either explicit storage, that loses the functional nature of the model and also introduces the need for access synchronization, or infinite structures that require demand-driven evaluation that is not a natural part of the dataflow model.
- **Cost-effectiveness:** the tagged dataflow model used by most practical machines, e.g. the Manchester Dataflow Machine (Kirkham, 1990), requires data to carry a large amount of context information. When physically implemented this not only requires a large amount of store but, more importantly, the token-matching requires an *associative mechanism* so that the availability of data for computation can be detected. In the Manchester Dataflow Machine this was implemented using a hardware hashing mechanism that is both complex and expensive. Further, because of the very fine granularity that most dataflow machines have used, the costs of scheduling and communication are very high.

The Applicative Language Idealized Computing Engine (ALICE) parallel graph reduction machine (Darlington and Reeve, 1981), designed at Imperial College, was the major influence on the Flagship machine. Although the design of ALICE predates Flagship, the prototype ALICE machines were built, and experimentation carried out, as part of the project.

The abstract architecture of ALICE (and all reduction machines) is a collection of reducible tasks and a collection of reduction agents that can access these tasks. In the physical machine the reducible tasks are graphs of packets held in separate, distributed store, and the reduction agents are processing elements connected to the store via a delta network (see Fig. 1). A delta network is a *multistage shuffle-exchange*

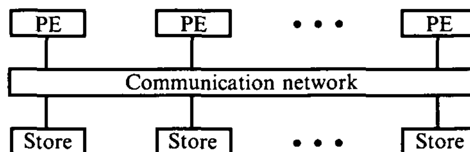

Fig. 1. The ALICE physical machine architecture.

network. It has the total connectivity of a crossbar network with better cost scalability but increased network latency (Almasi and Gottlieb, 1989; Greenberg and Woods, 1990).

The reduction model is a slight variant of supercombinator reduction. The process size on the ALICE machine is somewhat larger than dataflow being at the level of a

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

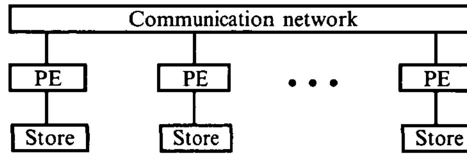

Fig. 2. The Flagship physical machine architecture.

language function definition. ALICE uses code-copying, and hence there is no need for associative mechanisms.

The major limitation of the ALICE approach is that the granularity of a rewrite is at the level of a programming language function. Most functions contain conditional computation thus most rewrites will involve access to remote store that results in the rewrite being suspended while the required packets are fetched. The suspending and restarting of rewrites make it unlikely that ALICE would be linearly extensible beyond a relatively small number of processors (Watson *et al.*, 1986).

3.2 *The Flagship machine*

Flagship is a parallel, data-driven, graph reduction machine. In a data-driven execution model resources are allocated only to processes that have a complete set of inputs, and this condition is achieved by the arrival of data at a point where other process information is stored. Such data-driven properties are considered essential to an efficient parallel machine implementation. A machine with such a data-driven property should be capable of exploiting the principles that make dataflow machines linearly extensible (Watson *et al.*, 1986).

The following sections discuss the representation and rewriting of programs as graphs on the machine, and the physical architecture. The primary references for these sections are Watson *et al.* (1989) and Watson (1990).

3.2.1 *Model of graph representation and rewriting*

A program to be evaluated on the machine is compiled into a graph. This graph is reduced according to rewrite rules until a normal form is reached that represents the result of the computation. The nodes of the graph are represented as *packets* in the store of the machine. Conceptually, all packets contain an *identifier* by which they can be referenced by other packets. Most packets consist of a *header*, containing housekeeping information, and a number of *item* fields, containing atomic values or pointers to other packets. A packet header contains its *type* and its *state*. This *state* denotes what stage of reduction the packet is in:

- **Active** – the packet is a candidate for reduction. An active packet is the root of a rewritable sub-graph (RSG), i.e. a reducible task in the abstract architecture.
- **Suspended** – the packet will become active once the evaluation of its arguments has been completed.
- **Dormant** – the packet is an application that has not yet been reduced.
- **Ground** – the packet is in ground form and cannot be reduced further, e.g. packets holding integers or code.

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

A reducible task, represented by RSG, is selected and then reduced. The reduction involves examining the type field of the root packet and then rewriting the packet and its arguments according to the 'rules' associated with that packet type.

3.2.2 Physical architecture

The physical architecture of the machine consists of a set of closely-coupled processor (PE)-store pairs connected together by a high performance delta network (see Fig. 2). The difference from ALICE is motivated by the observation that high performance in conventional machines is obtained by a close coupling between store and processors. The reducible tasks are divided among the PE-store pairs in the physical architecture. Each PE can directly access that part of the store to which it is closely-coupled. Access to a non-local store is achieved by sending a request message to the PE coupled to that store. Each PE simplifies the RSGs contained in its own local store. The computational graph is distributed dynamically among the PE-store pairs as evaluation proceeds; details of load balancing on Flagship can be found in Sargeant (1987).

Although the store is physically distributed, it is globally addressable by any PE because it is felt that a more flexible and general-purpose parallel architecture results from not placing any restriction on the way a computational graph can be distributed across the physical machine. One immediate consequence is that the potential parallelism is increased. With this approach it is essential, for efficiency, to localize activity as much as possible. Mechanisms to exploit locality on Flagship are discussed in Section 3.2.4.

A consequence of global addressability is that garbage collection must to an extent also be global. Two new garbage collection schemes were proposed for Flagship: (1) weighted reference counting, which removes the synchronization necessary for ordinary reference counting in a distributed system (Watson and Watson, 1987); and (2) real-time process-based mark-scan (Derbyshire, 1990). A combination scheme using weighted reference counting for general garbaging, with mark-scan acting as a background collector for garbage that reference counting was unable to detect, was also proposed (Derbyshire, 1990). On the prototype machine, two schemes were embodied in different run-time systems either of which could be selected when the machine was loaded: (1) ordinary reference counting on a per-PE basis, with weighted reference counting between PEs; and (2) the mark-scan scheme. No comparative performance figures are available.

The Flagship global addressability approach contrasts with the ZAPP machine (Sleep and Kennaway, 1984) where only self-contained sub-graphs can be exported to other PEs. In this case, all accesses to store are necessarily local, and garbage collection can be performed locally. However, potential parallelism within the self-contained sub-graph is lost.

