STUDY ON A REFLECTIVE ARCHITECTURE TO PROVIDE EFFICIENT DYNAMIC RESOURCE MANAGEMENT FOR HIGHLY-PARALLEL OBJECT-ORIENTED APPLICATIONS

高並列オブジェクト指向アプリケーションのための効率のよい動的資源管理方式を提供する自己反映アーキテクチャの研究

by
Hidehiko Masuhara
増原 英彦

Master Thesis
修士論文

Submitted to

the Graduate School of
the University of Tokyo
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Information Science

February 7, 1994

ABSTRACT

Recent progress in implementations of object-oriented concurrent programming languages on highly-parallel processors makes it feasible to construct large-scale parallel applications having complicated structures. Such applications can not exhibit good performance without dynamic resource management (e.g., load-balancing and object scheduling) tailored to the characteristics of the applications and/or machine architectures. Since dynamic resource management systems are usually intertwined with the language implementation, modification/extension of the management systems requires complicated programming in the low-level language. Reflective systems can provide abstractions to modify/extend the implementation-level facilities within the application-level language. This study proposes a reflective architecture of an object-oriented concurrent language for highly-parallel processors to provide resource management systems for parallel applications. To make our architecture practical, much attention is payed on balancing the trade-offs between extensibility and efficiency; we examine the requirements for realistic applications by developing resource management systems for search problems and N-body simulation, and design the architecture based on the requirements. The architecture is evaluated by experiments through a prototype system running on a parallel computer.

論 文 要 旨

高並列計算機上の並列オブジェクト指向言語の実現方式の研究の発展により、大規模で複雑な構造を持つ並列アプリケーションの構築が可能となってきた。このようなアプリケーションを効率よく実行するためには、アプリケーションの性質や計算機アーキテクチャに応じて、適切な負荷分散やスケジューリングなどの動的資源管理を提供する必要がある。従来、これらの資源管理システムは、言語の実行時システムなどに組み込まれて設計されているため、変更や機能拡張のためには、低レベル言語による煩雑なプログラミングが必要であった。本研究では、自己反映計算が、そのような言語実装に関わる機能を、アプリケーションレベルの言語で容易に変更・拡張するためのインタフェースを提供する点に着目し、高並列計算機向けの並列オブジェクト指向言語の自己反映アーキテクチャを提案する。特に、自己反映計算の持つ拡張性とオーバヘッドのバランスをとるために、探索問題や N 体シミュレーションなどの実際的なアプリケーション向けのスケジューリング・マイグレーションをおこなうシステムの記述を通して、アーキテクチャの設計をおこなう。さらに、並列計算機上のプロトタイプシステムでの実験によって、提案したアーキテクチャの評価をおこなう。

Acknowledgements

I would like to express my deep appreciation to Professor Akinori Yonezawa, who gave me numerous comments and encouragement. Also, deep appreciation goes to Satoshi Matsuoka, who gave helpful and precious comments and encouragement. I am grateful to Naoki Kobayashi for his encouragement. I would like to thank Kenichi Asai and Jeff McAffer for numerous and fruitful discussions on reflective architectures. My appreciation goes to Kenjiro Taura, and Masahiro Yasugi. Daily discussions on language implementations and concurrent applications with them gave me important foresight to practical systems. Discussions with Masaaki Nagatuka, Naohito Omori, Kazuhiro Konno, and Tomio Kamada were also helpful. I would like thank Takuo Watanabe, Yuuji Ichisugi, Shigeru Chiba, Takashi Tomokiyo, Ken Wakita, and the members of OOPS! group at Keio University. I also thank the members of our laboratory, who always encourage me. Finally, special thanks to Kazuhiro Konno, who always keeps our computers in the best condition. Without his endeavor, my research progress should have been far from satisfaction.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, advances of implementation techniques of object-oriented concurrent languages on distributed memory multicomputers[46] make it feasible to execute large-scale and complicated concurrent applications. To make such applications exhibit as much performance as the language provides, good dynamic resource management (DRM), such as load balancing and scheduling, is necessary. Unfortunately, there is no ultimate DRM in which any application could exhibit its best performance. Hence each application tends to have a tailored DRM system, in which assumptions to application's behavior (e.g., communication pattern and data structure) and run-time information from the application are extensively used. The problem is how to easily incorporate tailored DRM systems for various kinds of applications. Approaches that have been tried so far are embedding DRM related operations into an application program, or customizing a run-time system of a language—so called hacking. In the former approach, since operations for original application and for DRM are intertwined into one program, it is difficult to reason about such a program and to re-use it. Another problem of the former approach is efficiency. Since a language provides limited flexibility, it is often that a DRM requires facilities beyond the language provides. In such a case, we should simulate those facilities in an application program, which forces the application program written awkwardly, and sometimes poses unacceptable overheads. On the other hand, the latter approach is far from convenient. Without deep knowledge of the run-time system implementation, a modification to the run-time system easily leads to a disaster. Even if some customization protocols, which promise safe modification, are provided, following problems remain unsolved. (1) Since interface between the run-time system and applications program is fixed, it is hard to access application-level information from a DRM system. (2) A customization to a part of the run-time system usually alters the entire system; i.e., it is hard to localize the customization. (3) Since descriptions of DRM systems should be given in a low-level language same to the one that implements the run-time system, a complicated DRM system—which tends to be a concurrent computing itself—is hard to be programmed. To summarize, we need a language extension mechanism in which an extension is clearly separated from application programs,

while it can access application-level information, and the extension can be localized in an arbitrary scope. In addition, unit of customization should be fine enough so as to meet with requirements from various kinds of DRM, at the same time, such a customization can enjoy high-level language facilities including concurrency supports.

This study proposes a reflective architecture as a solution. Computational reflection is a process that accesses its own computational process[34, 21, 22]. In a reflective language, its computational process, which executes application programs at a *base-level*, is provided as a *meta-level* in an appropriate abstraction, and ways to modify/extend the meta-level within the language are available. The extensibility of reflection has two aspects in terms of target of extensions. The one is the *run-time system extensibility,* which is to extend run-time modules of a language such as a scheduler, and the other is the *language facility extensibility,* which is to extend/modify interpretation of a program, like in sequential reflective languages. It has been noted that the run-time extensibility makes it easy to provide tailored DRM systems for parallel/distributed environment in a clear and separated way[47]. In addition, the language facility extensibility is also useful because it can provide ways to access application-level information from DRM systems, and to localize the scope of modification. Moreover, better compilation could be done to for DRM systems in reflective approaches, compared to application-embedded ones. This is because compilers can gain better knowledge of modification from the reflective approach, which can describe modifications directly *into* a language, while application-embedded one does just *on* a language. So far, a number of reflective systems have been designed mainly for distributed systems[40, 41, 24, 15, 26, 43, 8, 38, 25], and a few DRM systems actually constructed at the meta-level of reflective systems are reported[23, 14, 27]. However, to the best of our knowledge, there are no reflective systems (except for ongoing one[38]) which is aiming to provide DRM systems for highly concurrent applications.

The proposed reflective architecture originates in the Hybrid Group Reflective Architecture (HGA)[24]. HGA is a general framework to provide control of shared computational resources in parallel/distributed applications. However, as a way to develop practical DRM systems of highly concurrent applications, provided abstractions are too high. Since it lacks the notion of a node (abstraction of a processor element), management relevant to distributed memory multiprocessors is hard to be described; and the meta-level design somewhat conflicts with a practical implementation. Our proposing architecture, called the *Distributed Memory Reflective Architecture*, is designed so as to overcome the above-mentioned problems. Its key features are as follows: (1) Notion of a node is explicitly available at the meta-level, while it is transparent to the base-level programming. (2) The meta-level design is based on the requirements from realistic DRM systems of concurrent applications so that they can be an appropriate abstraction of the run-time system of the language. (3) *Chain of executors*, which is a set of meta-circular interpreters (each of which is basically same to the 'evaluator' in HGA) giving operational interpretations of base-level programs

in delegation style, realizes language facility extensions. The language facility can be dynamically altered by replacing an executor in a chain at run-time; and the scope of the extended facility can be localized to a specific object or a group of objects. (4) Interface to system level information is defined to make good use of low-level information at DRM systems.

Not only extensibility, but also efficiency is given high consideration in the architecture's design. Based on the requirements from realistic DRM systems, the meta-level design carefully avoids providing unnecessary abstractions of the run-time system, which tend to pose unacceptable overheads by interfering with the low-level optimizations. Chains of executors is a good abstraction to facilitate language extensions whose facilities are dynamically (but not so frequently) changed. Our proposed scheme, called the *Dynamic Code Replacement,* is an efficient way to implement the chains of executors, provided a sophisticated partial evaluation technique that can collapse a fixed set of meta-level definitions.

The effectiveness of the architecture is verified by describing DRM systems for several non-trivial concurrent applications. Such DRM systems include load balancing, object allocation, and object scheduling. It is also tested by experiments; a simple application-specific DRM system is constructed on a prototype system running on Fujitsu AP1000, which is a distributed memory multicomputer. It exhibits performance improvement for a search problem, compared to a naive DRM system.

The main contributions of this study are as follows:

- A new reflective architecture, called the *Distributed Memory Reflective Architecture*, which purposes to easily provide DRM systems for highly concurrent applications, is designed.
- Design of the meta-level, which is an abstraction of the run-time systems relevant to DRM, is presented.
- A scheme for language facility extension, called *chains of executors*, is proposed. This scheme provides localized and dynamic extensions described in an operational way.
- An implementation scheme for language facility extension, called the *Dynamic Code Replacement*, is proposed. With this scheme, a base-level program can be executed with fully compiled codes, provided a sophisticated partial evaluation technique.

The rest of the thesis is organized as follows. In Chapter 2, ways to provide application specific DRM systems are examined, and problems in terms of the re-usability and the efficiency are pointed out. Next how reflection contributes to solve these problems are explained. Chapter 3 is devoted to the proposed reflective architecture. Firstly, the design requirements of reflective architectures from realistic DRM examples are examined. Next the Distributed Memory Reflective Architecture is explained. This chapter also contains issues on efficient implementation. In Chapter 4, evaluation of the architecture through the descriptions of DRM systems is presented, and an

experiment with a DRM system on a prototype system running on a multicomputer is shown. Finally, Chapter 6 concludes the thesis, giving future directions to the architecture design, the efficient implementation, and the DRM systems.

Chapter 2

Reflection and Highly Concurrent Object-Oriented Applications

In this chapter, we first explain the necessity of a language extension mechanism that supports construction of efficient and tailored dynamic resource management (DRM) systems. We then explain what reflection is, and that the reflective system is a suitable language extension mechanism, along with its advantages to the other (non-reflective) approaches.

2.1 Why Dynamic Resource Management?

Many highly concurrent applications have dynamic nature. In other words, their data structure and computational pattern are determined at the run-time. To execute such applications efficiently on multicomputers, the system must do appropriate dynamic resource management, such as load balancing, object/data allocation, and scheduling. In fact, there are several studies on high performance applications that have dynamic nature, and such studies mention dynamic resource management to greater or lesser extent[33, 11]. Many DRM algorithms and also mechanisms are proposed for multicomputer applications[42, 20, 28].

The algorithms used in dynamic resource management for such applications are tend to be application specific. An algorithm often assumes on computational patterns of a target application, such as number of concurrent objects (or tasks) generated at run-time. Moreover, many algorithms uses run-time information that is available at the application-level. For example, some dynamic object-allocation system uses the application-level information for predicting the amount of communications between objects[33]. An application specific DRM algorithm is tend to be embedded into the application programs, to have better access the application level knowledge.

While such a specialization shows a dramatical improvement in performance, there are the following problems. (1) The use of application level information in a DRM system degrades the re-usability of the DRM system's code, even to the same kind of applications. (2) The mixture of codes for the resource management and for the original application prevents both codes from

being modified easily.

Another feature of DRM systems is the concurrency. Since they manage resources for distributed memory computing, cooperation among node processors (e.g., gathering load information) can be understood as a concurrent computing. Such concurrency makes difficult to provide DRM systems using complicated algorithms.

To summarize, the description of DRM system should be: (1) separated from the ones for the original application as much as possible (at the same time, the system should be able to use application-level knowledge), (2) easily customized for various kinds of target architectures and low-level optimizations, and (3) easily written even if the management involves concurrent computing.

2.2 Extensibility that Reflection Provides

Computational reflection, or *reflection* in short, is a computational process that accesses its own computation process[34, 21, 22, 17]. So far, many reflective languages—programming languages that supports reflection—are proposed to provide clear and formalized extensibility to programming languages.

A reflective language has data which abstracts its computation process. A modification to the data results in a change to the actual computation process, and vice versa. Such data is called *causally-connected self representation* (CCSR), or *meta-level representation*. We emphasize that CCSR should be an *appropriate abstraction* from the viewpoint of extensions. For example, the implementation of a language itself can be regarded as a CCSR when an access method is provided. However, the implementation is too low abstraction for most kind of extensions. Therefore, most reflective languages uses a meta-circular definition of the language as a CCSR, and provides ways to modify/extend the definition.

Extensibility of a reflective language can be classified two groups. One is *language facility extension,* in which a new language facility can be used as if it is built-in, or the interpretation of an operation of the language can be changed. For example, [39, 10] shows that `call-with-current-continuation` of Scheme can be used in the base-level programs via the meta-level programming. This kind of extensions is usually accomplished by modifying an interpreter running at the meta-level. The other is *run-time system extension,* in which the functionalities of a run-time system of a language[1] can be modified or added in order to improve the system in performance. For example, an application specific scheduling that is implemented by reflection is shown in [23]. To provide this kind of extensions, a reflective system has CCSRs in which the functionality of a run-time system is implemented by several modules, and allows user to replace the modules by user-defined ones written for his own purpose. In object-oriented reflective languages,

[1] Not only programming languages, but also various kinds of systems have the run-time extensibility. Reflective window system[29] and reflective operating system[44] are the examples.

such modules are provided as objects so that extensions can be achieved in more encapsulated and modular ways.

2.3 Reflection for Dynamic Resource Management Systems

In this study, an object-oriented concurrent reflective architecture to provide various DRM systems for highly concurrent applications is proposed. This is because the extensibility of reflection has advantages to do this:

- The run-time system extensibility makes it easier to construct various DRM systems. In other words, a reflective system provides better abstraction to its run-time systems and allows user to customize it in high-level language, compared to a language whose run-time system implementation is opened. Moreover, the ability to customize the run-time system in an object-oriented concurrent language helps to program DRM system on distributed memory multicomputers that have concurrent and dynamic nature.

- Meta-level encapsulation makes both application programs and DRM systems more re-usable. In a reflective language, application specific DRM systems can be separately described, while such DRM systems have been embedded into application programs without reflection. As a matter of result, descriptions of both applications and DRM systems become much independent from each other.

- The language facility extensibility gives DRM systems more affinity to application programs, while preserving the separated descriptions. The language facility extension enables to obtain application level information, and to selectively use user-customized run-time system, with minimal modifications to application programs.

- Since the extensions are straightforwardly given to the language facilities, they can be regarded as extensions to the compiler. Provided sophisticated compilation techniques, application programs with such extended language facility could be optimized.