Although it is desirable to localize all arguments to a rewrite, the Flagship machine must also deal with the case where an argument is in a remote store. Consequently, before a PE begins a rewrite, all packets within the rewritable sub-graph (RSG) must be localized on that PE-store pair, where the RSG is defined as a graph of packets that

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

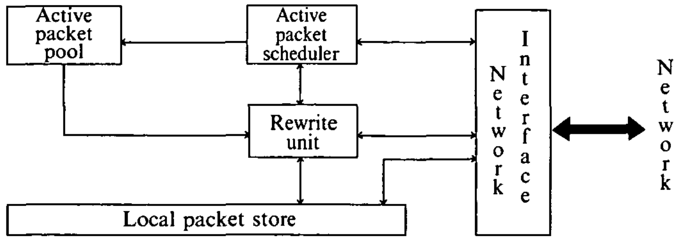

Fig. 3. Flagship PE-store pair.

must be available to a PE performing a reduction (Watson, 1987*b*). The root packet of the RSG contains information to allow the machine to bring together all of the sub-graph into a single PE-store pair before rewriting is attempted. Usually, this pair will be the one in which the root of the RSG is located, however, rewrites involving state are handled differently (discussed in Section 4.1.1). Autonomous hardware connected to each PE can organize this gathering while the PE is reducing another sub-graph, and so the performance of the machine is, to an extent, independent of the latency of the network provided there are enough RSGs to keep the machine busy.

A most important consequence of all packets being available locally when a rewrite is begun is that a rewrite on the Flagship machine is an indivisible action, i.e. either completed or not yet started. This overcomes the major limitation of the ALICE design where rewrites can be suspended. This feature of indivisible rewrites is discussed in more detail in Section 4.1.1.

3.2.3 Implementation of graph representation and rewriting

A simplified block diagram of a PE-store pair is given in Fig. 3. The local packet store holds the part of the packet pool that is local to the PE. The active packet pool contains currently active locally-held packets. The rewrite unit is essentially a conventional sequential processor. The primary references for this Section are Watson (1987*b*, *c*).

A packet is physically just a collection of contiguous storage locations in a linearly addressable store. In practice, the identifier of a packet is the address where it can be located in this store. Similarly, an active packet is recognized by the presence of its address in the active packet pool of the local PE. The action of rewriting an RSG involves the following:

1. An active packet is obtained from the active packet pool, and the rewrite unit performs the actions appropriate for its packet type.
2. If the rewrite generates new active packets, they are allocated positions in the local packet store and the active packet scheduler is passed their addresses. The active packet scheduler determines when and where a packet is to be processed.
3. When a rewrite results in more than one new active packet then their associated RSGs can be reduced in parallel. Work can be exported to other PEs via the interconnection network. When a PE receives an active packet it is added to its own active packet pool.

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

The implementation of rewriting is split into two levels: (1) the basic execution mechanism; and (2) the computational models. This enabled the two levels to be developed independently.

Basic execution mechanism

The basic execution mechanism defines the primitives of the machine: (1) the packet representation of graph nodes; (2) packet states; (3) the firing of packets; (4) the returning of values as the result of evaluations; (5) primitive packet types; (6) garbage collection; and (7) the localizing of the RSG.

Computational models

A computational model for a language defines the packet types of the model being supported, how expressions in that model are represented, and how the building blocks provided by the basic execution mechanism are used to rewrite those expressions.

DACTL, the Declarative Alvey Compiler Target Language (Glauert *et al.*, 1989), is the target intermediate language for Flagship. DACTL can be used as a specification language for computational models for Flagship because the basic execution mechanism aims to implement DACTL. Several different models can be supported simultaneously because they can use different packet types.

The MONSTR model (Banach and Watson, 1989), discussed in Section 4.1.1, supports extensions to declarative programming. A packet-based model for Prolog (Syed-Mustaffa, 1992), and a coarse operation-grain packet-based model for a parallel object-oriented language (POOL) (Sharifi, 1991), have also been developed.

The models for functional languages include one based on the lambda-calculus (Watson, 1986), one based on SKI combinators (Cheese, 1990) and one based on supercombinators (Watson, 1987*c*). The *supercombinator* model is between 10 and 100 times more efficient than the others because it increases the granularity of the reductions, and it reaches the result in fewer reductions. As a consequence, it is used to implement the Hope$^+$ language. The source program is compiled into a set of supercombinator definitions and an expression to be evaluated that contains the application of supercombinators to arguments. The expression is then reduced to Weak Head Normal Form, i.e. with no top-level reducible expressions.

The model can be considered to contain two sub-models. The first sub-model provides a general reducer that can reduce an application of a supercombinator to arguments independent of how the arguments are split into packets. This general reducer is capable of handling higher-order functions. The second sub-model is a reducer for a restricted type of application in which a function is applied to its arity of arguments. This is just a special case of the general reducer but, as it is the most common, it is important to deal with it as efficiently as possible.

In general, a rewrite may need to distinguish between strict and non-strict arguments, and to evaluate in a demand-driven fashion both its non-strict arguments and some of its strict arguments, e.g. lazily evaluated lists. A field in the packet header

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

determines the number of strict arguments. Pointers to argument packets are tagged with a bit that indicates whether they are evaluated or not. Unevaluated arguments can be evaluated by firing the argument's packet. Firing involves activating any argument packets that are not already active.

The *general reducer* is used where code is to be planted for an unknown function applied to arguments. The arity and strictness of the arguments of the function are not known at compile-time. This lack of arity information is a problem because a supercombinator cannot be executed unless all the arguments it needs are available. The model uses IPAP (Incomplete Processable APplication) packets for applications of unknown functions. An IXAP (Incomplete eXecutable APlication) packet is used for the application of a function to less than its full arity of arguments. When an unknown function is rewritten the function field is evaluated, as are the arguments known to be strict. A suspended Build packet is left waiting for the results of evaluating the function and any strict arguments. When the Build packet becomes active, further checks are required to ensure that the number of arguments in the Build packet are sufficient for the supercombinator to be executed.

The *special reducer* uses two types of application packet: XAPP (eXecutable APPlication) and PAPP (Processable APPlication). Data-driven evaluation is achieved by using the XAPP packet to represent a function application. The XAPP contains a pointer to a CODE packet (the function), and pointers to the arguments. Demand-driven evaluation is achieved using the PAPP packet type. A PAPP fires its unevaluated strict arguments and turns into a suspended XAPP that becomes active when the evaluated arguments are returned.

Data structures can be built using Constructor packets provided by the supercombinator model. A Constructor packet when activated merely returns a pointer to itself.

The execution process is a combination of novel and conventional computational mechanisms. At one level there is a process of graph reduction where parallel synchronization and dynamic scheduling of any available rewrite is provided by the packet-based graph. At a lower level, all the detailed computation that involves store access and data manipulation to modify the graph is executed with the efficiency of serial code.

The failure of one or more of the PEs of the machine should be tolerated, such that the behaviour of the system will be the same. One approach to *fault-tolerance* has concentrated on the specification and verification of an algorithm that can ensure the successful execution of a program in the presence of the failure of a PE of the machine. To ensure that a recovery can be made from a failure each node of the graph is replicated on another PE. The objectives of the algorithm are to recover the computational graphs that are lost due to PE failure. The specifications of the algorithm, the graph reduction model and the *enhanced* graph reduction model that can tolerate the failure of a PE are described formally using CSP notation (Hoare, 1985). The algebraic transformation rules of CSP are used to prove that, in the presence of PE failure, the fault-tolerant graph reduction model behaves correctly. Greater detail on the fault-tolerant graph reduction model can be found in Ye *et al.* (1990).

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

3.2.4 Mechanisms to exploit locality

The Flagship packet abstraction is mapped onto a set of contiguous addresses in a physical linearly addressable memory, i.e. a packet corresponds to a 'block of memory'. The links between packets, i.e. the arcs, are pointers to (the addresses of) the appropriate packets.

Functional programs, in general, tend to manipulate large data structures. As a consequence most rewrites will contain references to data. When a copy of a packet representing data is made to a store, it is possible to use it in the requesting rewrite and then discard it. Such a method is safe, but ignores the *working set* model of program behaviour, i.e. data access tends to be grouped into small localities of address space. In functional programs, where the execution of a function is performed as a series of rewrites as on Flagship, the working set model manifests itself in a high probability that a child rewrite may require access to a packet copied as a result of a request from its parent rewrite (Greenberg and Woods, 1990). If the original copy has been discarded then another copy will have to be made, thus wasting resources. If a copy of a packet p is not discarded then further rewrites that are scheduled on that PE that reference packet p should use the existing local copy of p. Another copy will not be made although, if necessary, a local copy can always be garbage collected and re-copied when a subsequent request is made.

In Flagship, the working set model is exploited by each PE having a cache in which remote, global addresses are mapped to local addresses. Caching is a well-known technique for increasing locality by keeping copies of recently-referenced data so that the copy is rapidly available if the same data are referenced again. Cache coherency and the problems it introduces are not relevant in a pure graph reduction machine due to the property of referential transparency. Unevaluated subgraphs are evaluated before they are copied to avoid the potential for re-evaluation (Greenberg, 1989).

The cache is a very general mechanism that ensures that any packet that is shared globally (via its identifier) is also shared locally. Before a copy of a remote packet is requested the local cache is checked to see if a local copy already exists. Thus, copies of remote packets are made only once, although local copies of packets can be garbage collected if necessary.

The major problem with the cache mechanism is efficiency: the mechanism requires an association to be made between the global address and the local address. Such a calculation is expensive to implement and is required on every access. On the Flagship prototype machine, the cache is implemented in firmware as a dynamic hash table that adds to the inefficiency.

A complementary approach has been adopted based on the following analysis: information that a functional program contains can be divided into two categories:

- Information that is *static* at the language level but *dynamic* in the way that local copies of packets representing this information are made in individual PEs, e.g. function definitions and constant data (the term *function definitions* will be used to refer to both).
- Information that is 'dynamic' in both senses, e.g. data created at run-time.

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

By exploiting the static information, a fixed relative address can be allocated at compile time for each function definition. This address provides sufficient information to indicate where on each PE the copy of that function definition *will* be held at run-time. The fixed relative address is an index into a table held on each PE. The entry associated with an index is the address where a packet, representing a function definition, will be held at run-time. The table is termed the Static Copying Environment Table (SCET). A SCET is a tuple (*Master SCET*, *set-of*(*Local SCET*)). The Master SCET and each Local SCET are located on different PEs. Each Master and Local SCET consists of (*SCET index*, *SCET entry*) tuples. Each SCET entry is a tuple (*global-address*, *local-address*). The global address indicates where the master copy of the packet associated with this index is stored. The local address indicates where the local copy, if any, of this packet is stored. When a SCET index is accessed, if the local address contains a *NULL* value then a local copy does not exist, and a copy is made from the Master SCET. The local SCET is updated appropriately with the address of the local copy, and any subsequent references to that SCET index on that PE will make use of the local copy. The SCET look-up involves a *base address + offset* calculation rather than the more complex calculation required in the cache scheme. The basic execution mechanism of the machine does not distinguish between a pointer containing a 'real' address and the 'indirect' address that is a SCET index (pointer).

Evidence from other projects suggests the benefits of locality mechanisms. Simulation figures for Rediflow (Keller, 1987) (a parallel graph reduction machine with a similar physical architecture to Flagship) suggest that, typically, between 5% and 20% of data references generated by a processor were to a non-local physical store (quoted by Almasi and Gottlieb, 1989). The combination of both mechanisms, the SCET to deal with constant data and the cache to deal with run-time created data, should reduce the percentage of non-local references to data. The SCET also provides efficient access to function definition, i.e. code. It is encouraging to note that the 'rule of thumb' adopted in the IBM RP3 parallel machine project was that two-thirds of *all* memory references are instructions, i.e. references to code (Almasi and Gottlieb, 1989). Thus, the effort spent on lowering the access time to local copies of code by using the SCET mechanism should further improve efficiency.

This section is based on Watson (1987*a*) and Keane (1989, 1990*b*).

The prototype Flagship machine: Structure and performance

A prototype machine was built at ICL, consisting of 16 68020 processor boards, each with 16 megabytes of store, connected by a custom-built, delta network (Townsend, 1987).

There are few detailed performance figures for the prototype. Figures for the following 'benchmarks' are available: *nqueens*, *Tak* (a recursive algorithm with cubic explosion), *Triangle* (a board game) (the figures for these three benchmarks are given as run-time in seconds) and a *Transaction Processing System* (*TPS*) (the figures for which are given as the number of transactions per second, *no.t.p.s*). The Flagship figures are from programs in Hope^{+}, and programs hand-coded in the Flagship

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

assembler-like language, IIS. The figures, in comparison with various other systems, are given in Table 1 (Robertson, 1990; Trinder, 1989).[2]

Table 1. *Flagship figures*

Machine	Language	*nqueens*	*Tak*	*Triangle*	TPS – *no.t.p.s.*
Flagship	IIS	1	18	80	140
Flagship	Hope+	11	172	2400	45
Sun 3/260	Hope+ (FPM)	1·1	6·9	387	—
Sun 3/50	Hope+ (FPM)	3·5	19·1	1088	5
iPSC 16 nodes	CCLISP	—	—	68	—
DEC VAX 8830	—	—	—	—	27
IBM 4381-P22	—	—	—	—	22·1

It is believed that the performance of the Hope+ versions could be considerably improved. The performance figures for the Transaction Processing System are impressive, and indicate that a Flagship-type machine may be suited as a relationship database engine.

Overall, the performance of the machine was less than anticipated from simulation results because of: (1) the medium granularity of the rewrites (Watson and Watson, 1990); (2) the overhead of the scheduling requirements imposed by the system software; and (3) the work distribution mechanism, that is considered to be overly sophisticated (Peyton Jones *et al.*, 1989).

Very large grained rewrites appear to solve this problem (this work is part of the EDS project; see Section 5). They have two effects: (1) the amount of work done by the rewrites becomes larger (in the general case); and (2) there is a less frequent need for scheduling, thus reducing this cost because it occurs less often.

A problem with increasing the granularity is that some potential parallelism is likely to be lost, and this reduces the possibility of near-linear extensibility. However, all potential parallelism is made available at run-time and performance is improved either by combining serial parts of computation into single larger grain parts, or by dynamically combining parallel parts based on information of the current load-balancing in the machine (Watson and Watson, 1990).

More recent work on large-grain rewriting is described by Sargeant and Watson (1991).

The prototype ALICE machines: Experimentation

In the early stages of Flagship the first ALICE prototype was built by ICL and delivered to Imperial College in June 1986 (Townsend, 1988). Two further prototype machines have been constructed.