Although reflection has ability to provide good resource management systems, the ability largely depends on its design of reflective architecture; i.e., what abstraction is provided by reflection. Design issues of reflective architectures is discussed in next chapter.

Chapter 3

Distributed Memory Reflective Architecture: the Reflective Architecture for Highly Concurrent Computation

This chapter presents a reflective architecture, called the Distributed Memory Reflective Architecture. Before explaining the architecture itself and its detail, Section 3.1 examines the requirements to reflective architectures with respect to our purpose—providing various dynamic resource management (DRM) systems for highly concurrent applications. Next the architecture is presented in Section 3.2, and examples of simple reflective programming are shown in Section 3.3. Finally, Section 3.4 discusses how to implement the proposed architecture efficiently.

3.1 Requirements from Dynamic Resource Management Systems

In a reflective system, the design of the meta-level representation, or *abstraction* of the language, determines how clearly user customization can be described. For example, consider describing a user customized scheduling that gives specific concurrent objects high priority. If the level of abstraction is too high—for example, each base-level object runs of its own accord even in the meta-level's view, such scheduling is achieved via awkward programming such as explicit token passing at the meta-level. On the other hand, if the provided abstraction is too low—for example, activation frames are the meta-level representation, the user should get involved in low-level programming to achieve such scheduling. The latter case may not be a great trouble if a sophisticated abstraction, or a library, is provided onto the low-level representation at the same time. However, such unnecessary low-level abstraction probably poses overheads because it often conflicts with the low-level optimizations. This observation leads to the following design principle: (1) the meta-level should be appropriately designed with respect to the user's customization; and (2) no unnecessary low-levels should not be exposed.

This section examines realistic DRM systems for highly concurrent applications, so that our

```
[class fibonacci ()
 (script
  (=> n @ reply-to
      (cond ((< n 2) !n) ; "!n" returns the value of n.
            (t
             ;; each of the following two expressions creates a sub-object,
             ;; and sends an asynchronous message.
             [[new 'fibonacci] <= (- n 1) @ Me]
             [[new 'fibonacci] <= (- n 2) @ Me]
             ;; it waits return values from sub-objects.
             ;; returned values are bound to answer1 and answer2.
             (wait-for
              (=> answer1
               (wait-for
                (=> answer2
                 ;; the sum of returned values are sent to the specified object as an answer.
                 [reply-to <= (+ answer1 answer2)])))))))]]
```

Figure 3.1: Original Fibonacci Program

architecture provides appropriate abstraction with respect to the DRM systems.

3.1.1 Locality Control for Fibonacci

Firstly, we examine a simple example: the *locality control* for Fibonacci number computation. As is shown in Figure 3.1, when a Fibonacci object is requested to compute $fib(n)$ for $n \geq 2$, two sub-objects are created for $fib(n-1)$ and $fib(n-2)$.

In this program, parallelism is gained by creating a sub-object at a remote node. Since communications with remote node much slower than the node-local operations, creation at remote node does not always speeds up the computation. Hence it is sufficient to get the speed-up by only letting objects at shallow levels of the tree do the remote creation. We call this scheme the *locality control* for Fibonacci. (Figure 3.2)

Let us look how the locality control is accomplished in a non-reflective language. We assume that the language offers a way to cntrol in which node a new object is created—for example, `[:new` *class* `{:at :remote}]`. A reasonable solution would become a program shown in Figure 3.3; a condition (b) that checks whether it is at the deep level in the tree is added, and one of the sub-object creation forms (c) has the remote-creation annotation.

However, this solution has the following problems:

Mixture of DRM codes and original codes. In Figure 3.3, the condition for original computation (a) and for the locality control (b) appear at the same level. Thus a change to either original program or locality control policy could be harmful to the other.

For example, consider another locality control policy which determines the target nodes of object creation by the object's depth in the tree, not by the argument `n`. To implement this, an additional argument, which indicates the depth in the tree, should be passed along with

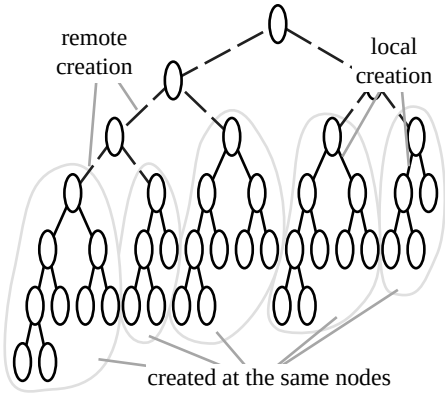

Figure 3.2: Locality Control for Fibonacci

the message, and many modifications should be done on the program.

Efficiency. In the program in Figure 3.3, once the condition (b) holdes for an object, the condition also holds for its sub-objects. Hence this condition is redundantly checked. Although the overheads for condition checks is not large in this case, it could become considerable overheads when the complicated condition is employed.

This problem can be avoided by giving two class definitions—the one checks the condition (`fib-R`) and the other does not (`fib-L`)—as is shown in Figure 3.4. However, this makes the problem 'mixture of DRM codes and original codes' worse; two classes are defined for one algorithm.

Incompatibility with other DRM. When this program is integrated into a large-scale application, and is concurrently executed with other modules, the locality control should obey higher level decisions. For example, when global load balance is good, remote creation is not needed. Hence the object creation form (c) in Figure 3.3 should designate "if higher level decision allows, this sub-object shall be created at a remote node," rather than designating the every-time remote creation.

To summarize, the requirements to the reflective systems are as follows:

- A mechanism to provide different interpretation to the same expression can be specified. (In this case, the target node for an object creation form.)
- User specified interpretation can be hierarchically defined. Some mechanism to override lower-level decisions by the higher-level ones could be useful.

```
[class fibonacci ()
  (script
   (=> n @ reply-to
       (cond ((< n 2)(a) !n)
             ((< n *threshold*)(b)
              [[new 'fibonacci {:at :local}] <= (- n 1)]
              [[new 'fibonacci {:at :local}] <= (- n 2)]
              (wait-for
               (=> answer1
                (wait-for
                 (=> answer2
                  [reply-to <= (+ answer1 answer2)])))
             (t
              [[new 'fibonacci {:at :local}]  <= (- n 1)]
              [[new 'fibonacci {:at :remote}](c) <= (- n 2)]
              (wait-for
               (=> answer1
                (wait-for
                 (=> answer2
                  [reply-to <= (+ answer1 answer2)])))))))]
```

Figure 3.3: Locality Control without Reflection

```
[class fib-R ()  ;; Fibonacci object for large n.
  (script
   (=> n @ reply-to
       (cond ((< n 2) !n)
             ((< n *threshold*)
              [[new 'fib-L {:at :local}] <= (- n 1)]
              [[new 'fib-L {:at :local}] <= (- n 2)]
              (wait-for
               (=> answer1
                (wait-for
                 (=> answer2
                  [reply-to <= (+ answer1 answer2)])))))
             (t
              [[new 'fib-R {:at :local}]  <= (- n 1)]
              [[new 'fib-R {:at :remote}] <= (- n 2)]
              (wait-for
               (=> answer1
                (wait-for
                 (=> answer2
                  [reply-to <= (+ answer1 answer2)])))))))]

[class fib-L ()  ;; Fibonacci object for small n.
  (script
   (=> n @ reply-to
       (cond ((< n 2) !n)
             (t
              [[new 'fib-L {:at :local}] <= (- n 1)]
              [[new 'fib-L {:at :local}] <= (- n 2)]
              (wait-for
               (=> answer1
                (wait-for
                 (=> answer2
                  [reply-to <= (+ answer1 answer2)])))))))]
```

Figure 3.4: Another Solution to Locality Control for Fibonacci without Reflection

3.1.2 Dynamic Load Balancing for Parallel Search

Parallel search problem is one of the best-known examples where good dynamic load-balancing policies are effective, because of its high-degree of concurrency and the simplicity of the algorithm. The algorithm creates new objects as search subtasks[1].

The load-balancing system must decide: (1) how to distribute tasks to all the nodes; and (2) how to do so with minimal overhead. For this purpose, the load-balancing policies can generally be classified into centralized[12] and decentralized[32] one. In general, centralized policies can obtain good load balance, but task distribution is slow. On the other hand, the decentralized policies can quickly distribute tasks to all the nodes, but it incurs higher overheads to obtain a better balance.

Here, consider to construct a dynamic load balancing system that appropriately switches between these two policies according to the current state of the system: i.e., (1) when most nodes have enough tasks, the centralized policy (*balance*) is used; and (2) when many nodes are idle, the decentralized policy (*distribute*) is used. Furthermore, the system uses a customized scheduler to invoke the load balancing processes in an appropriate intervals[2]. To do this, the scheduler has a functionality to report after a certain number of tasks are scheduled.

Requirements to the reflective architectures in order to construct such a system are as follows:

- The object creation mechanism can be dynamically changed in order to support two object creation policies—*distribute* and *balance.*

  It is possible to support these two policies with a single mechanism that checks the current policy at each object creation. However, this approach poses a certain overhead because the policy is scarcely changed.

- Node-local schedulers can be customized so that they have the sentinels, which will notify after a certain number of tasks are scheduled.

- The way to collect current status of the system can be programmed in order to determine which object allocation policy is used.

3.1.3 Scheduling for Best-First Parallel Search

Best-first seacrh problems are to find the answer in the search tree that gives the best value to an given evaluation function. In parallel algorithms for best answer search problems, the order of execution, or *scheduling*, is important. In this subsection, based on a parallelized A*-search algorithm, requirements to reflective architectures are pointed out.

[1] Throughout the thesis, we call a *node* on a search tree as 'task' or 'subtask' to avoid confusion with processor elements.

[2] Such a customized scheduler example can be found in [32]. To adjust the load balancing intervals appropriately, the notion of *generation* is introduced. Roughly, the interval of the generation updating, which invokes the monitoring process, is proportional to the number of tasks in the system. Detailed discussion on the generation can be found in [32].

In an A*-search algorithm, there is a list l to hold all intermediate states concurrently searching. In one step of the search process, the most promising state is picked from l, and its successors are generated and put into l. An intermediate state that can only give worse answers than the best one found so far is removed from l. Parallelizations of the A*-search is straightforward: each node has its own l, and runs above search process individually; the difference is that promising states are exchanged between nodes in some way.

It is natural to think the A*-search algorithm can be described as an extension to a parallel (exhaustive) search program in object-oriented concurrent languages. However, the following difficulties arise to do so:

1. Since the scheduling order of objects is implicit, explicit scheduler must be introduced into the program to prioritize the most promising task. The introduction of an explicit scheduling entirely destroys the benefits of concurrent language. For example, in one step of search process could be concurrent; in this case, context switching at synchronization explicitly written makes program complicated and difficult to understand.

2. Some languages may allow to claim a scheduling priority for each object. However, it is not sufficient because the scheduling queue should be more freely accessed for task migration and pruning. Specifically, promising tasks in a node may be get migrated to the other nodes, and hopeless tasks (i.e., having bad estimation values) may be removed from the scheduling queue before executing.

3. An improvement for pruning, which switches the scheduling strategy from the depth-first to the breadth- and best-first, is difficult to be implemented.

Above difficulties leads the following requirements to reflective architectures:

- Scheduling mechanism should be customizable such as to support most-promising-task-first scheduling.
- In addition, it should allow to switch several scheduling mechanisms dynamically.
- Elegant way to obtain application level information by the scheduler. Without interrupting the control flow at the application level, such information should be obtained.

3.1.4 Object Allocation for N-Body Simulation

Some fast algorithms for N-body simulation has dynamic nature; i.e., data structure and message patterns can only be determined at the run-time. Studies shows that better load balancing can be obtained by specialized object allocation algorithm which uses the application-level knowledge[33]. In this subsection, object allocation for an algorithm for N-body simulation which is a parallelized version of the one proposed by Barnes and Hut[4] is discussed.

In that algorithm, a tree structure, which corresponds to the spatial distribution of particles, is used. The entire space is divided into cells (or subspaces) with child cells that covers part of the parent's space. The space is divided until all cells contains at most one corresponding particle. Each cell supposed to have the total mass and the center-of-gravity. Using this tree structure, force calculation for a particle can be approximated by using an non-terminating cell, which is sufficiently apart from the particle, as a group of particles.

The object allocation problem to this algorithm is how to place objects so that the amount of computation is evenly divided as well as the locality is persevered. This originates in the following characteristics of the algorithm: (1) the amount of computation for each object varies (usually depends on the number of particles in a cell), (2) the more communications taken place the closer two cells placed, and (3) since the life-time of cell object is not persistent (a tree structure is repetitively re-created for each force calculation), the load balancing should be done at allocation time rather than after the tree is created.

Without reflection, the allocation-time load balancing is accomplished by adding list of processor nodes into the other arguments such as list of particle data. Figure 3.5 shows the outline of the program (only a script for tree construction is shown). Underlined expressions are added for the object allocation. Controlled object allocation is done by following steps: (a) along with a message for tree construction, an additional parameter `node-list`, which denotes a list of nodes processors assigned to the cell, is passed; (b) when the cell has more than one particle, the `node-list` is divided in order to assign each sub-cells; (c) upon the creation of each sub-cell object, creation target node for it is explicitly specified, and (d) the assigned nodes are passed with other arguments.

This program has the following problems and advantages:

Mixture of object allocation codes and tree construction codes. Codes for object allocation are embedded in the codes for tree construction. This prevents the program from easy modification. For example, consider a new allocation strategy that uses another information (e.g., depth of the tree). To implement this strategy requires the modifications to the program although the application level algorithm is not changed at all.

In reflective systems, codes for object allocation should be placed at the meta-level, so that the codes for tree construction should not be modified by the modification to the object allocation strategies. Hence the requirement to the reflective system is that such data for object allocation can be passed at the meta-level along with the base-level arguments.

Application level information. Codes for dividing nodes according to the distribution of the particles can be straightforwardly described because application level information such as how particles are distributed can be directly available.

To describe this at the meta-level in a reflective language as easily as the non-reflective language, easy ways to access to application level information should be provided.

```
[class cell ()
 (state [subcells := '()]  ... )
 (script
  ;; script for tree construction
  (=> [:create-tree space particle-list node-list(a)]
   (if (= 1 (length particle-list))
     ;; it is a leaf cell
     ⟨set up with given particle⟩
     ;; it is an intermediate cell
     (let* ((subspaces (divide-space space))
            (subparticles
             (mapcar #'(lambda (space) (projection particle-list space))
                     divided-spaces))
            (subnodes
             (divide-nodes subparticles (length particle-list) node-list))(b))
       (setf
        subcells
        (mapcar #'(lambda (subspace subparticle-list subnodes-list)
                    ;; create a sub-cell
                    (let ((subcell [new cell {:at (first subnodes-list)}(c)]))
                      [subcell <= [:create-tree subspace subparticle-list
                                                subnodes-list(d)]]))
                subspaces subparticles subnodes))
       ⟨set up with created subcells⟩)))]
```

Figure 3.5: Object Allocation for N-Body Simulation in Non-Reflective Language

Hardware information. How to divide `node-list` is depends on the target hardware architecture (especially on the network topology). Thus there should be a portable way to deal with a set of nodes (e.g., partitioning) at the user's will.