ALICE has been used to assess the value of the transformation activity that improves the efficiency of the execution of functional programs on different type of parallel machines. An example of this type of machine-specific transformation would

[2] FPM (see Table 1) is a stack-based abstract machine developed at Imperial College (Field and Harrison, 1988).

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

be data structures that are most naturally accessed sequentially, e.g. lists. This sequential access means that there is little possibility for parallel computations. Such data structures can be transformed into equivalent, distributed structures that give the potential for increased parallelism (Darlington *et al.*, 1989*b*).

ALICE has been used to measure the effect of transformations of this sort on the execution time of some example functional programs. ALICE has also been used to predict the effect of such transformations on the execution of the same set of programs on the Flagship machine despite the difference in physical architecture (Darlington *et al.*, 1989*b*). Performance figures for the ALICE machine are discussed by Reeve and Wright (1990).

4 Software environment

In this section, the software environment of the Flagship machine is described. A development route to assist in the design and implementation of the software environment is also discussed.

It is unusual in a parallel graph reduction machine project to concentrate effort into the design of a software environment. The motivation came from the belief that, despite the properties of declarative languages and the availability of specially tailored engines for their efficient execution, such systems would not generally be used unless they provided an environment similar to that of conventional machines (Broughton *et al.*, 1987).

It was hoped that functional languages would help in the very difficult area of parallel operating system development and that the implicit parallelism of the languages would increase efficiency. Thus the software environment was implemented in a *declarative style* using Hope^{+}.

4.1 Dealing with non-determinism

The major problem in using functional languages for operating systems is that the languages are *deterministic*, whereas operating systems must handle *non-determinism*. Extensions to functional languages to implement operating systems are surveyed in Jones and Sinclair (1991); very few of the systems described were specifically designed to exploit a parallel machine.

The Flagship approach is that it is both natural and efficient to express the behaviour of operating systems using a collection of state variables that are updatable over the course of time, and whose value at any instant represents the state of the computation at that instant (Banach *et al.*, 1988). This led to an object-oriented approach allied to the declarative style.

The unit of abstraction, design and decomposition for the system software is the Flagship ADT (FADT). FADTs are effectively *objects* with the following characteristics (Leunig, 1987):

- Each instance of a FADT can have its own state. The state is encapsulated in that access to it, from outside the FADT, can only occur via the interfaces provided. State is introduced as part of the basic building block; its encapsulation is based

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

on the principle of information hiding. A mechanism is provided to ensure the consistency of the state.

- A 'state transition function' style of definition. The general form of an FADT operation will be $\mathbf{f}: \mathbf{S} \times \mathbf{I} \rightarrow \mathbf{O}$, where **f** is the operation, **S** is the type of the state of the FADT, **I** is the type of the input parameter, and **O** is the type of the result of the operation.
- A procedural style for the interfaces to the FADTs. For the state transition function above, an interface to the function would be $\mathbf{f}': \mathbf{I} \rightarrow \mathbf{O}$.

The following sections discuss the implementation of FADTs at three levels of abstraction: (1) the execution mechanism level; (2) the system software level; and (3) the PRM level. The requirement is that access to the state of the FADT should be serializable at each level, i.e. concurrent execution of multiple operations have the same effect as some serial execution.

4.1.1 Execution mechanism level

The MONSTR (Maximum of One Non-root STateholder per Rewrite) computational model ensures the consistency of the state of an FADT at the execution mechanism level. MONSTR is the base computational model supported by the hardware, and imposes some minor restrictions on the DACTL rulesets that can be allowed on Flagship. The MONSTR model is designed to permit state to be used in a clean and controlled way while eliminating opportunities for performing side-effects through other mechanisms in the model (Banach and Watson, 1989).

The basic execution mechanism defines a state-holder packet type with the following properties: (1) it cannot be copied, i.e. once a state-holder packet is created it is unique and is located on a single processing element (PE)-store pair; and (2) at most one state-holder can be involved in a rewrite (Watson, 1987*b*). These two properties, allied with the indivisibility of a rewrite, allow serialized update and interrogation of state-holder packets.

The root packet of a rewritable sub-graph (RSG) involving a state-holder is moved to the PE-store pair where the state-holder packet is located. The rest of the RSG is in turn copied to that PE-store pair and then the rewrite occurs. Because a single state-holder is involved in the rewrite only one PE is involved. This PE is the only one able to access the state-holder and, because a rewrite is indivisible, the state-holder can never be involved in two rewrites at the same time. Consequently, access to the state-holder is serializable.

FADTs implemented at the MONSTR level are constrained in two ways: (1) the state of an FADT must be representable as a single state-holder packet; and (2) all operations on an FADT must be the size and complexity of a single rewrite.

4.1.2 System software level

FADTs at the system software level, generally, have the following properties: (1) the operations are of the size of a Hope$^+$ function (implemented as a series of rewrites); and (2) the state is of arbitrary size and complexity. Consequently, the two restrictions

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

on an FADT at the execution mechanism level are lifted and a mechanism is required to ensure serialized access to a number of state-holder packets by a series of rewrites. This mechanism is built upon MONSTR.

The *guardian* mechanism is used to implement FADTs in the kernel of the system software. Guardians build a stream processing function from a state transition function, and then hide the stream from the interface seen by users (Broughton *et al.*, 1987). The concept was introduced by Dennis (1981), and is similar to that of *managers* (Arvind and Brock, 1984).

The guardian mechanism ensures that an operation application has sole access to the state value of the FADT for the duration of the application. The guardian mechanism accesses the current state value in the stateholder, allows the application of an operation to this value, updates the state in the stateholder with the new value and delivers the result of the operation to the caller (Mayes *et al.*, 1991).

The Flagship guardian consists of a state component and a set of state transition functions. Both the state and the state transition functions are inaccessible to the caller. Instead, the caller sees a set of interface functions that have no explicit reference to the state in their signatures. The interface to the state transition function is represented by a pseudo-function that has the state parameter stripped. The state manipulation is handled internally by the guardian mechanism. The pseudo-functions are termed *currencies*.

An FADT instance consists of a set of currencies, one currency for each interface function. Guardians were chosen because the guardian mechanism maps efficiently onto the basic execution mechanism. The currencies of a guardian are implemented using an Incomplete eXecutable APplication (IXAP) packet, an application without its full arity of arguments, and the application of a currency is implemented as an Incomplete Processable APplication (IPAP) packet, an application of an unknown function (Holdsworth, 1988).

Multiple entries into a guardian/manager must be serialized, consequently entry to the guardian/manager effectively acts as a *merge* operation (see Fig. 4). Guardians/managers are, however, significantly more powerful than merely adding a *merge* primitive to a language (as have many of the attempts to use functional languages to handle non-determinism), because they allow systems that dynamically determine the use of resources (Arvind and Brock, 1984).

4.1.3 Programming reference model (PRM) level

The architectural style of the Flagship system as viewed by language users and compilers is guided by the PRM (Thomson, 1987). The PRM is based on declarative programming using graph reduction as the computational mechanism. However, it is recognized that real applications need to represent entities that have changeable state. The PRM deals with this directly by adding the notions of *state* and *update of state* to declarative programming.