3.1.5 Object Based Dynamic Load Balancing

Parallel applications, whose patterns of computation and data is known at the compile-time, can be executed efficiently by the static load-balancing. On the other hand, applications whose computational pattern and data distribution is *not* known at compile time, *dynamic load balancing* (DLB) should be done for efficient execution. The importance of DLB has been broadly noticed, and many studies on parallel (and practical) applications mention to it[33, 11, 42].

However, most works on DLB are application specific; i.e., how to collect load of nodes and how to balance are depends on the application level information, data structure, and program codes. On the other hand, this indicates that there are problems to support DLB at the language level. Followings are the problems and consequent requirements.

- For language level DLB systems, it is difficult to measure loads using application level information. For example, a DLB mechanism that appears in [37] measures a load of a node only by the usage of memory. Other systems would use number of objects, number of active (i.e., running) objects, or number of messages to be processed.

  It is usual that better load values can be measured by using application level information.

Thus the requirement to the reflective architecture is to easily incorporate application level information.

- Migration strategy—which object is moved to which node—is fixed, or far from flexible. In application specific DLB systems, migration strategy greatly varies. To support such variety of strategies the mechanism to determine migrating objects and target nodes should be opened to the user.

- Low-level techniques for migration may reduce cost for migration. For example, a mechanism proposed in [37] reduces the overheads of resolving (updating the references to migrated objects) by doing migration process with the node-local garbage-collection. To take advantages of such low-level optimizations, abstractions provided for migration should not conflict with them.

3.2 The Distributed Memory Reflective Architecture

Based on the requirements discussed in the previous section, a new reflective architecture, called the **Distributed Memory Reflective Architecture**, is designed. This section describes features of the architecture: mainly the language facility extensibility and the run-time system extensibility.

3.2.1 Overview

Firstly, the overview of the Distributed Memory Reflective Architecture is described. Similar to the *Hybrid Group Architecture* (HGA)[24], the proposed architecture allows per-object basis reflection through the metaobjects, and group basis reflection through the shared meta-level objects. However, our architecture distinguishes itself from the HGA in the notion of nodes (the abstraction of the processor elements on the distributed memory multicomputer) explicitly introduced at the meta-level. Figure 3.6 shows an overview. The key features are as follows:

- **Notion of nodes** is explicitly available at the meta-level while it is still transparent to the base-level computations.

- Customizable meta-level objects, including **node manager, scheduler,** and **metaobjects**, provide abstractions of the run-time system relevant to dynamic resource management.

- The **chain of executors**, which gives operational interpretation of base-level scripts, provides dynamic and localized language facility extension.

- **Interfaces to low-level systems** are provided so that information on (not customizable) run-time systems—memory usage, for example—can be used in DRM systems, and that facilities such as object migration and timer interrupts can be used.

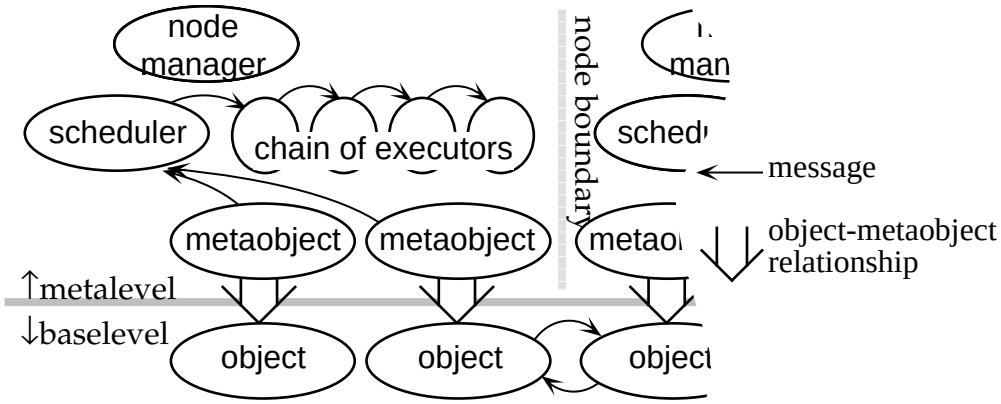

Figure 3.6: Overview of Distributed Memory Reflective Architecture

3.2.2 Language Facility Extension with Chains of Executors

Language facility extension is realized by the *Chains of Executors,* which provide interpretations of base-level programs with dynamic and localized customization. Its characteristics are as follows:

Operational Definition: An executor in a chain defines how an interpretation of each base-level expression goes in an operational way. The execution of a base-level expression is represented as a message to executors at meta-level. The message contains all information to evaluate the expression: the expression itself, an environment, a continuation, and so forth. Figure 3.8 shows a part of the the primary executor's definition. This definition is almost same to the ones of *evaluators* in ABCL/R[40] and ABCL/R2[24], which are meta-circular interpreters defined in a continuation passing style(CPS).

Since interpretations are given in an operational way, it is possible to define dynamic interpretations; i.e., the interpertation for an expression can be changed according to a dynamic condition. This ability distinguishes the operational extensions to the compile-time extensions.

Hierarchical Scope: Interpretation of a base-level script is defined by a sequence of executors at the meta-level. Table 3.1 shows names of executors, their containers, and scopes of controls. Each executor in the sequence defines only interpretations of expressions specialized to the objects in the scope of the executor; interpretations of other expressions are defined by another executors at higher rows in the hierarchy. This is done by a delegation mechanism. Usually, executors except for the primary executor define no specific interpretations, and the primary executor does for all expressions.

Executor Name	Container	Scope
primary executor	—	all baselevel objects
class executor	class object	objects in the class
node executor	node manager	objects in the node
object executor	metaobject	corresponding base-level object

Table 3.1: Hierarchy of Executors Chain

Figure 3.7 shows how a script execution is delegated through executors. Consider there are base-level objects foo1 and foo2 in class Foo, and bar in class Bar. Objects foo1 and bar reside at node #1, and object foo2 resides at node #2. In this case, script execution for each object is taken place along the bold arrows in Figure 3.7. For example, a script execution message for foo1 is delegated in the following order: object executor foo1, node exectuor #1, class Foo executor, and the primary executor.

Dynamic Replacement: Any executor in a chain except for the primary one, can be dynamically replaced. This is accomplished by sending a request to a container object, or by customizing a container object to replace its executor dynamically. This facilitates that language extensions in which the functionality of an operation changes dynamically, but not frequently. For example, consider a system that counts number of variable references for specified periods of time. Such a system can be realized by defining an executor which increases a counter whenever a variable is referred. Then the executor is used instead of the original one only for the specified periods. In this realization, the interpretation of variable reference in non-specified periods are defined by the original executor, and no unnecessary overheads are imposed on the variable references.

The reader may have a fear that the operational style extension could not be implemented efficiently. In fact, several reflective systems which have the language extensibility defined in operational style are implemented, and they prove that they are slow in the order of magnitude compared to the corresponding implementations of non-reflective languages[23]. However, more efficient implementation could be possible with sophisticated compilation techniques. Basic ideas for compiling base-level scripts with executors are discussed in Section 3.4.2.

3.2.3 Runtime System Extension with Meta-level Objects

3.2.3.1 Node Manager

Node manager is a meta-level object that represents a processor node. Any node manager can be referred from any object in the system by a pseudo variable or a function call. The forms

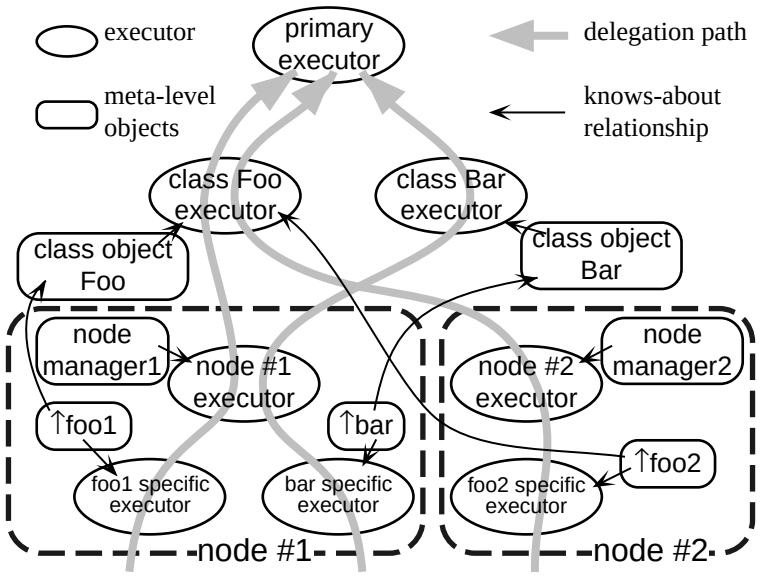

Figure 3.7: Example of Delegation Path

`node-manager` and `(node-manager-of` n`)` give references to the node manager where executing object resides and the node manager at node n, respectively. The definition of the default node manager is shown in Figure 3.9.

The major purpose of a node manager is to hold meta-level objects shared by members in the same node. By default, it holds two meta-level objects: scheduler and node-executor. In addition, arbitrary meta-level objects and information can be held by customized node managers.

The other role of a node manager is local object creation. By default, execution of an object creation form (`[new ...]` form) at the base-level produces a request message to the node manager. Then the node manager decides where the new object should reside; if it is local, the node manager itself creates a metaobject. Otherwise, it selects a remote node, and forwards the request to the node manager at the node.

To construct dynamic resource management systems, user defined node manager is often used to keep track of load information of nodes, exchange load information, decide scheduling/object allocation policies, and so forth.

3.2.3.2 Scheduler Object

Scheduler is a meta-level object that controls execution order of base-level objects in a node where the scheduler resides. In our architecture, the definition of scheduler can be customized; but the problem is what abstraction level the scheduler should provide. As we have observed in Section 3.1, an abstraction level that the scheduler controls the execution order and time quantum of objects

```
[class primary-executor ()
 (script
   ;; for object creation forms
  (=> [:do [:new class arguments annotation] env id] @ cont
     ;; send a creation request to a node manager
    [node-manager <= [:object-creation class arguments annotation] @ cont])
   ;; for past type message forms
  (=> [:do [:send-past target body reply-to] env id] @ cont
     ;; send a metalevel message to the metaobject of target
    [target <= [:message body relpy-to [den id]]]
    [cont <= nil])
   ;; for variable references
  (=> [:do [:variable name] env id] @ cont
     ;; return a value of name in env.
    [cont <= (lookup name env)])
        ⋮
 )]
```

Figure 3.8: Abridged Definition of a Primary Executor

```
[class node-manager (scheduler-class executor-class metaobject-class)
 (state [scheduler        := [new scheduler-class]]
        [executor         := [new executor-class
                              [new 'primary-executor nil]]]
        [metaobject-class := meta-object-class])
 (script
  (=> [:scheduler] !scheduler)
  (=> [:set-scheduler new-scheduler]      ; replacement of scheduler
      [scheduler <== [:copy-contents-to new-scheduler]]
      (setf scheduler new-scheduler))
  (=> [:executor] !executor)
  (=> [:set-executor new-executor]        ; replacement of node executor
      (setf executor new-executor))
  (=> [:new class arguments [&key (metaobject-class metaobject-class)
                             &rest annotation]]
      ![den [new metaobject-class class arguments annotation]]))]
```

Figure 3.9: Definition of a Node Manager

is enough for most dynamic resource management systems.[3]

A default scheduler, whose definition is presented in Figure 3.10, has a state variable that holds active objects. The variable is an abstraction of an active object queue in a node.

Basically, a scheduler receives an execution request by a message `[:request-execution thunk]`, where `thunk` is data structure holding enough information to execute. The execution is started by sending a context data (e.g., an expression, an environment, etc.) to an object specific scheduler.

[3] Another possible abstraction level would be that implementation of a message transmission—whether it is executed as a procedure call like sequential languages[36], or as a enqueue operation to the target object—is controlled by a scheduler. However, abstraction level is so low that it makes the architecture implementation specific, although it would improve performance in some cases.

```
[class scheduler ()
 (state (active-object-queue (make-queue))
        (status :idle)
        time-quantum)
 (script
  (=> [:request-execution thunk]
      (if (eq status :idle)
       (schedule-thunk thunk)
       (queue-put thunk active-object-queue)))
  (=> [:yield]
      (schedule-thunk (queue-get active-object-queue)))
  (=> [:yield-check thunk]
      (if (zerop (decf time-quantum))
        (progn  ;; time quantum is expired.
          (queue-put thunk active-object-queue)
          (schedule-thunk (queue-get active-object-queue)))
        (schedule-thunk thunk :time-quantum time-quantum)))
  (=> [:finished]
      (if (queue-empty? active-object-queue)
        (setf status :idle)
        (schedule-thunk (queue-get active-object-queue)))))
 (routine
  (schedule-thunk (thunk &key (next-quantum *default-time-quantum*))
    (setf status :busy
          time-quantum next-quantum)
    (let* ((context (thunk-context thunk))
           (id (thunk-id thunk))
           (executor [id <== [:executor]])
           ;; final-cont notifies the scheduler and the metaobject
           ;; of the end of the script execution.
           (finale-cont [new 'final Me id]))
      [executor <= [context . annotation] @ final-cont]))))]
```

Figure 3.10: Definition of Default Scheduler

When a script execution terminates, one of the following three messages is sent to the scheduler: `:yield` in case of the script execution should be suspended, `:yield-check` for preempting the script execution, and `:finished` in case of the script execution is finished.

Typical customizations of the scheduler are on the scheduling order and the scheduling time. As for the scheduling order, it is controlled by using priority queue. In this case, priority value is based on application level information explicitly passed from the base-level programs, or meta-level information that the metaobject holds. As for the scheduling time, time quantum is controlled based on some policy. Also, it is based on application or meta-level information.

3.2.3.3 Metaobject

Metaobject is a meta-level object that represents structural and object-specific aspects of an associated base-level object. It is designed so that the user can do customizations/introspections like follows: (1) the message queue can be inspected—this facility is useful to obtain the load value of individual objects; (2) the state variables can be inspected—useful to obtain application level information from the meta-level; and (3) behaviors to the events on the base-level object (i.e., behaviors to a message reception, invocation of a script execution, and end of script execution) can be customized—useful to attach information into an executable script and to record information, which is used for DRM systems.

Based on this observation, the definition of metaobject is in an event driven style(Figure 3.11). Basically, this definition is similar to the one in ABCL/R[40] and ABCL/R2[24]. The major difference is that our definition has a reference to a class object to hold the script data, while metaobjects directly hold the script data in ABCL/R and /R2.

3.2.3.4 Class Object

Class object is a meta-level object that holds resource shared by objects belongs to the class. By default, it serves just as a holder of the class executor.

In some reflective languages, the mechanisms for the method dispatching can be altered by regarding a class object as a unit of the method dispatching[17]. In this study, however, such extensibility is beyond our focus, although our architecture can support such extensibility by letting a class object process the method dispatching. This is because it has little to do with our aim, which is to provide efficient DRM systems by the reflection.

The definition of default class object is shown in Figure 3.12. It has one state variable `executor` to hold a class executor, and two scripts to reply/change the class executor.

3.2.4 Interface to Run-time Systems

The meta-level objects, such as the scheduler, are abstractions of the *modifiable* components of the run-time system. Not only the modifiable, but also interface to non-modifiable components is

```
[class metaobject (class arguments
                           &key (executor 'default-object-executor))
 (state [message-queue := (make-queue)]
        [class := class]
        [state := (make-state-memory class arguments)]
        [executor := [new executor]]
        [mode := ':dormant])
 (script
  ;; arrival of a message
  (=> [:message body reply sender]
      (queue-put (make-message body reply sender) message-queue)
      (when (eq mode :dormant)
        (accept-one-message)))
  ;; end of a script execution
  (=> [:finished]
      (if (queue-empty? message-queue)
        (setq mode :dormant)
        (accept-message)))
  ;; other meta-level operations on an object (omitted)
          ⋮
 )
 (routine
  ;; acceptance of a message
  (accept-one-message ()
   (let* ((m (queue-get message-queue))
          (c (find-script m class state))
          (thunk (make-thunk :context c :id Me))
          (scheduler [node-manager <== [:scheduler]]))
     (setq mode :active)
     [scheduler <=  [:request-execution thunk]])))]
```

Figure 3.11: Definition of Default Metaobject

defined in our architecture. For example, functions to utilize the run-time system services such as object migration, and to obtain the information on hardware (e.g., network configuration) are defined. Since the interface is defined as non-reflective (i.e., not modifiable) style, many low level optimizations can be applied. Detailed description of the interface appears in Appendix C.

3.2.4.1 Object Migration

In object oriented-concurrent languages, since each object requires computational power, some load-balancing system may resort to the object migration. The interface provided in our archi-

```
[class default-class-object (&key (class-executor 'default-class-executor))
 (state [executor := [new class-executor [new 'primary-executor]]])
 (script
  (=> [:executor] !executor)
  (=> [:set-executor new-executor]
      (setf executor new-executor)))]
```

Figure 3.12: Definition of Default Class Object

tecture is only to utilize the object migration functionalities that are provided by the run-time system. In other words, mechanism of the object migration cannot be altered like some reflective systems[26].

Actually, provided primitives are to direct that *what* object is to migrate to *which* node, and so forth, as appear in Appendix C.

3.2.4.2 System Level Information

Systems level information—state of the hardware and the low-level run-time system—is a good clue for the decision-making in user-defined DRM systems. For example, information from garbage-collecting system, such as amount of available memory in a node, is a good approximation of the load of the node.

Another kind of information that is necessary to the DRM programming is about the hardware configuration. In particular, network topology is important to have good application specific data (object) distribution. In our architecture, the network topology can be accessed in "mapped" data structure. That is, mapping functions between a real network topology and several well-known topologies are provided.

3.3 Reflective Programming Examples

To explain how to describe user defined DRM systems in our architecture, this section presents two simple examples. The one is a simple dynamic load balancing system, in which a customized scheduler, a customized node-manager, and a customized node-executor are defined at each node. The other is a heaviest-object-first scheduling system, in which a customized scheduler is defined at each node and a customized metaobject is defined for each base-level object. More practical examples can be found in Chapter 4.

3.3.1 Simple Dynamic Load Balancer

Simple dynamic load balancing (DLB) system is an example for the cooperation of the language facility extension and the runtime system extension. Figure 3.13 is an overview of the system.

The outline of this DLB scheme is as follows:

(a) When a node (say node A) becomes idle, the scheduler of the node notifies the node manager at the same node.

(b) The node manager randomly selects another node (say node B), and sends a request to it.

(c) The node manager B remembers the node A, and replaces its node executor with the remote create executor.

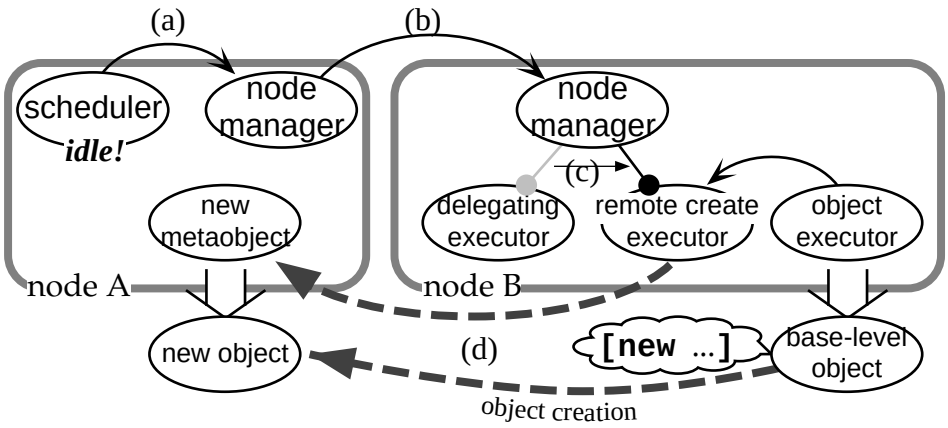

Figure 3.13: Overview of Simple Dynamic Load Balancing System

(d) When an object creation form is executed at node B, the remote create executor forwards the object creation request to the node manager A; consequently, a new object is created at the node A.

The definitions of the customized meta-level objects are shown in Figure 3.14. Since most parts of codes are same to the default ones, only customized parts are shown here.

3.3.2 Heaviest Object First Scheduling

As an example of the runtime system extension, a heaviest object first scheduling system, which schedules an object that has the longest message queue first, is presented.

In the Distributed Memory Reflective Architecture, the scheduler object controls the node-local scheduling. The design of the scheduler allows a straightforward extension for such customized scheduling. The heaviest-object-first scheduling is realized by a customized scheduler and a metaobject (Figure 3.15). The key customizations are as follows:

- The scheduler has a priority queue, instead of a FIFO queue in the default definition.
- Before the scheduler puts a thunk into the active object queue, it gets a priority value from the metaobject that creates the thunk via a message `[:message-queue-length]`.
- The metaobject has the script for a message `[:message-queue-length]` which returns the length of its message queue.

```
[class simple-LB-scheduler ()
 (state  ... )
 (script
  (=> [:finished]
      (if (queue-empty? active-object-queue)
        (progn (setf status :idle)
               [node-manager <= [:become-idle]](a))
        (schedule-thunk (queue-get active-object-queue)))))]

[class simple-LB-node-manager ( ... )
  (state  ...
   [idle-nodes := '()]
   [local-create-exec := [new 'delegating-executor]]
   [remote-create-exec := [new 'remote-create-executor]]
   [executor := local-create-exec])
  (script
   (=> [:become-idle]
       [(node-manager-of (random)) <= [:notify-idle Me]](b))
   (=> [:notify-idle idle-node-manager]
       (push idle-node-manager idle-nodes)
       (if (eq executor local-create-exec)
           (setf executor remote-create-exec)(c)))
   (=> [:get-idle-node]
       !(pop idle-nodes)
       (if (null idle-nodes)
           (setf executor local-create-exec))))]

[class remote-create-executor (primary-executor)
 (script
  (=> [:new class arguments annotation] @ cont
      (let ((idle-node [node-manager <== [:get-idle-node]]))
        [idle-node <= [:new class arguments [:at :local annotation]]
                   @ cont]))(d))]
```

Figure 3.14: Definition of Simple Dynamic Load Balancer

3.4 Implementation Issues

Elaborate dynamic resource management systems that are constructed at the meta-level of a reflective architecture will prove fruitless if the reflective implementation poses serious overheads. This section discusses implementation issues of proposed reflective architecture, especially of efficiency.

3.4.1 Runtime System Extensions

Compared to the language facilities extensions, overheads due to the runtime system extensions are less harmful. The reason is that a run-time extension usually replaces a run-time module, which has (relatively) high functionality. Assuming that the user defined run-time module is compiled as efficiently as the default one, overheads from the extensibility (e.g., indirect references) are negligible. (On the other hand, a language facility extension causes replacement of *primitive operations*, each of which is usually implemented in a few machine instructions.)

However, reflective facilities for the run-time system extension impose other overheads, which

```
[class HOF-scheduler ()
 (state [active-object-queue := (make-priority-queue)])
 (script
  (=> [:request-execution thunk]
      (if (eq status :idle)
        (schedule-thunk thunk)
        (let ((p [(thunk-id thunk) <== [:message-queue-length]]))
          ;; message queue length is used as a priority
          (queue-put thunk active-object-queue :priority p))))))]

[class HOF-metaobject ( ... )
 (state [message-queue := (make-queue)]
        ... )
 (script
  (=> [:message-queue-length]
      ;; message queue length is answered
      !(queue-length message-queue))
   ... )]
```

Figure 3.15: Definition of Heaviest Object First Scheduling System

are less obvious—interference with low-level optimizations. Suppose a reflective system provides a user customizable memory management mechanism. In the system, interfaces between the memory management system (e.g., how to allocate and reclaim memory fragments) are defined, and a default memory management module, which can be replaced with the user-defined one is provided. Here, a system with a user-defined module is less efficient than a (non-reflective) system with integrated and optimized memory management module (which can not be replaced), although the former system could be as efficient as the default system. This is because letting the system be flexible—making its memory management replaceable—poses overheads. For example, the system forces application programs to do their memory allocation via function calls, which can be done in a few machine instructions in an optimized implementation with an integrated memory management scheme. In fact, SML/NJ implementation requires only a few instructions to allocate a heap memory fragment[2].

Based on this observation, the design of the run-time system extension is conservative; i.e., only the run-time system extensions that are relevant to dynamic resource management systems are supported. As a result, other run-time facilities can not be replaced in order that as many low-level optimization techniques are applied as possible. For such design, the way to access the integrated modules (which can not be replaced/modified) from user defined modules should be supplied. In our architecture, a set of primitive functions are defined to do so. For example, a metaobject should search an executable script for a given message. To do this, a primitive function `(find-script` $\langle message \rangle$`)` is provided. A call to this function can be compiled into a table look-up operation, provided a compiler with enough knowledge for optimization.

3.4.1.1 Replication of Shared Meta-Level Object

A class object is a meta-level object whose primary purpose is to hold globally shared data—the class-executor in a state variable. If a class object changes its class executor, the change should affect how the class member objects interpret their scripts. In the definition, the current class executor is inquired for each script execution step. Thus naive implementations, in which the inquiry is represented as a remote (i.e., costly inter-node) messages, are too sluggish.

However, access to a class object are biased—frequency of modifications of the class executor is far lower than that of references. Hence it can be implemented efficiently by making a replication of class executors at each node[4]. Regarding a class object as a globally shared variable, the implementation using the replications can be achieved with some consistency protocol.

3.4.2 Language Facility Extensions

As a way for language facility extension, meta-level interpretation imposes overheads that can not be neglected. Previous studies regarding implementation of reflective systems (which are categorized for language facility extension, in this context) shows that the reflective computations usually slower than corresponding non-reflective computations in the order of magnitude.

In the course of language facility extension, smaller unit of operations is replaced with user-defined modules (or objects). A mechanism that makes such small operations replaceable forces relatively costly interfaces. For efficient implementation, as a matter of course, it is inevitable to remove such costly interfaces between replaceable modules—which is so called collapsing of meta-levels. Several studies have tried to do this, but as far as we know, few successful and practical solutions are presented. (One exception might be *metaobject protocol based compilers*, but their purpose and approach are different to ours, as will be discussed in Section 5.2.)

Here, we will show that such collapsing is possible to our architecture, while preserving its dynamic extensibility. Firstly, how a base-level script and a customized executor definition are collapsed by the partial evaluation is presented in the following subsection. Next a scheme to use (fully) compiled code while preserving the semantics of dynamic replacement of executors is explained.

3.4.2.1 Compilation to Collapse Meta-Level

Firstly, we focus on how to compile a base-level program with a *fixed* chain of executors(i.e., executors in a chain are not replaced dynamically). On this assumption, it is not so difficult to compile out the meta-level interpretations by the aid of sophisticated compilation techniques, like

[4] In the actual implementation, class objects will serve as abstraction to change compiled codes using mechanisms described in Section 3.4.2, and messages asking current executor will be eliminated by the partial evaluator. Replication technique is still applicable to such a case, by regarding a update operation as replacement of compiled codes.

the *partial evaluation.* Since several studies have already shown that this is possible[31, 3], a small example is shown here.

Let us look the object-creation form that appears in Section 4.1.1:

```
[new 'fibonacci {:control}]
```

In addition to the expression to be compiled, the compiler requires a set of executors. In this case, class- and node-executors are default ones—delegating executors, and either `fib-main-executor` or `fib-tip-executor` is used as an object-executor. Hence the two compiled codes are generated corresponding to object-executors. (Hereafter, we call each compiled code "main-code" and "tip-code," according to the object executor compiled with.)

Here, only the compilation of the main-code is explained. In the script of `fib-main-executor`, an object creation is defined as follows:

```
(=> [:do [:new class arguments [:control . annotation]] env id] @ cont
    (let* ((number (lookup 'n env))
           (extended-annotation
            (if (< number *threshold*)
              [:at :local :executor 'fib-tip-executor . annotation]
              annotation)))
      [[node-manager <== [:executor]]
       <= [:do [:new class arguments extended-annotation] env id]
       @ cont]))
```

The original expression is expanded with above definition. Expanded code[5] becomes as follows:

```
(let ((class 'fibonacci) (arguments '())
      (annotation '()) (env ⟨a compile-time environment⟩)
      (id ⟨the ID of a metaobject⟩) (cont ⟨cont. to next operation⟩)
      (parent ⟨the Fibonacci class object⟩))
  (let* ((number (lookup 'n env))
         (extended-annotation
          (if (< number *threshold*)
            [:at :local :executor 'fib-tip-executor . annotation]
            annotation)))
    [[parent <== [:executor]]
     <= [:do [:new class arguments extended-annotation] env id]]))
```

A partial evaluation of `(lookup 'n env)` gives an expression `n`. After substituting variables defined in the outer `let` form, the following code is generated:

```
(let* ((number n)
       (extended-annotation
        (if (< number *threshold*)
          [:at :local :executor 'fib-tip-executor]
          '())))
  [[⟨Fibonacci class object⟩ <== [:executor]]
   <= [:do [:new 'fibonnaci '() extended-annotation]
           ⟨the environment⟩ ⟨ID of metaobject⟩]
   @ ⟨cont. to next operation⟩])
```

Since most meta-level objects—the class object for Fibonacci, the node manager, and those of class- and node-executors—are not customized, partial evaluation of the following expression generates a primary executor:

[5] Since the annotation {`:control`} matches to the pattern `[:control . annotation]`, the value `annotation` is an empty list.

```
[⟨Fibonacci class object⟩ <== [:executor]]
```

Thus the expression becomes as follows:

```
(let* ((number n)
       (extended-annotation
        (if (< number *threshold*)
          [:at :local :executor 'fib-tip-executor]
          '())))
  [⟨primary executor⟩
    <= [:do [:new 'fibonnaci '() extended-annotation]
            ⟨the environment⟩ ⟨Id of metaobject⟩]
    @ ⟨cont. to next operation⟩])
```

The partial evaluator can not expand the expression `[⟨primary executor⟩ <= [:do ...]]` because the value of `extended-annotation` is not statically determined. Instead, it splits the body form of `let*` at the `if` expression. Resulted code is as follows:

```
(let ((number n))
 (if (< number *threshold*)
  [⟨primary executor⟩
    <= [:do [:new 'fibonnaci '() [:at :local :executor 'fib-tip-executor]]
            ⟨the environment⟩ ⟨Id of metaobject⟩]
    @ ⟨cont. to next operation⟩]
  [⟨primary executor⟩
    <= [:do [:new 'fibonnaci '() '()] ⟨the environment⟩ ⟨Id of metaobject⟩]
    @ ⟨cont. to next operation⟩]))
```

Since the compiler knows that above two message transmission forms are for object creation, they are compiled as same as the primitive object creation forms. Finally, we get the following form that switches two object creation forms according to a dynamical condition:

```
(if (< n *threshold*)
  ⟨local creation of  fibonacci with tip-code⟩
  ⟨remote creation of  fibonacci with main-code⟩)
```

3.4.2.2 Dynamic Code Replacement: Compilation Living with Dynamic Language Customization

Thus far, the meta-level programs can be compiled out assuming that a chain of executors is not dynamically alter its elements. The rest problem is how to deal with a chain of executors whose elements are dynamically replaced. This section describes a scheme, called the *Dynamic Code Replacement,* to efficiently do this.