An *object* is the smallest entity that can have state. The action of changing the state is called an *update*. To accommodate the non-determinism introduced by *objects* and *updates*, the 'pure' graph reduction computational mechanism must be extended to

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

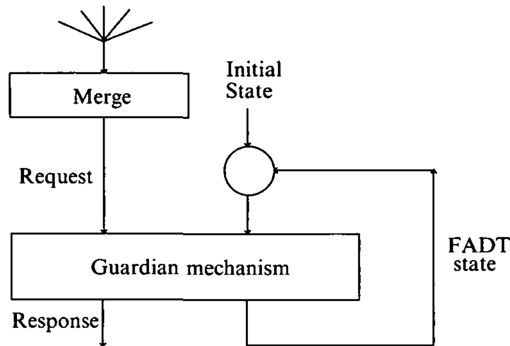

Fig. 4. The Guardian/Manager approach (Holdsworth, 1988).

provide: (1) control over the number of times a particular reduction is done; (2) control over the order in which reductions are performed; and (3) the possibility that distinct reductions of the same sub-graph will produce different results. Unconstrained use of the 'enhanced' mechanism will neutralize almost all the benefits gained from a declarative approach. Thus, a concept is needed to deal with objects and their updating in some more controlled way. This concept is the *atomic action* that collects a group of non-declarative actions into a single operation that is effectively atomic. This atomicity implies that an atomic action cannot be affected by partial completion of another atomic action (Thomson, 1987).

FADTs above the kernel of the system software, therefore, would be implemented using objects and atomic actions. Little work has been done at this level, but it has been shown how, as the guardian mechanism is built upon the atomicity provided by MONSTR, PRM objects and atomic actions can be constructed using guardians (Holdsworth, 1988).

4.2 The structure of the Flagship system software

The structure of the Flagship system software is hierarchical. The base software has been divided into four main layers (Leunig, 1988) (see Fig. 5):

1. **The process kernel:** the basic concepts provided by this layer are processes and interprocess communication, and store. In addition, the Kernel Object Store Manger is to support basic protection mechanisms and objects.
2. **The basic abstract machine:** the objective of this layer was to add the interfaces for driving devices and communication connections.
3. **The raw persistent object store abstract machine:** the objective of this layer was to provide the concept of persistent objects.
4. **The PRM abstract machine:** this layer was to be an implementation of the PRM.

Mayes *et al.* (1991) present the system software as a series of abstract machines each providing a more coarse-grained 'atomic' action, e.g. the *MONSTR machine* provides an atomic rewrite, the *guardian machine* provides an atomic function, and the *PRM machine* provides an atomic action.

As can be seen in Fig. 5, the layers of the system software have been further subdivided. In particular, the kernel has been divided into a number of resource

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

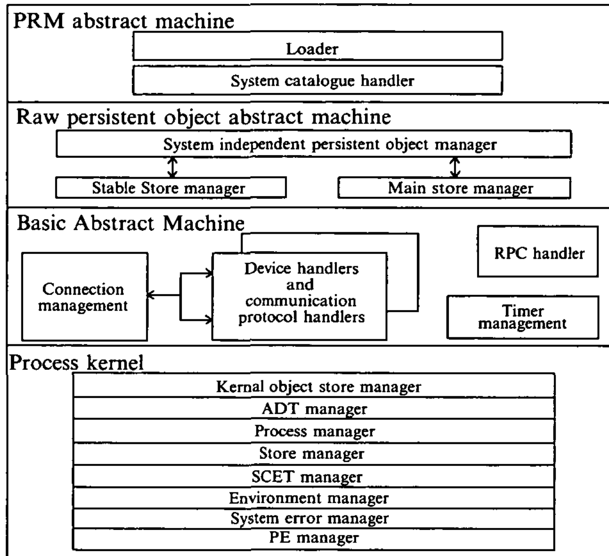

Fig. 5. The Flagship system software.

manager subsystems. An important consideration in the design of these subsystems has been their structuring into FADTs (it should be recognized that little work has been done above the kernel level).

An important consideration in a parallel machine is the global-to-local mappings that occur. Ideally, the vast majority of accesses to system facilities should be done locally. This principle has been emphasized in the design of the resource manager subsystems of the Flagship kernel. The subsystems, in general, are structured as a tuple (*global-resource-manager*, *set-of* (*local-resource-manager*)). The *global-resource-manager* is used when there is a need to co-ordinate kernel activity across two or more PEs in the management of its associated resource. The vast majority of requests to access resources by rewrites are handled by the *local-resource-manager* on the PE where the rewrite is being performed thus causing no network traffic. In most cases, there should be no need to make use of their associated global manager. These *local-resource-managers* manage their particular resource on the PE on which they are situated (Keane and Mayes, 1992).

The state-handling aspects of the system software have led to a different use of the Static Copying Environment Table (SCET) mechanism (see Section 3.2.1) by the process kernel. The distributed implementation of the SCET binding mechanism allows *nodal variants*, i.e. the same SCET index binds to different packets on different PEs (Keane, 1990*b*). When a rewrite requires access to resource manager interfaces a SCET index is planted at compile-time. When the rewrite calls the resource manager the nodal variant SCET ensures that the *local* resource manger is always used, thus reducing cross-system activity. Each of the interface functions of the local resource manager is addressed via the same index in each local SCET. However, each local SCET contains a *different* reference to its local resource manager. The difference being that the local interfaces of a resource manager will be associated with a different

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

FADT instance. The nodal variant SCET requires a mapping between each PE and the local packet representing the interface of the local resource manager. This mapping is used to place the appropriate instance reference in the appropriate local SCET. Consequently, the usual SCET copying from the Master SCET to the Local SCET is inhibited because of the presence of a local copy before any reference is made to that entry.

Protection/security in the software environment

The security system is based upon the PRM object model. A process receives its access rights from the *principal* object under whose auspices it is run. A principal object is the system representation of a user. Computations occur within a process and inherit the access rights of their owning process. Access is regulated in two ways, i.e. an access must be allowed by both:

- *Static* security controls the *type* of access permitted to each object (usually termed *discretionary* access controls). For any two objects a and b, there is a set of operations s that computations that derive their access rights from object a can perform on object b. When an operation o is requested on b, a check is made to see whether $o \in s$.
- *Dynamic* security is based on the 'B2' class (US Department of Defense, 1983). Access to objects is based on classification levels (usually termed *mandatory* access controls). A set of classifications is used to limit the flow of information from highly classified objects to objects of lower classification. Each object in the system has a *classification*, and each process has a *current classification* and an *upper limit classification*. A computation is restricted to reading from objects with a *classification* less than or equal to the *current classification* of the process at which it is rooted. If the accessed object's *classification* is higher than the *current classification* of the process, then the classification of the process is set to the object's classification, provided that the new *current classification* of the process will not exceed its *upper limit classification*. A computation can only write to objects that have a *classification* higher than the *current classification* of the process at which it is rooted. It is possible to override the classification check if the static access permissions include a *write down* operation.