Dynamic Code Replacement is a scheme to synchronize a replacement of the compiled code with the replacement of an executor on the assumption that every script has complied codes for each chain of executors (Figure 3.16). This assumption is acceptable because we are assuming the whole program is compiled at a time.[6]

However, when an executor in a chain is dynamically replaced, how do we statically determine which object is used as an executor? Since the 'containers' of an executor is a subject of user-customization, it is difficult to predict which object becomes as an executor at run-time.

[6] This assumption may be violated under interactive programming environments. Our scheme is still applicable by resorting to *dynamic compilation* techniques like which is developed for SELF[7].

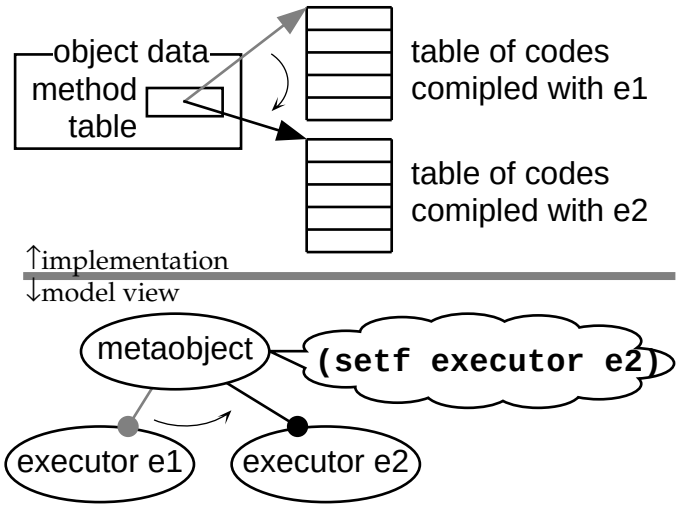

Figure 3.16: Synchronization of Compiled Codes with Replacement of Executors

A naive solution would regard any objects in any class as an candidate for an executor. However, this approach is unacceptable for the combinational explosion. Thus, we must predict which object is used as an executor more precisely. To do this, several approaches are applicable:

User declaration: By compelling the user who customizes containers to declare possible list of classes which may be used as executors. This lets the compiler easily know the possible combination.

Subclass or subtyping: When the language has an inheritance mechanism or a (strong) type system, an object used as an executor should be in a subclass/subtype of the default-object class/type. In this case, number of candidates for executors is fairly small.

Static analysis: Even in customized definitions of the containers, classes of objects that are used as executors can often be statically analyzed. For example, set of node executor classes used by `simple-LB-node-manager` in the definition of Figure 3.14 is statically determined as `delegating-executor` and `remote-create-executor`.

Chapter 4

Evaluation

This chapter shows evaluation of our proposed reflective architecture. It is evaluated in two ways: (1) by describing the resource management systems that are discussed in Section 3.1, and comparing to the other (non-reflective) approaches, and (2) by experiments on the prototype system that implements some resource management systems shown in (1).

4.1 Descriptions of Dynamic Resource Management Systems

In this section, descriptions of dynamic resource management systems in our reflective architecture are presented. Compared to the non-reflective approaches, the advantage of our architecture is also discussed.

4.1.1 Locality Control for Fibonacci

Firstly, the solution to the locality control for Fibonacci computation that is presented Section 3.1.1 is shown. The solution shown in Figure 4.1 consists of a base-level class definition, which is almost same to the original one except for a few annotations, and definitions of two object specific executors. By default, a Fibonacci has `fib-main-executor` as an object-specific executor, because of the annotation {`:object-executor 'fib-main-executor`}. The executor `fib-main-executor` only defines behaviors to object creation forms; all the other behaviors are defined by the other evaluators in a chain. The first script defines the execution of an object creation form with {`:control`} annotation, which corresponds the form `[new 'fibonacci {:control}]` at (a). In this script, a condition `(< n *threshold*)` is checked; when it holds, two annotations `:at :local` and `:executor 'fib-tip-executor` is added in order to create a new object at the local node with executor `fib-tip-executor`. Otherwise, no annotations are added, which means that decisions on the object creation is up to a node executor; thus it may be created at a remote node. The second script defines the execution of object creation without {`:control`} annotation, which corresponds the form `[new 'fibonacci]` at (b). In this case, a new object is created at the local node.

Once a condition `(< n *threshold*)` holds, a Fibonacci object with `fib-tip-executor` is created. For such an object, the execution of object creation forms are defined by the first script of `'fib-tip-executor`. The script adds two annotations `:at :local` and `:executor 'fib-tip-executor` in order to create a new object at the local node and with the same executor.

Hereafter, let us look how the problems of the non-reflective approach are solved in our architecture:

1. *Code for the resource management is clearly separated from the original algorithm.* In this example, decisions and designations for locality control are placed in the executors. Therefore, both the application and the resource management programs can be easily modified.

2. *Minimal redundancy for the application level description.* In our approach, different compiled codes can be obtained by compiling a Fibonacci object program with different executors.

3. *Better integration with other resource management policies.* In this example, the object allocation strategy for Fibonacci computation can be integrated with the node-level strategy. This is achieved *without code modification*. For objects whose `n` is larger than `*threshold*`, the executor `fib-class-executor` just delegates a message to the node-manager's one, allowing node-level locality control.

4. Redundant checks can be avoided. In this example, once the condition `(< n *threshold*)` holds for an object, it also holds for all sub-objects. When the condition holds for a Fibonacci object, objects created by that object will have the `fib-tip-executor`, which does not check the condition further. Furthermore, sophisticated optimization technique may make possible to implement Fibonacci computations after the condition has held by the recursive function calls like sequential languages.

4.1.2 Dynamic Load Balancing for Parallel Search

Here, a dynamic load balancing system for parallel search problems is presented. The overview of the system is shown in Figure 4.2. Roles of meta-level objects are as follows:

- Object decision maker manages the "generation," collects load information from all node managers, and decides which of the balancing policies should be used. If it decides to use the *balance,* this object also determines at which node a new object will be created for each node.

- Object scheduler is augmented to deal with *sentinels,* which is a boundary between generations. When the scheduler founds a sentinel at the top of the scheduling queue (i.e., all the tasks before the sentinel have been processed), it notifies the node manager.

```
[class fibonacci ()
 (script
  (=> n @ reply-to
      (cond ((< n 2) !n)
            (t
             [[new 'fibonacci {:control}](a) <= (- n 1)]
             [[new 'fibonacci](b)            <= (- n 2)]
             (wait-for answer1
               (wait-for answer2
                 [reply-to <= (+ answer1 answer2)])))))))]
 {:object-executor 'fib-main-executor}]

;; Object specific executor for larger n.
[class fib-main-executor ()
 (script
  ;; With :control annotation, check the condition.
  (=> [:do [:new class arguments [:control . annotation]] env id] @ cont
      (let* ((number (lookup 'n env))
             (extended-annotation
              (if (< number *threshold*)
                ;; When the condition holds, created object will use fib-tip-executor.
                [:at :local :executor 'fib-tip-executor . annotation]
                annotation)))
        [[node-manager <== [:executor]]
         <= [:do [:new class arguments extended-annotation] env id]
         @ cont]))
  ;; Without :control annotation, create locally.
  (=> [:do [:new class arguments annotation] env id] @ cont
      [[node-manager <== [:executor]]
       <= [:do [:new class arguments [:at :local . annotation] env id]
       @ cont])
  ;; All the other messages are delegated.
  (=> any-other-message @ cont
      [[node-manager <== [:executor]] <= any-other-message @ cont]))]

;; Object specific executor for smaller n.
[class fib-tip-executor ()
 (script
  ;; All object creations are taken place at local node.
  (=> [:do [:new class arguments annotation] env id] @ cont
      [[node-manager <== [:executor]]
       <= [:do [:new class arguments
                      [:at :local :executor 'fib-tip-executor . annotation]]
               env id]
       @ cont])
  ;; All the other messages are delegated.
  (=> any-other-message @ cont
      [[node-manager <== [:executor]] <= any-other-message @ cont]))]
```

Figure 4.1: Solution to the Locality Control in Distributed Memory Reflective Architecture

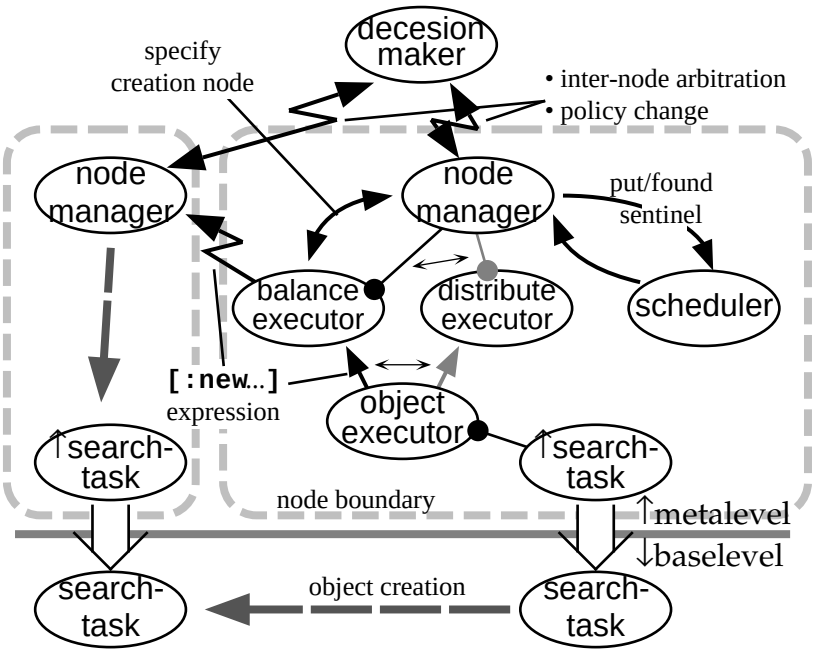

Figure 4.2: Dynamic Load Balancing System for Parallel Search

- Object node manager has two roles: (1) upon receiving a message to change the load balancing policy, it replaces current node executor; and (2) it puts a sentinel into the scheduling queue in order to find the boundary of a generation.
- Executors balance executor and distribute executor implement two load balancing policies—*balance* and *distribute,* respectively. Balance executor requests a node where new object will be created to the node manager, while distribute executor randomly chooses a node.

In this system, target nodes of object creations are controlled for load-balancing. When a search task is to create its subtask at the base-level, an object creation request is sent to the node executor at the meta-level. A target node of this object creation is determined by the current node executor, which is either balance or distribute.

Recalling the requirements listed in Section 3.1.2, the contribution of our reflective architecture is as follows:

- Chains of executors allows two object creation mechanisms switched according to the load balancing status of the system.
- The scheduler that can deal with sentinels can be implemented by extending the default scheduler definition.

```
[class search-task ()
 (script
  (=> [:search ⟨parameters⟩] @ reply-to
    (if ⟨is it a goal?⟩
        (progn
          {:found-answer ⟨value of the answer⟩}(a)
          [reply-to <= [:answer ⟨parameters⟩]])
        (let ((children-parameters (generate-next-level ⟨parameters⟩)))
          (dolist (child-parameter children-parameters)
            (let ((estimation ⟨estimated value of the child⟩))
              [[new 'search-task {:estimation estimation}(b)]
               <= [:search child-parameter]])))))))
 {:metaobject 'best-answer-metaobject}
 {:class-executor 'best-answer-executor}]
```

Figure 4.3: Base-level Program for Best Answer Search Problem

- The status of the system is gathered by the object decision-maker, which is user-defined meta-level object.

4.1.3 Scheduling for Best-First Parallel Search

A scheduling system for best-first parallel search problems is described on our reflective architecture. In this system, the base-level program (Figure 4.3) is a straightforward extension to the parallel (exhaustive) search problem. It is extended in following points: (a) when an answer is found, the value of the answer is notified to the meta-level, and (b) a subtask claims its estimation value in an object creation form. Prioritized scheduling and pruning are taken place at the meta-level. In addition, the system uses two scheduling strategies—depth first and best first—for better pruning.

The meta-system in a node, whose overview is given in Figure 4.4, consists of metaobjects that have additional parameters depth and estimation, a depth-first scheduler and a best-estimation-first scheduler, a class executor for class `search-task`, and a node manager that manages the scheduling policy and load balancing. Their roles are as follows:

Metaobject: A metaobject for class `search-task` holds parameters `estimation` and `depth` in addition to the other parameters that the default metaobject holds.

Schedulers: Both depth-first and best-first schedulers have a priority queue for the priority based scheduling. A search task at a deeper level in a search tree has higher priority for the former scheduler, and a task with a better estimation value for the latter. Until any answer is found, the depth-first scheduler is used; and then, the node manager replaces it with the best-first one.

Class Executor: The class executor defines interpretation of the object creation form (`[new ...]`) and the user-defined annotation (`{:found-answer` v`}`). For the object creation form,

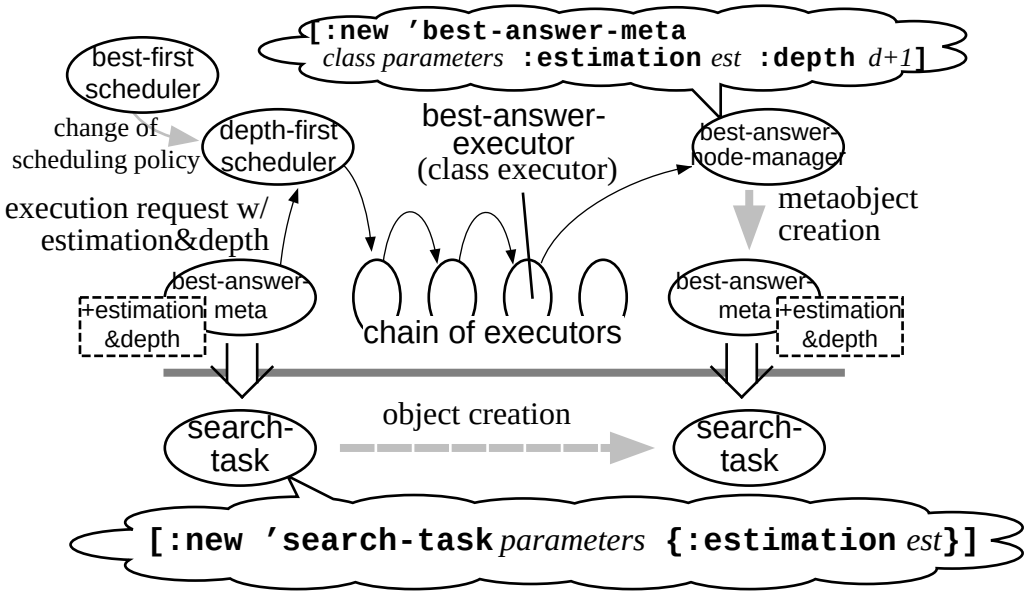

Figure 4.4: Overview of Scheduling System for Best-First Parallel Search

it obtains the value of depth parameter d of the creator task, then it adds a depth parameter $(d+1)$ for the metaobject being created. For the user-defined annotation, it simply reports the value to the node-manager.

Node Manager: The node manager has three roles: management of scheduling policies, management of the global best value of the answers found so far for pruning, and the load balancing in order that every node searches tasks that have global high priority.

Contributions of our reflective architecture in terms of the requirements listed in Section 3.1.3 are as follows:

- Customized schedulers are defined with only a few extensions to the default one.
- Two scheduling mechanisms can be used according to whether an answer is found or not.
- Estimation value of an intermediate search task, which is an application level information, can be available at the meta-level without intrinsic changes to the base-level program.

4.1.4 Object Allocation for N-Body Simulation

Object allocation system specific to the N-body simulation is described at the meta-level of our architecture. In this system, *list of nodes*, which has been passed along the base-level objects in a non-reflective implementation, is passed along the metaobjects. Figure 4.5 shows how base- and meta-level cooperates to control object allocation.

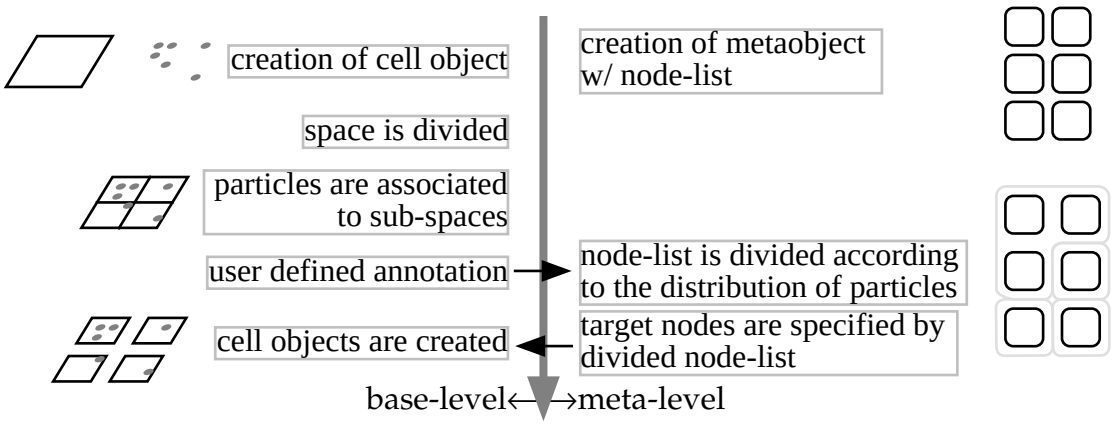

Figure 4.5: Overview of Dynamic Object Allocation System for N-Body Simulation

1. A cell object is created at the base-level. At the same time, a corresponding metaobject is created *with an additional argument node-list* at the meta-level.

2. At the base-level, the initialization script divides a given space into subspaces. Next, a given list of particles are distributed to the divided subspaces.

3. Upon an user-defined annotation, how the particles are distributed to the subspaces is informed to the meta-level. The metaobject divides the node-list according to the distribution of particles and associates to each subspaces.

4. When the cell object creates a sub-cell object at the base-level, a class executor, which handles the creation request, sends the request to a node in which the new object is to reside, and creates a metaobject at the node with the divided node-list associated to the sub-space.

The code for class `cell` is shown in Figure 4.6; those for the meta-level objects are in Section B.3. Operations for the object allocation and for the original computation are clearly separated. Contributions of our reflective architecture are: (1) using the customized class executor, the object allocation mechanism specific to a class can be described; (2) using the customized metaobject, meta-level information that is associated to a specific object can be dealt with; and (3) the user defined annotation allows to use the application specific information at the meta-level.

4.1.5 Object Based Dynamic Load Balancing

In this subsection, a load balancing system that is based on the load of individual object is presented. The balancing mechanism is simplified to make the explanation easier.

The system consists of customized metaobjects, each of which keeps track of the load, and load-balancer objects, which balances the loads among nodes.

```
[class cell ()
 (state  ... )
 (script
  (=> [:create-tree space particle-list]
   (if (= 1 (length particle-list))
     ⟨set up with given particle⟩
     (let* ((subspaces (divide-space space))
            (subparticles
             (mapcar #'(lambda (space) (projection particle-list space))
                     divided-spaces)))
       {:division subspaces subparticles}  ; notify the division to meta-level
       (setf
        subcells
        (mapcar #'(lambda (subspace subparticle-list subnodes-list)
                    (let ((subcell [new cell]))
                      [subcell <= [:create-tree subspace subparticle-list
                                                subnodes-list]]))
                subspaces subparticles subnodes))
       ⟨set up with created subcells⟩)))))
 {:class-executor 'cell-executor}
 {:metaobject 'cell-metaobject}]
```

Figure 4.6: Definitions for Dynamic Object Allocation for N-Body Simulation

The metaobject is responsible to monitor the load of its corresponding base-level object in some sense and to reply the load when asked. For example, the load would be measured by the length of its message queue, or the time spent for the recent script execution.

At each node, a load balancer object (whose definition is shown in Appendix B.4) is created. A load balancing process starts by a broadcast message. Receiving the message, the load balancer executes the following steps:

1. A local load value of the node (which is defined as a sum of loads of every active objects) is calculated. This is done by getting a list of active objects from the scheduler.

2. It exchanges the local load value with neighboring nodes.

3. Based on exchanged loads, the amount of loads that should be migrated to other nodes is determined.

4. Requests for migration of objects are issued according to the amount determined in the above step. (If the load of node is smaller compared to the neighboring nodes, this step is not taken place.)

Contributions of our reflective architecture are: (1) the customized metaobject is good abstraction to deal with the object specific information—in this case, the object specific load; (2) the user defined meta-level object that does load balancing can be defined; and (3) that the definition of the load balancer can be written in an object-oriented concurrent language is a good way to describe inter-node negotiations like the load balancing.

4.2 Experiment with Prototype System

A locality control system for search problems is implemented on a prototype system which is running on a multicomputer. In this section, outline of the system and results of our experiments are presented.

4.2.1 Locality Control at Meta-Level

The locality control system used here is almost same to the one presented in Section 4.1.1. For an object at the shallow level in a search tree, its child objects are created at randomly-chosen nodes. For objects at the deep level in the search tree, on the other hand, all its child objects are created at the same node to the creator's.

The target application program used here is the N-Queens problem in which a node in a search tree is implemented as a concurrent object. Since the depth in the search tree is not explicitly available in the base-level program, it is maintained by metaobjects. A metaobject is created with depth parameter in addition to the default parameters. The metaobject adds the depth parameter whenever it sends an execution request to a scheduler. The parameter is passed along with an ordinary context of a script execution, such as environment and metaobject's ID. On an object creation, an object executor determines a node where new object is created, based on the value of the depth parameter, instead of picking up the value of a base-level variable (it is obtained through the form `(lookup 'n env)`).

4.2.2 Improvements for Better Utilization

Although the above locality control system will reduces overheads from communications with remote nodes, it also leads to low processor utilization because of load-imbalance. An improvement for better utilization, which adjusts the threshold depth according to the load balancing status, is also implemented at the meta-level of our reflective architecture. In this system, a monitoring object, which checks if there are any idle node at certain time intervals, is defined. When one or more nodes become idle, the threshold depth is decreased globally. The outline of the definition for load monitor object is listed in Figure 4.7.

4.2.3 Experiments and Results

A prototype compiler for an object-oriented concurrent language is developed[1], and the above locality control systems are programmed[2] on it. All experiments are taken place on 64-nodes

[1] Technically speaking, the compiler generates C programs in which a basic-block in a script is implemented as one C-function.

[2] The implementation has several shortcuts and redundancies due to the limitation of the prototype system. (1) Since a compiler that collapses executor scripts is not established yet, depth parameter is explicitly introduced into the base-level programs. (2) Instead of the object-executor, the node manager decides where a new object is

```
[class load-monitor (initial-threshold)
 (state [current-threshold := initial-threshold])
 (script
  (=> [:timer-event]
   Message [:idle?] is broadcasted to all node managers,
   and counts number of idle nodes.
   (when ⟨number of idle nodes is large⟩
     (decf current-threshold)
     Message [:set-threshold current-depth] is broadcasted
     to all node managers)
   (set-timer-event [:timer-event] time-span)))]
```

Figure 4.7: Definition of Load Monitor Object

AP1000, which is a multicomputer having 25MHz sparc processors connected via torus network.

Table 4.1 shows the elapsed times for the execution with the different level of depths. The first row in each table shows a result without the locality control system; i.e., all object creations are remote. In the best case, the execution with the locality control system exhibits approximately two times faster than the one without the system. On the other hand, the utilization figures, which shows how long processors are busy in a total computation time, decreases for the executions with small threshold values.

Table 4.2 shows the elapsed times and the utilizations with different locality control strategies. With locality control strategies "depth" and "depth+adjust," the value of (initial) threshold depth is 5, which resulted in worse utilizations. The table shows that the dynamic adjustment of the depth threshold results in only few percent performance improvements while the utilization is improved (in 12-Queens case; it is not so in 11-Queens case, however). It is presumed that the depth adjustment causes unnecessary remote creation of objects, and this overhead covers the improvements of the utilization.

To summarize, the locality control system, which is implemented at the meta-level, improves the performance of a search program by the factor of two in the best case. The utilization of node processors can be improved by the dynamic adjustment of the depth threshold, but the improvement is small.

created. Thus the depth threshold is tested redundantly. (3) The implementation of the node manager is redundant; a dispatching object is inserted for dynamic replacement of node managers.

12-Queens

threshold depth	elapsed time (sec.)	utilization(%)
—	13.88	97
8	7.655	95
7	6.595	93
6	6.541	87
5	7.301	77

11-Queens

threshold depth	elapsed time (sec.)	utilization(%)
—	2.742	95
7	1.511	92
6	1.403	88
5	1.524	81
4	1.917	70

Table 4.1: Benchmark Results for Different Threshold Depth Parameters

12-Queens

locality control strategy	elapsed time (sec.)	utilization(%)
random	13.88	97
depth	7.301	77
depth+adjust	7.189	86

11-Queens

locality control strategy	elapsed time (sec.)	utilization(%)
random	2.742	95
depth	1.524	81
depth+adjust	1.465	80

Table 4.2: Benchmark Results for Different Locality Control Strategies

Chapter 5

Related Work

5.1 Distributed/Parallel Reflective Languages

Several reflective languages/systems have been proposed for distributed/parallel computing. Although there are similarities between such reflective architectures and ours, the meta-level abstraction—what can be altered at the meta-level—is different. Our architecture is designed for flexible control of the dynamic resource management for highly concurrent applications, while others are designed mainly to provide flexible language mechanisms in distributed environment, which involves heterogeneity, fault-tolerance, etc.

- OCore is a concurrent language for highly-parallel computing. It has a reflective architecture which aims to provide controls of parallel computations through the meta-level programming[38]. This work is distinguished from ours in the target of control. In OCore, its meta-level architecture focuses on describing exception handling, (static) object distribution, profiling/debugging statistics, and so, forth.

- AL-1/D is an object-oriented concurrent reflective language for distributed environment[26, 27]. One of the major purpose of AL-1/D is to provide flexible controls to the low-level implementation of the language. Hence the low-level mechanisms, such as marshaling, can be customized by reflection in AL-1/D, while our architecture expects that the non-customizable low-levels are efficiently implemented by allowing integrated low-level optimizations.

- The design principle of RbCl[15] is to make as many low-level mechanisms reifiable as possible for maximum flexibility. On the other hand, it is not easy to construct a new resource management system that controls facilities provided by such reifiable low-level mechanisms. Since a controlled facility is represented as a collective behavior of low-level mechanisms, the user must carefully implement his/her own policies without conflict. Such discipline could be enforced by the metaobject protocol, but the MOP (metaobject protocols) for RbCl is not completely defined.

- Open C++[8] is a sequential reflective language; i.e., no distributed or parallel computing facilities are provided by default. As a result, all such facilities are provided by the meta-level programming; i.e., they are implemented using the low-level mechanisms available at the meta-level.

  An interesting feature of Open C++ is its optimization. Code analysis makes a reflective method call in Open C++ almost equal speed to the one in native C++ in the best case[9]. This optimization is suggestive to the compiler development, although such code analysis would be difficult in many reflective languages, which has more elaborate architectures.

5.2 Metaobject Protocol based Open Compilers

Open compilers are the system that can extend/modify its compilation. Inspired by the reflective systems, there are several studies to provide *meta-object protocols* (MOP) to open compilers[18, 30], so that customizations on a compiler can be achieved in more modular and elegant way.

As a way for language extension, MOP based open compilers have similar advantages to reflective languages; it provides a modular and clean interface for the language extension, and the extension can be achieved with minimal modification to the application program (mostly, inserting several annotations).

The important difference between MOP based open compilers and our reflective architecture is the way of customizations; *operational* way in ours, while *rule-based* way in open compilers (here, by *rule-based,* we mean customization that is described as extensions to compilation rules).

For language extensions that is defined for dynamic resource management systems, the operational approach is advantageous. This is because open compilers are mainly designed for *static* (compile-time) extensions, and not for *dynamic* (run-time) extensions, in which a behavior of a program fragment dynamically changes. For example, consider a case that a programmer wants an operation in his program to have two different behaviors according to a certain dynamic condition. In our approach, it can be described as a conditional branch that selects one of the two operations. With good compilation scheme, it can provide two different compiled code is selected according to the condition. On the other hand, MOP based open compilers does not provide to handle such a dynamic condition in a elegant way, since the compiler is designed to handle things that can be statically decided, but not things that can be only dynamically decided.

5.3 Open Implementations

Open implementation is a way to build a flexible system by constructing components of the system in a modular and exchangeable way. Similar to reflective languages, such system can extend its internals. The major difference is, however, the approach of extension. In open implementation systems, a user-defined run-time *module* can be used as a replacement to an original one, but the

module is regarded as a black box to the system. In reflective language, on the other hand, user extension is done by describing a operational specification (or program) to the meta-level, which has more possibility to efficiently compiled out by analyzing the *semantics* of the extension.

Distributed Operating System: Apertos[43] is a reflective operating system for distributed environment. In Apertos, 'object migration' is a primitive operation that changes its *meta-space.* The purpose of object migration is for using different OS services, such as paging mechanisms.

Distributed Shared Memory: Some systems for distributed shared memory provide several memory coherence protocols and allow the user to use an appropriate one for each shared object[5]. Moreover, some systems can dynamically change coherence protocols for a shared data.