It was felt that to provide the PRM security, the basic execution mechanism of the machine required protection mechanisms. During reduction, the current protection domain, i.e. the domain of entities that may be accessed by the computation, is established by the basic execution mechanism. The crossing of the boundary of a protection domain is controlled by 'capability packets' that depend on the secure implementation of pointer and store management. A capability (*cap*) pointer type is provided by the basic execution mechanism. If the root packet of a rewrite references a packet in a different protection domain, then that reference must be via a *cap* pointer. The presence of a *cap* tag in a pointer item of a packet transforms that item into a *capability* for the referenced packet.

This section is based on Holdsworth *et al.* (1989).

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

4.3 *Software environment development route*

The aim of the development route was to enable a rigorous approach to the design and implementation of the software environment to be adopted. The development route activity placed particular emphasis on the specification and design stages of the software lifecycle, and so was complementary to the work described in Section 2 on the programming environment.

The model-oriented specification language VDM (Jones, 1990) was chosen because it provides both functions and operations (that specify state changes). Flagship VDM (Boddy, 1988) is a hybrid that adds some of the properties of Hope$^+$, such as polymorphic data types, to a VDM framework. A specification of a Flagship ADT in Flagship VDM consists of data type definitions together with functions and operations on those data types. This specification is then transformed into Hope$^+$ to allow rapid prototyping. For this transformation to occur, non-executable parts of Flagship VDM, such as quantification over infinite sets and implicit functions, must be manually removed or modified to be executable.

4.4 *Status of software environment activity*

All the subsystems in the process kernel layer of the system software were specified in Flagship VDM and prototyped using Hope$^+$. A subset of this kernel has been demonstrated running on the prototype machine. A semantics of the parallel execution of the subsystems in terms of the possible effects on state using *rely and guarantee* conditions (Jones, 1983) is discussed by Keane and Hussak (1992). The development route was implemented using PS-algol (Atkinson *et al.*, 1983), and used in the development of the Process Kernel.

Because only a subset of the kernel was put onto the prototype, very little experimentation of running a number of programs concurrently (i.e. multiprogramming) on the prototype has been carried out. A process in the system software was intended to approximately correspond to a functional program.

A 32-processor Flagship machine has been simulated running four benchmark programs (*nfib. nqueens*, *matrix multiply* & *binary integration*), sequentially and concurrently (the four programs run at the same time). In comparison to running the programs sequentially, running them concurrently gives enhanced speedup (an improvement of approximately 25%), throughput (an improvement of approximately 25%), better processor utilization (an improvement of approximately 15%) and a reduction in network activity (a reduction of approximately 25%). The speedup, throughput and processor utilization improvements are due to there being more rewritable sub-graphs to keep the machine busy. The reduction in network activity appears to come about because the load balancing mechanism is more effective with larger loads. Because the machine load is better balanced and the processors are less idle, there is less need to export work and network traffic is reduced (Tan and Woods, 1990).

Various distributed algorithms suited to a Flagship-style architecture have been developed, these include deadlock detection (Hilditch and Thomson, 1989), relational

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

queries (Hilditch and Thomson, 1990) and termination detection (Ye and Keane, 1992).

5 Conclusions

This section summarizes the achievements of Flagship. Some conclusions about the approaches taken are also presented. The achievements are:

- The language Hope$^+$, based on Hope, has been designed and extensively used in the project. A number of other experimental languages that extend Hope$^+$ in various directions have been developed. A programming environment supporting these languages and providing many other facilities has been developed. This work has been highly innovative and has shown that many of the benefits claimed for functional languages are valid.
- A prototype 16 processor Flagship machine has been designed and built to support graph reduction. The performance of this machine has been studied. In addition, three prototype ALICE machines have been built, and performance studies carried out.
- A number of computational models for various declarative languages have been developed for this machine. These include three models for functional languages, models for object-oriented languages, and a model for Prolog.
- A software environment has been designed and the kernel of this environment has been formally specified and modelled. A subset of this kernel has been implemented on the prototype machine. A development route to assist in this work has been built.

By concentrating on an *entire* system, the higher layers were able to provide a certain amount of feedback at *design* time to the machine architecture and computational models. However, in some ways, the breadth of the project has been a weakness. For example, it can be argued that GRIP (Peyton Jones *et al.*, 1989), a less ambitious project than Flagship, has been more successful as an experimental parallel machine.

The software environment activity focused on aspects that are rarely considered in experimental declarative systems, e.g. security/protection mechanisms. However, its interface was highly complex (based on the view that a Flagship-like system would be commercially available in the early 1990s and would require a 'conventional' interface). This complexity was reflected in the design of the system software, and as a result only a subset of the kernel was actually put onto the prototype. Thus, the prototype machine did not have a usable interface which meant experimentation was time-consuming and error-prone. In retrospect, a very minimal interface would have sufficed for early experimentation, and an evolutionary approach could have been taken if and when a commercial version was required. It is difficult from this work to make any firm conclusions with regard to the usefulness of functional languages in this area.

Much of the specific work started in the project is continuing. This paper has covered most of this work to the middle of 1992. A follow-on Esprit II project, the *European Declarative System* (EDS), has developed some of the achievements of

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

Flagship (for a brief overview of EDS, see Keane, 1990*a*). The success of the Flagship machine in performing database queries and the importance of this application commercially has made this area the main focus of the EDS project. As the machine architecture will remain the same, the EDS machine and its successors may be seen as commercial derivatives of the Flagship machine.

Acknowledgements

This paper is an updated and abridged version of Keane (1990*a*). Thanks to all members of Flagship who contributed to the work described. Special thanks to Richard Banach, Graham Boddy, Margaret Derbyshire, Andrew Grant, Mark Greenberg, Sean Holdsworth, Walter Hussak, Steve Kellett, Steve Leunig, Ken Mayes, Billy Mitchell, Iain Robertson, Jin Sa, Martyn Spink, Salleh Syed-Mustaffa, Gary Tan, George Tsakogiannis, Brian Warboys, Ian Watson, Paul Watson and Xinfeng Ye for comments and/or discussions over the years. Thanks also to all members of the Centre for Novel Computing at Manchester, and to the anonymous referees whose comments have greatly improved the paper.

References

Almasi, G. S. and Gottlieb, A. (1989) *Highly Parallel Computing*. Benjamin/Cummings.

Arvind, X. and Brock, J. D. (1984) Resource Managers in Functional Programming. *J. Parallel and Distributed Computing*, **1** (1): 5–21.

Atkinson, M. P., Bailey, P., Cockshott, W. P., Chisholm, K. J. and Morrison, R. (1983) *The PS-algol Reference Manual*, Persistent Programming Research Report 4, University of Glasgow & University of St Andrews.

Banach, R., Sargeant, J., Watson, I., Watson, P. and Woods, V. (1988) The Flagship Project. In *Proc. IEE/BCS UK IT 88*, 242–245.

Banach, R. and Watson, P. (1989) Dealing with state on Flagship: The MONSTR computational model. In C. Jesshope and K. D. Reinartz (editors), *CONPAR 88*, Cambridge University Press, 595–604.