Thread Scheduling: By providing an interface between an OS kernel and a user-level thread scheduler, significant performance improvement can be achieved[1]. Although this has similarity to reflective systems in terms of incorporating the user's policy to the kernel, there are several differences: (1) the performance improvement is mainly due to the reduction of the interactions between the user-level and the kernel; and (2) actions to only a fixed set of events (e.g., I/O blocking) can be controlled by the user-programming.

Chapter 6

Conclusion and Future Work

We have proposed a new reflective architecture called the *Distributed Memory Reflective Architecture*, whose purpose is to easily describe dynamic resource management systems for highly concurrent applications. The architecture is designed under the requirements from dynamic resource management systems for non-trivial concurrent applications. The following features of the architecture contribute for describing various resource management systems:

- Notion of node processors, which is available at the meta-level, enables to describe resource management systems relevant to the distributed memory multicomputers such as a load-balancing system and an object allocation system.
- Meta-level objects, including node manager, scheduler, and metaobjects, provide abstractions of the language's run-time system, which facilitate descriptions of various resource management systems in an encapsulated manner.
- Language facility extensibility, which is implemented by the chains of executors, enables the user to give different interpretations to a single program. In addition, it provides clean ways to access application-level information from a resource management system.
- Interface definitions of the run-time systems provide easy access to the low-level and implementation dependent mechanisms (e.g., object migration and network topology) in a portable way.

We have also put careful consideration to the efficiency. Since the meta-level design, which is based on the requirements from the realistic DRM examples, does not expose unnecessary part of the run-time systems (e.g., the memory management and the network messages are not modified by reflection), hence many low-level optimization techniques are applicable. Language facility extensibility, which is provided by chains of executors, could be efficiently implemented, based on our proposed scheme Dynamic Code Replacement and sophisticated compilation techniques.

Proposed architecture has been evaluated by describing dynamic resource management systems for non-trivial concurrent applications. These systems include a load balancing system for parallel search problems, a scheduling system for best-first parallel search problems, an object allocation system for N-body simulations, and a load balancing system based on individual objects. The architecture is also evaluated through an experimental locality control system developed on a prototype system running on a multicomputer. A search application with the locality control system is executed twice as fast as the one without locality control (i.e., it has a naive object allocation strategy).

Future work of this study are broadly divided into three directions: refinement of the reflective architecture, development and formalization of implementation techniques, and further examples of dynamic resource management systems described on our architecture.

As for reflective architecture, *heterogeneous object group*, which is introduced in ABCL/R2[24], would be incorporated in order to share meta-level resources among objects in different classes. Consider a large-scale application in which several program modules—each module consists of many kinds of objects and has high concurrency—are simultaneously running. The object group is a good abstraction as a unit of module specific DRM. At the meta-level, such an object group can be modeled as a *group manager*, which holds shared resources like executors as state variables. The other benefit of the object groups is that it can generalize class objects. Since a class object at the meta-level is defined only to share an executor in our architecture, it could be defined as a special kind of an object group.

As for implementation, elemental techniques should be advanced further. Recent studies have shown that partial evaluation techniques to collapse meta-levels of a reflective language are becoming feasible[31, 3, 9] (however, all target sequential reflective languages). This encourages us to develop a partial evaluator for our architecture. In addition, as studies on metaobject protocol based compilers[18] shows, extensibility to compilers is also useful mechanism. Our ambition is to enable such compiler level extensions with operational descriptions. Refinement to Dynamic Code Replacement scheme is also important. Although the applicability of this idea could be wide, we must investigate detailed mechanism of it, such as how to do it efficiently when an executor that covers more than one object is replaced, representation of a table holding the code fragments compiled with different executors, how to replace a code fragment for a running object. To do this, the dynamic compilation/de-compilation techniques such as the one developed for SELF[13] would be useful. To concentrate these implementation issues, we are planning to design a compiler that does partial evaluation for a dialect of Scheme.

As for applications, what we should do is further experiments on our prototype system, and more descriptions of dynamic resource management systems. Both are essential to refine the architecture's detail. For example, the following DRM systems would be investigated: scheduling and load balancing systems for the TimeWarp system[16, 6, 19], scheduling systems at coarse grain

concurrency like the self-scheduling[28, 20], and scheduling systems for soft real-time systems[14].

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa. Duplication and partial evaluation to implement reflective languages. In *Proceedings of OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, Washington, D.C., September 1993.

[4] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[5] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of Symposium on Principles & Practice of Parallel Programming (PPOPP)*, Seattle, WA, March 1990.

[6] Christopher Burdorf and Jed Marti. Non-preemptive time warp scheduling algorithms. *ACM Operating Systems Review*, 24(2):7–18, 1990.

[7] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–15, Phoenix, Arizona, October 1991.

[8] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 1993.

[9] Shigeru Chiba and Takashi Masuda. Open C++ and its optimization. In *Proceedings of OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, Washington, D.C., September 1993.

[10] Olivier Danvy and Karoline Malmkjær. Intensions and extensions in the reflective tower. In *Proceedings of Conference on LISP and Functional Programming*, pages 327–341. ACM, 1988.

[11] Karen D. Devine, Joseph E. Flaherty, Stephen R. Wheat, and Arthur B. Maccabe. A massively parallel adaptive finite element method with dynamic load balancing. In *Proceedings of Supercomputing*, pages 2–11, Portland, OR, November 1993.

[12] Masakazu Furuich, Kazuo Taki, and Nobuyuki Ichiyoshi. A multi-level load balancing scheme for OR-parallel exhaustive search programs on the Multi-PSI. In *Proceedings of Symposium on*

Principles & Practice of Parallel Programming (PPOPP), pages 50–59, Seattle, WA, March 1990. ACM.

[13] Urs Hölzle, Craig Chambers, and David Unghar. Debugging optimized code with dynamic deoptimization. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 32–43, San Francisco, CA, June 1992.

[14] Yasuaki Honda and Mario Tokoro. Soft real-time programming through reflection. In *Proceedings of International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, pages 12–23, Tama City, Tokyo, November 1992.

[15] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCl: A reflective object-oriented concurrent language without a run-time kernel. In *Proceedings of International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, pages 24–35, Tama City, Tokyo, November 1992.

[16] David R. Jefferson. Virtual time. *Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[17] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

[18] Gregor Kiczales, John Lamping, and Anurag Mendhekar. What a metaobject protocol based compiler can do for lisp. Draft paper, 1994.

[19] Yi-Bing Lin and Edward D. Lazowska. A study of time warp rollback mechanisms. *Transactions on Modeling and Computer Simulations*, 1(1):51–72, 1991.

[20] Steven Lucco. A dynamic scheduling method for irregular parallel programs. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 200–210, San Francisco, CA, June 1992. ACM.

[21] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 147–155. ACM, October 1987.

[22] Pattie Maes. Issues in computational reflection. In P. Maes and D. Nardi, editors, *Meta-Level Architecture and Reflection*, pages 21–35. Elsevier Science, North-Holland, 1988.

[23] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 127–145, Vancouver, B.C., October 1992. ACM.

[24] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 1991.

[25] Jeff McAffer. The CodA MOP. In *Proceedings of OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, Washington, D.C., September 1993.

[26] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. AL-1/D: Distributed programming system with multi-model reflection framework. In *Proceedings of International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, pages 36–47, Tama City, Tokyo, November 1992.

[27] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. Object migration on reflective distributed programming system AL-1/D. Draft paper, 1993. (In Japanese).

[28] Constantine D. Polychronopoulos and David J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C–36(12):1425–1439, December 1987.

[29] Ramana Rao. Implementational reflection in Silica. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 251–266, 1991.

[30] Luis Rodriguez, Jr. A study on the viability of a production-quality metaobject protocol-based statically parallelizing compiler. In *Proceedings of International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, pages 107–112, November 1992.

[31] Erik Ruf. Partial evaluation in reflecitve system implementation. In *Proceedings of OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, Washington, D.C., September 1993.

[32] Reiko Satoh, Hiroyuki Sato, and Katsuto Nakajima. A diffusional load balancing scheme on loosely coupled multi-processors =LLS-G=. In *Joint Symposium on Parallel Processing (JSPP)*, pages 363–369, Tokyo, May 1993. (In Japanese).

[33] Jaswinder Pal Singh, Chris Holt, John L. Hennessy, and Anoop Gupta. A parallel adaptive fast multiploe method. In *Proceedings of Supercomputing*, pages 54–65, Portland, OR, November 1993.

[34] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Conference record of Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[35] Guy L. Steele Jr. *Common Lisp the Language*. Digital Press, second edition, 1990.

[36] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Proceedings of Symposium on Principles & Practice of Parallel Programming (PPOPP)*, San Diego, CA, May 1993. ACM SIGPLAN.

[37] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. Incorporating locality management into garbage collection in massively parallel object-oriented languages. In *Joint Symposium on Parallel Processing (JSPP)*, pages 277–282, may 1993.

[38] Takashi Tomokiyo, Hiroki Konaka, Munenori Maeda, Atsushi Hori, and Yutaka Ishikawa. Meta-level architecture in *ocore*. In *Proceedings of OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, Washington, D.C., September 1993.

[39] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflecitve tower. In P. Maes and D. Nardi, editors, *Meta-Level Architecture and Reflection*, pages 111–134. Elsevier Science, North-Holland, 1988.

[40] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 306–315, San Diego, CA, September 1988. ACM. (revised version in [45]).

[41] Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In *Proceedings of REX School/Workshop on Foundation of Object-Oriented Languages*, 1990.

[42] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, September 1993.

[43] Yasuhiko Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 414–434, Vancouver, B.C., October 1992.

[44] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In Stephen Cook, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 89–106. Cambridge University Press, 1989.

[45] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System.* MIT Press, Cambridge, MA, 1990.

[46] Akinori Yonezawa, Satoshi Matsuoka, Masahiro Yasugi, and Kenjiro Taura. Implementing concurrent object-oriented languages on multicomputers. *IEEE Parallel & Distributed Technology*, 1(2):49–61, May 1993.

[47] Akinori Yonezawa and Takuo Watanabe. An introduction to object-based reflective concurrent computations. In *Workshop on Object-Based Concurrent Programming*, pages 50–54, 1989. published as ACM SIGPLAN Notices 24(4).

Appendix A

Meta-Circular Definition of Distributed Memory Reflective Architecture

The detailed definitions of the meta-level objects are presented below. Definitions are slightly complicated compared to the ones in Chapter 3, because the definitions in this appendix include code to handle various kinds of annotations.

Node Manager: The default node manager requires three keyword arguments at creation time. Top-level annotations can specify the values of these arguments. For example, a top-level annotation {`:node-executor 'my-node-executor`} specifies the value of `node-executor` to be `my-node-executor`, which is `'default-node-executor` without annotations.

```
[class default-node-manager (&key (scheduler 'default-scheduler)
                                  (node-executor 'default-node-executor)
                                  (metaobject-class 'default-metaobject))
 (state [scheduler := [new scheduler]]
        [executor := [new node-executor]]
        [default-metaclass := metaobject-class])
 (script
  (=> [:scheduler] !scheduler)
  (=> [:set-scheduler new-scheduler]
    [scheduler <== [:copy-contents-to new-scheduler]]
    (setf scheduler new-scheduler))
  (=> [:executor] !executor)
  (=> [:set-executor new-executor]
    (setf executor new-executor))
  ;; local object creation
  (=> [:new class arguments &key (metaclass default-metaclass) . annotation]
    ![den [new metaclass class arguments . annotation]]))]
```

Metaobject: A metaobject is created with parameters `class` and `arguments`. In addition, it takes a keyword argument `executor` specifying the class of an object executor. An annotation in an object creation form can specify the value of this keyword argument. For example, the execution of a form `[:new 'foo {:executor 'my-object-executor}]` eventually causes a creation of a metaobject whose keyword argument `:executor` is `'my-object-executor`. For the time being,

the relationship between an object and its class is fixed. Thus the value of the state variable `class` must not be changed, and the script `[:class]` must return the value of `class` in any case.

```
[class default-metaobject (class arguments
                           &key (executor 'default-object-executor))
 (state [message-queue := (make-queue)]
        [class := class]
        [state := (make-state-memory class arguments)]
        [executor := [new executor]]
        [mode := ':dormant])
 (script
  ;; arrival of a message
  (=> [:message body reply sender]
      (queue-put (make-message body reply sender) message-queue)
      (when (eq mode :dormant)
        (accept-one-message)))
  ;; end of a script execution
  (=> [:finished]
      (if (queue-empty? message-queue)
        (setq mode :dormant)
        (accept-message)))
  (=> [:executor] !executor)
  (=> [:set-executor new-executor]
      (setf executor new-executor))
  (=> [:class] !class))
 (routine
  ;; acceptance of a message
  (accept-one-message ()
   (let* ((m (queue-get message-queue))
          (c (find-script m class state))
          (thunk (make-thunk :context c :id Me))
          (scheduler [node-manager <== [:scheduler]]))
     (setq mode :active)
     [scheduler <=  [:request-execution thunk]])))]
```

`Thunk` is a data structure that contains a context (static information for script execution such as an expression and environment) and a reference to the metaobject.

Scheduler: The default scheduler can be created with optional keyword parameter, which specifies the length of initial time quantum for script execution.

```
[class default-scheduler (&key (initial-time-quantum *default-time-quantum*))
 (state [active-object-queue := (make-queue)]
        [status := :idle]
        [initial-time-quantum := initial-time-quantum]
        time-quantum)
 (script
  (=> [:request-execution thunk . annotation]
    (if (eq status :idle)
     (schedule-thunk thunk . annotation)
     (queue-put thunk active-object-queue)))
  (=> [:yield]
    (schedule-thunk (queue-get active-object-queue)))
  (=> [:yield-check thunk]
    (if (zerop (decf time-quantum))
      (progn  ;; time quantum is expired.
        (queue-put thunk active-object-queue)
        (schedule-thunk (queue-get active-object-queue)))
      (schedule-thunk thunk :time-quantum time-quantum)))
  (=> [:finished]
```

```
    (if (queue-empty? active-object-queue)
      (setf status :idle)
      (schedule-thunk (queue-get active-object-queue))))
  ;; following two scripts are for dynamic replacement of schedulers
  (=> [:copy-contents-to scheduler]
    [scheduler <= [:set-active-object-queue (listify active-object-queue)]])
  (=> [:set-active-object-queue thunk-list]
    (queue-set-list active-object-queue thunk-list)))
 (routine
  (schedule-thunk (thunk &key (next-quantum initial-time-quantum) . annotation)
    (setf status :busy
          time-quantum next-quantum)
    (let* ((context (thunk-context thunk))
           (id (thunk-id thunk))
           (executor [id <== [:executor]])
           (finale-cont [new 'finale Me id]))
      [executor <= [context . annotation] @ finale-cont]))))

;; when a script is finished, a result is sent to this object.
[class finale-cont (scheduler metaobject)
 (script
  (=> any-value
    [scheduler <= [:finished]]
    [metaobject <= [:finished]]))]
```

Class Object: The default class object only holds a class executor.

```
[class default-class-object (&key (class-executor 'default-class-executor))
 (state [executor := [new class-executor [new 'primary-executor]]])
 (script
  (=> [:executor] !executor)
  (=> [:set-executor new-executor]
      (setf executor new-executor)))]
```

Delegating Executors: By default, object-, node-, and class-executors only delegate received messages to executors residing above; i.e., node-, class-, and primary-executors, respectively. Note that the target of delegation is obtained for each script execution (except for the primary executor). This assures that any executors in a chain—except for the primary one—can be replaced dynamically.

```
[class default-object-executor ()
 (script
  (=> message @ cont  ;; delegated to a node-executor
      [[node-manager <== [:executor]] <= message @ cont]))]

[class default-node-executor ()
 (script
  (=> message @ cont  ;; delegated to a class-executor
    (match message
      (is [:do _ _ Id . _]
        (let* ((class [Id <== [:class]])
               (class-executor [class <== [:executor]]))
          [class-executor <= message @ cont]))))))]

[class default-class-executor (pe)
 (state [primary-executor := pe])
 (script
  (=> message @ cont  ;; delegated to the primary-executor
    [primary-executor <= message @ cont]))]
```

Primary Executor: The primary executor defines default semantics of the base-level language in an operational way. Presented code is a part of the definition; this is because enormous features of the base-level language (the base-level language includes a subset of Common-Lisp[35]) requires many lines of definitions, which are not crucial in this paper.

```
[class primary-executor ()
 (script
  (=> [:do exp env id scheduler . context-annotation]  @ cont
   (match exp
     ;; object creation
     (is [:new class arguments &key at target-node . metaobj-annotation]
       (let ((target-node-manager
              (case at
                (:remote (or target-node (random max-node-id)))
                (:random (random max-node-id))
                (:local this-node-id))))
         [target-node-manager <= [:new class arguments . metaobj-annotation]
                              @ cont]
         [scheduler <= [:yield]]))
     ;; past type message transmission
     (is [:send-past target body reply-to . message-annotation]
         [[meta target] <= [:message body reply-to id . message-annotation]]
         [cont <= body])
     ;; now type message transmission
     (is [:send-now target body . message-annotation]
         [[meta target] <= [:message body cont id . message-annotation]]
         [scheduler <= [:yield]])
     ;; followings define base language constructs
     (is [:variable name . _]  !(lookup name env))
     (is [:constant value . _] !value)
     (is [:primitive operator arguments . _] !(apply operator arguments))
     (is [:if predicate then else . _]
       (let ((selected-form (if predicate then else))
             (object-executor [id <== [:executor]]))
         [object-executor <= [selected-form . context-annotation] @ cont]))
     ;; and so forth...
     )))]
```

Appendix B

Definitions of Dynamic Resource Management Systems

In this chapter, programs of the dynamic resource management systems discussed in Section 4.1 are presented.

B.1 Dynamic Load Balancing for Parallel Search

Node Manager The node manager has two node executors that correspond to two object distribution policies. On receiving a `:change-policy` message, which is issued by object `decision-maker` according to the global load-balancing status, the node manager changes the value of variable `executor`; this means the change of the object creation policy of that node. (From the viewpoint of implementation, this assignment causes changes of method tables of all objects in the node.)

```
[class node-manager-for-search (...)
 (state [distribute-executor := [new 'distribute-executor [new 'primary-executor]]]
        [balance-executor    := [new 'balance-executor    [new 'primary-executor]]]
        [executor := distribute-executor]
        [scheduler := [new 'scheduler-w-sentinel]]
        [generation := 0]
        target-nodes remote-rate)
 (script
  ;; The assignment to variable  executor changes the object creation policy of the node.
  (=> [:change-policy new-policy &optional targets remote-rate]
      (case new-policy
        (:distribute (setf executor distribute-executor))
        (:balance    (setf executor balance-executor
                           target-nodes targets
                           remote-rate remote-rate))))
  ;; Under the 'balance' policy, some of new objects are created at a
  ;; remote node, which is specified by object  decision-maker.
  (=> [:choose-target-node]
      (if (< (random) remote-rate)
        !nil ;; results in local creation
        !(select-one-node target-nodes)))
  (=> [:start-new-generation new-gen] @ load-reply-to
      where (= (1+ generation) new-gen)
```

```
      [scheduler <= [:report-load-value load-reply-to]]
      [scheduler <= [:put-sentinel new-gen]]
      (setf generation new-gen))
  (=> [:sentinel-found g] from scheduler where (= g generation)
      [decision-maker <= [:start-generation-update generation]]))]
```

Decision Maker The decision maker is an object that collects global status of load-values, and decides the next object creation policy based on the status. For 'balance' policy, it also decides where new tasks will be created by assigning a "target node list" for each node.

Since the process of global status collection is distributed, we can make good advantage of concurrent object-oriented language here. For example, whether *most nodes are starving for tasks* or not can be determined by a distributed computation at the collection process.

```
[class decision-maker ()
 (state [broadcaster := [new 'broadcaster  ... ]]
        [generation :=  0])
 (script
  (=> [:start-generation-update old-gen] where (= old-gen generation)
      (setf generation (1+ generation))
      [broadcaster
       <= [:broadcast-and-collect [:start-new-generation generation]
           :sum-up-function  ... ] @ Me])
  (=> [:collected-load-values <load-values>]
      (if ⟨most nodes are starving for task⟩
          [broadcaster <= [:broadcast [:change-policy :distribute]]]
          [broadcaster <= [:distribute ⟨target-nodes-lists⟩
                          :with [:change-policy :balance]]]))))]
```

Executors Following two executors embody two object creation policies: balance and distribute.

```
[class balance-executor ()
 (script
  (=> [:new class parameter annotation] @ cont
      (let* ((target-node [node-manager <== [:choose-remote-target]])
             (location (if target-node :remote :local)))
        [⟨class executor⟩ <= [:new class parameter
                              [:at location :target target-node . annotation]]
                    @ cont]))))]

[class distribute-executor ()
 (script
  (=> [:new class parameter annotation] @ cont
      (let ((location (if ⟨at some frequency⟩ :random :local)))
        [⟨class executor⟩ <= [:new class parameter [:at location . annotation]]
                    @ cont]))))]
```

B.2 Scheduling for Best-First Parallel Search

Metaobject A metaobject is created with additional keyword arguments `estimation` and `depth`. The additional scripts are to hold these two parameters, and to reply the values upon the query messages.

```
[class best-answer-metaobject ( ...  &key estimation depth)
 (state [estimation := estimation]
        [depth := depth]
```

```
        ⋮ )
 (script
  (=> [:estimation] !estimation)
  (=> [:depth] !depth)
   ⋮ )]
```

Class Executor The class executor has two scripts each of which defines interpretation of the object creation form and of the user-defined annotation. The third script delegates expressions in other types.

```
[class best-answer-executor ()
 (state [primary-executor :=  ... ])
 (script
  ;; object creation
  (=> [:do [:object-creation class parameters annotation] Id  ... ] @ cont
    (let ((depth (+ [Id <== [:depth]] 1)))
      [primary-executor
       <= [:do [:object-creation class parameters [:depth depth . annotation]]
               Id  ... ] @ cont]))
  ;; user-defined annotation
  (=> [:do [:found-answer value] Id  ... ] @ cont
    [node-manager <= [:found-answer value]]
    [cont <= nil])
  (=> any-other @ cont
    [primary-executor <= any-other @ cont]))]
```

Schedulers Each of schedulers used in the system has a small extension to the default one. The depth first scheduler uses a depth parameter held by metaobjects as a priority, and the best first scheduler uses an estimation value. In addition, the best first scheduler discards an execution request if its requester has a worse estimation than the value of the best answer found so far; by this, search tasks that have no chance to yield the best answer are pruned.

```
[class depth-first-scheduler (...)
 (state [active-object-queue := (make-priority-queue)])
 (script
        ⋮
    (let ((p [(thunk-id thunk) <== [:depth]]))
      (queue-put thunk active-object-queue :priority p))
        ⋮
 )]

[class best-first-scheduler (value)
 (state [active-object-queue := (make-priority-queue)]
        [best-so-far := value])
 (script
        ⋮
    (let ((p [(thunk-id thunk) <== [:estimation]]))
      (when (< best-so-far p)
        (queue-put thunk active-object-queue :priority p)))
        ⋮
  (=> [:update-best v] (setf best-so-far := v]))]
```

Node Manager The node manager has four scripts for scheduling management. Script `:found-answer` is invoked when an answer is found in the node. Script `:change-policy` is invoked when the (globally) first answer is found. Script `:update-global-best` is invoked when an answer that has a better value than the ones found so far. Script `:timer-event` is periodically invoked from the run-time system. This is for load balancing; on this event, a node simply sends a search task that is the most promising to one of the neighboring nodes.

```
[class node-manager ( ... )
 (state [scheduler := [new 'depth-first-scheduler]]
        [current-policy := 'depth-first]
        global-best
        [balancing-target-no := 0])
 (script
  (=> [:found-answer value]
    (cond ((eq current-policy 'depth-first)
           (broadcast [:change-policy value]))
          ((< global-best value)  ; better than ever found
           (broadcast [:update-global-best value]))))
  (=> [:change-policy value]
    (let ((new-scheduler [new 'best-first-scheduler value]))
      [scheduler <== [:replace-with new-scheduler]]
      (setf scheduler      new-scheduler
            global-best    value
            current-policy 'best-first)))
  (=> [:update-global-best value]
    (when (< global-best value)
      [scheduler <= [:update-best value]]
      (setf global-best value)))
  (=> [:timer-event]
    (let ((node-id (aref ⟨array of neighboring node IDs⟩
                         balancing-target-no))
          (task [scheduler <== [:get-a-task]]))
      (migration-request task :to node-id))
    (incf balancing-target-no)
    (set-timer-event ⟨some time span⟩)))]
```

B.3 Object Allocation for N-Body Simulation

Class Executor As a class executor specific to class `cell` shown in Figure 4.6, class `cell-executor` is defined. When an annotation {`:division` *ss sp*} is executed in the base-level program, the first script of the class-executor is invoked. It simply sends received parameters to a metaobject. The second script is for the object creation. When an object-creation form is executed at the base-level, this script is invoked. The cell executor firstly requests the 'node-list' of newly creating object from the metaobject. The new object is created at the first node in the specified node-list, with the node-list as an additional argument.

```
[class cell-executor ( ... )
 (script
  ;; for user defined annotation
  (=> [:do [:division subspaces subparticles] Id  ... ] @ cont
    [Id <= [:divide-assigned-nodes subspaces subparticles]]
    [cont <= t])
  ;; object creation
  (=> [:do [:new class parameters annotation] Id  ... ] @ cont
```

```
(let ((node-list [Id <== [:creation-node]]))
  [parent-executor
   <= [:do [:new class parameters
            [:at (first node-list) :nodes node-list . annotation]] Id  ... ]
   @ cont]))]
```

Metaobject The metaobject is created with keyword argument `node-list`. This list is divided according to the distribution of the particles, which is informed by the user defined annotation.

```
[class cell-metaobject ( ...  &key node-list)
 (state [assigned-nodes := node-list]
        divided-nodes)
 (script
  (=> [:divide-assigned-nodes subspaces subparticles]
    (setf divided-nodes
          (divide-nodes subparticles (length particle-list)
                        assigned-nodes)))
  (=> [:creation-node]
      !(pop divided-nodes)))]
```

B.4 Object Based Load Balancer

The definition of the load balancer has only one script. Note that the messages from the load balancers at neighboring nodes are received by `wait-for` form in the script.

```
[class load-balancer (s)
 (state [scheduler := s]
        [neighbor-load-list := '()]
        [neighbor-balancer-list := ⟨load balancer objects on neighboring nodes⟩]
        this-node-load)
 (script
  (=> [:start-balancing]
   ;; calculate a load of the node
   (let ((active-objects [scheduler <== [:list-active-objects]]))
     (setf this-node-load 0)
     (dolist (o active-objects)
       (incf this-node-load [o <== [:object-load]])))
   ;; tell the calculated load to neighboring load balancers
   (dolist (b neighbor-balancer-list)
     [b <= [:node-load this-node-load this-node-ID]])
   ;; collect loads from neighboring nodes into variable neighbor-load-list
   (dotimes (balancer neighbor-balancer-list)
     (wait-for
      (=> [:node-load load-value b] where (eq b balancer)
        (push [load-value b] neighbor-load-list))))
   ;; calculate the amount of loads to balance into  load-to-move
   (let* ((neighbor-load-sum
           (reduce #'+ (mapcar #'first neighbor-load-list)))
          (average-load
           (/ (+ this-node-load neighbor-load-sum)
              (1+ number-of-neighbors)))
          (load-to-move (* (- average-load this-node-load) factor))
          (move-load-list
           (mapcar #'(lambda (load&balancer)
                       (list (/ (* load-to-move (first load&balancer))
                                neighbor-load-sum)
                             (second load&balancer)))
                   neighbor-load-list)))
     ;; Here, the amount of load that should be moved to a neighboring node
```

```
;; is kept in  move-load-list
(when (< 0 load-to-move)
  (dolist (move-load move-load-list)
    (let ((moved 0))
      (while (< moved (first move-load))
        ;; Active objects are removed from the scheduler, and
        ;; requests for migration are issued until the amount of
        ;; migrating objects' load satisfies the calculated one.
        (let ((obj [scheduler <== [:get-active-object]]))
          (incf moved [obj <== [:object-load]])
          (migration-request obj :to (second move-load)))))))))]
```

Appendix C

Interface Definitions to Low-Level Systems

Object Migration Following primitive functions are available for object migration:

- `(migration-request` *object* `:to` *target node*`)`
  This form requests that *object* will be migrated to *target node*. Note that the completion of this form does not mean the completion of migration. This is because some migration mechanism suspends the request (for example, until the global garbage-collection occurs) to avoid overhead.

- `(migration-wait-completion)`
  This form waits the completion of the requests issued by above form.

- `(migration-now` *object* `:to` *target node*`)`
  This form requests an object migration and waits the completion of the request. That is, it is a shorthand of:

```
(progn (migration-request object :to target node)
       (migration-wait-completion))
```

The completion of migration is also notified to the node managers as messages:

`[:emigrated-objects ((`*object* `.` *destination node*`) ...)]`

and

`[:immigrated-objects ((`*object* `.` *source node*`) ...)]`.

The first message—which is sent to a node manager—means that *object* emigrates from the node to *target node*. The second one means that *object* immigrated from *source node* into the node.

System Level Information Several system level information can be obtained via primitive calls:

- `(free-memory)`
  A ratio of free memory of a node is returned.

- `(income-network-messages)`
  Number of incoming network messages is returned.
- `(outgo-network-messages)`
  Number of outgoing network messages is returned.
- `(parent-node` ⟨*node-ID*⟩ *n*`)`
  ID of the parent node in a n-ary tree is returned.
- `(children-nodes` ⟨*node-ID*⟩ *n*`)`
  IDs of children nodes in a n-ary tree is returned.
- `(mesh-address (`i_1 ... i_n`) (`d_1 ... d_n`))`
  ID of a node whose address is $(i_1, \ldots, i_n)$ in a $d_1 \times d_2 \times \cdots \times d_n$-ary mesh is returned.

As for the primitives related to network topology, we are planning to provide abstract data/class so that we can use in more flexible and convenient way.