Boddy, G. S. (1988) The use of VDM within the Alvey Flagship project. In R. Bloomfield, L. Marshall and R. Jones (editors), *VDM – The Way Ahead*, *LNCS*, vol. 328, Springer-Verlag, 153–160.

Broughton, P., Thomson, C. M., Leunig, S. R. and Prior, S. (1987) Designing system software for parallel declarative systems. *ICL Technical J.*, **5** (3): 541–554.

Burstall, R. M., MacQueen, D. B. and Sanella, D. T. (1980) *HOPE: An experimental applicative language*. Technical Report CSR-62-80, Department of Computing Science, University of Edinburgh.

Cheese, A. B. (1990) A distributed model of computation for combinatory code. In H. S. M. Zedan (editor), *Distributed Computer Systems*, Butterworths, 161–171.

Darlington, J. and Reeve, M. (1981) ALICE: A multiprocessor reduction machine for the parallel evaluation of applicative languages. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, 65–75.

Darlington, J. (1987) Software development using functional programming languages. *ICL Technical J.*, **5** (3): 492–508.

Darlington, J. and While, L. (1987) Controlling the behaviour of functional language systems. In G. Kahn (editor), *Functional Programming Languages and Computer Architectures*, *LNCS*, vol. 274, Springer-Verlag, 278–300.

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

Darlington, J. and Guo, Y. (1989) Narrowing and unification in functional programming – an evaluation mechanism for absolute set abstraction. In N. Dershowitz (editor), *Rewriting Techniques and Applications*, *LNCS*, vol. 355, Springer-Verlag, 92–108.

Darlington, J., Khoshnevisan, K., McLoughlin, L., Perry, N., Pull, H., Sephton, K. and While, L. (1989*a*) An introduction to the Flagship programming environment. In C. R. Jesshope and K. D. Reinartz (editors), *CONPAR 88*, Cambridge University Press, 108–115.

Darlington, J., Harrison, P., Khoshnevisan, H., McLoughlin, L., Perry, N., Pull, H., Reeve, M., Sephton, K., While, L. and Wright, S. (1989*b*) A functional programming environment supporting execution, partial execution and transformation. In E. Odijik, M. Rem and J.-C. Syre (editors), *PARLE '89 Parallel Architectures and Languages Europe volume 1*, *LNCS*, vol. 365, Springer-Verlag, 286–305.

Darlington, J., Field, A. J., Harrison, P. G., Harper, D., Jouret, G. K., Kelly, P. J., Sephton, K. and Sharp, D. W. (1991) Structured parallel functional programming. In H. Glaser and P. Hartel (editors), *Proc. Workshop on Parallel Implementation of Functional Languages*, Technical Report CSTR 91-07, Department of Electronics and Computer Science, University of Southampton, 31–51.

Dennis, J. B. (1981) *Data Should Not Change: A Model for a Computer System*. MIT Laboratory for Computer Science, Cambridge, MA.

Derbyshire, M. H. (1990) Mark scan garbage collection on a distributed architecture. *J. Lisp and Symbolic Computation*, **3**: 135–170.

Field, A. J. and Harrison, P. G. (1988) *Functional Programming*. Addison-Wesley.

Glauert, J. R. W., Hammond, K., Kennaway, J. R. and Papodopoulos, G. A. (1989) Using DACTL to implement declarative languages. In C. R. Jesshope and K. D. Reinartz (editors), *CONPAR 88*, Cambridge University Press, 116–124.

Greenberg, M. I. (1989) *An Investigation into Architectures for a Parallel Packet Reduction Machine*. PhD Thesis, University of Manchester.

Greenberg, M. I. & Woods, V. (1990) FLAGSHIP – a parallel reduction machine for declarative programming. *Computing & Control Engineering J.*, **1** (2): 81–86.

Harrison, P. G. (1987) Functional Inversion. In *Proc. Workshop on Parallel Evaluation and Mixed Computation*, North-Holland.

Hilditch, A. S. & Thomson, C. M. (1989) *Distributed Deadlock Detection: Algorithms and Proofs*. Technical Report UMCS-89-6-1, Department of Computer Science, University of Manchester.

Hilditch, A. S. and Thomson, C. M. (1990) *Distributed Relational Queries: Structures for Locking and Access*. Technical Report UMCS-90-8-1, Department of Computer Science, University of Manchester.

Hoare, C. A. R. (1985) *Communicating Sequential Processes*. Prentice-Hall.

Holdsworth, S. (1988) *The Implementation of Layers of Abstraction of Serialisability in a Parallel Packet Rewrite Machine*. Transfer Report, University of Manchester.

Holdsworth, S., Keane, J. A. and Mayes, K. R. (1989) Aspects of protection on the Flagship machine. *ICL Technical J.*, **6** (4): 757–777.

Hudak, P., Peyton Jones, S. and Wadler, P. (editors) (1992) *Report on the Programming Language Haskell, Version 1.2*.

Hughes, R. M. J. 1982. *Graph Reduction with Super-Combinators*. Technical Monograph PRG-28, Oxford University Computing Laboratory.

Jones, C. B. (1983) Tentative steps towards a development method for interfering programs. *ACM TOPLAS*, **5** (4): 596–619.

Jones, C. B. (1990) *Systematic Software Development Using VDM*. 2nd Ed. Prentice Hall.

Jones, S. B. and Sinclair, A. F. (1991) On input and output in functional languages. In J-P. Banâtre, S. B. Jones and D. Le Métayer (editors), *Prospects for Functional Programming in Software Engineering*, Esprit Research Reports Project 302 Volume 1, Springer-Verlag, 139–171.

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

Keane, J. A. (1989) *Aspects of Binding in a Declarative System.* MSc Thesis, University of Manchester.

Keane, J. A. (1990*a*). *The Flagship Declarative System.* Technical Report UMCS-90-2-1, Department of Computer Science, University of Manchester.

Keane, J. A. (1990*b*). *Distributed Binding Mechanisms in the Flagship System.* Technical Report UMCS-90-8-2, Department of Computer Science, University of Manchester.

Keane, J. A. and Hussak, W. (1992) The use of formal methods in parallel operating systems. In *Proc. CompSAC 92*, IEEE Press, 245–250.

Keane, J. A. and Mayes, K. R. (1992) Resource management on a packet-based parallel graph reduction machine. In *Proc. CONPAR 92 – VAPP V, LNCS, vol.* 634, Springer-Verlag, 417–422.

Keller, R. M. (1987) Rediflow architecture prospectus. In S. S. Thakkar (editor), *Selected Reprints on Dataflow and Reduction Architectures*, IEEE Press, 366–394.

Kennaway, J. R. and Sleep, M. R. (1984) The 'Language First' approach. In F. B. Chambers, D. A. Duce and G. P. Jones (editors), *Distributed Computing.* APIC Studies in Data Processing, 11–124.

Kewley, J. M. and Glynn, K. (1990) Evaluation annotations for Hope$^+$. In K. Davis and R. J. M. Hughes (editors), *Functional Programming, Glasgow, 1989*, Springer-Verlag, 329–337.

Khoshnevisan, H. (1988) *Efficient Memo-Table Management Strategies.* Department of Computing, Imperial College London.

Khoshnevisan, H. and Sephton, K. (1989) InvX: An automatic function inversion. In N. Dershowitz (editor), *Rewriting Techniques and Applications, LNCS*, vol. 355, Springer-Verlag, 564–568.

Kirkham, C. (1990) The MANCHESTER DATAFLOW project. In T. J. Fountain and M. J. Shute (editors). *Multiprocessor Computer Architecture.* North-Holland, 141–153.

Leunig, S. R. (1987) *Abstract Data Types in the Flagship System Software.* Flagship Document FLAG/DD/303.xxx, ICL.

Leunig, S. R. (1988) *Design Description of the System Software.* Flagship Document FLAG/DD/3SD.016, ICL.

Liu, F. (1990) Absolute set abstraction and its evaluation mechanism. In M. Plasmeijer (editor), *Implementations of Functional Languages on Parallel Architectures.* Technical Report No. 90-16, University of Nijmegen, 321–352.

McLoughlin, L. and Hayes, E. S. (1990) Imperative effects from a pure functional language. In K. Davis and R. J. M. Hughes (editors), *Functional Programming, Glasgow, 1989*, Springer-Verlag, 157–169.

Mago, G. and Stanat, D. F. (1989) The FFP machine. In V. M. Milutinovic (editor), *High Level Language Computer Architecture.* Computer Science Press, 430–468.

Mayes, K. R., Keane, J. A. and Holdsworth, S. (1991) *The Structure of the Flagship System Software.* Technical Report UMCS-91-11-1, Department of Computer Science, University of Manchester.

Michie, D. (1968) 'Memo' functions and machine learning. *Nature*, **218**: 19–22.

Perry, N. (1987*a*) *HOPE*$^+$. IC/FPR/LANG/2.5.1/7, Department of Computing, Imperial College.

Perry, N. (1987*b*) *HOPE*$^+$*C.* IC/FPR/LANG/2.5.1/21, Department of Computing, Imperial College.

Peyton Jones, S. L. (1987) *The Implementation of Functional Programming Languages.* Prentice-Hall.

Peyton Jones, S. L., Clack, C. and Salkild, J. (1989) High-performance parallel graph reduction. In E. Odijik, M. Rem and J.-C. Syre (editors), *PARLE '89 Parallel Architectures and Languages Europe Vol. 1, LNCS*, vol. 365, Springer-Verlag, 193–206.

Reeve, M. and Wright, S. (1990) The experimental ALICE machine. In T. J. Fountain and M. J. Shute (editors), *Multiprocessor Computer Architecture.* North-Holland, 39–56.

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press

Robertson, I. B. (1990) Hope+ on Flagship. In K. Davis and R. J. M. Hughes (editors), *Functional Programming, Glasgow, 1989*, Springer-Verlag, 296–307.

Sargeant, J. (1987) *Load Balancing, Locality and Parallelism Control in Fine-Grain Parallel Machines*. Technical Report UMCS-86-11-5, Department of Computer Science, University of Manchester.

Sargeant, J. and Watson, I. (1991) Some experiments in controlling the dynamic behaviour of parallel functional programs. In H. Glaser and P. Hartel (editors), *Proc. Workshop on Parallel Implementation of Functional Languages*. Technical Report CSTR 91-07, Department of Electronics and Computer Science, University of Southampton, 103–121.

Sharifi, M. (1991) *An Investigation into the Attributes of General Purpose Parallel Programming Models*. PhD Thesis, University of Manchester.

Sleep, M. R. and Kennaway, J. R. (1984) The Zero Assignment Parallel Processor (ZAPP) project. In D. A. Duce (editor), *Distributed Computing Systems Programme*, Peter Peregrinus, 250–269.

Syed-Mustaffa, S. M. S. (1992) A model for OR-Parallel Prolog execution using graph reduction. In D. R. Brough (editor), *Logic Programming – New Frontiers*. Intellect Books, 227–252.

Tan, G. and Woods, V. (1990) Load balancing and multiprogramming in the flagship parallel reduction machine. In *Proc. PARBASE-90*, IEEE Press, 560.

Thomson, C. M. (1987) *PRM Definition*. Flagship Document FLAG/DD/101.001, ICL.

Townsend, P. (1987) Flagship hardware and implementation. *ICL Technical J.*, **5** (3): 575–594.

Trinder, P. W. (1989) *A Functional Database*. DPhil Thesis, Oxford University.

Turner, D. A. (1979) A new implementation technique for applicative languages. *Software Practice and Experience*, **9** (1): 31–49.

U.S. Department of Defense Security Center, (1983) *Trusted Computer System Evaluation Criteria*. CSC-STD-001-83.

Watson, P. (1986) *The Parallel Reduction of Lambda Calculus Expressions*. PhD Thesis, University of Manchester.

Watson, I., Watson, P. & Woods, V. (1986) Parallel data-driven graph reduction. In J. V. Woods (editor), *Fifth Generation Computer Architectures*. North-Holland, 203–220.

Watson, I. (1987*a*) *Environments and Contexts*. Flagship Document FS/MU/IW/013-87, University of Manchester.

Watson, P. (1987*b*) *The Flagship Basic Execution Mechanism*. Flagship Document FS/MU/PW/018-87, University of Manchester.

Watson, P. (1987*c*) *A Supercombinator Model of Computation*. Flagship Document FS/MU/PW/021-87, University of Manchester.

Watson, P. and Watson, I. (1987) An efficient garbage collection scheme for parallel computer architectures. In J. W. de Bakker, A. J. Nijman and P. C. Treleaven (editors), *PARLE Volume 2, LNCS*, vol. 259. Springer-Verlag, 432–443.

Watson, I., Sargeant, J., Watson, P. and Woods, V. (1989) The Flagship parallel machine. In C. R. Jesshope and K. D. Reinartz (editors), *CONPAR 88*, Cambridge University Press, 125–133.

Watson, P. (1990) The FLAGSHIP parallel machine. In T. J. Fountain and M. J. Shute (editors), *Multiprocessor Computer Architecture*. North-Holland, 57–81.

Watson, P. and Watson, I. (1990) Evaluating declarative languages on a parallel graph reduction machine. In G. David, R. T. Boute and B. D. Shriver (editors), *Declarative System*. North-Holland, 165–188.

Ye, X., Warboys, B. C. and Keane, J. A. (1990) Specification and verification of a distributed recovery algorithm for functional languages. In *Proc. 20th Int. Symp. on Fault-Tolerant Computing – FTCS 20*, IEEE Press, 307–314.

Ye, X. F. and Keane, J. A. (1992) Token scheme: An algorithm for distributed termination detection and its proof of correctness. In J. van Leeuwen (editor), *Information Processing 92 – Volume 1: Algorithms, Software, Architecture*. North-Holland, 357–364.

https://doi.org/10.1017/S0956796800000927 Published online by Cambridge University Press