Distributed High-Level Module Binding for Flexible Encapsulation and Fast Inter-Modular Optimization

Christian S. Collberg

Department of Computer Science,* University of Auckland,
Private Bag 92019, Auckland, New Zealand
c_collberg@cs.auckland.ac.nz

Abstract. We present a new modular object-oriented language with orthogonal encapsulation facilities. The language provides full support for encapsulation and separate compilation which makes it difficult to compile using standard techniques. We present new distributed translating techniques which overcome these difficulties by allowing inter-modular information to be exchanged at link-time. The same techniques may also be used with other modular and object-oriented languages to facilitate fast inter-modular optimizations such as inline expansion.

1 Introduction

Zuse is a new modular object-oriented language with orthogonal encapsulation facilities: *concrete*, *abstract*, and *semi-abstract* export of types, constants, and (inline) procedures are all supported. Zuse's encapsulation and separate compilation facilities make the language more difficult to compile than other similar languages (such as Ada [21], Mesa [10], Modula-2 [23], and Modula-3 [17]): inter-modular information necessary for memory allocation, semantic analysis, and code generation is not always present at compile time. The Zuse translating system therefore exchanges inter-modular information at link-time. The module binder (called a Paster) binds modules together partially at the intermediate code level, performs inter-modular optimizations such as inline expansion, allocates memory, and generates code for procedures for which code generation could not be performed at compile-time.

In addition to the description of the Zuse language itself, the main contribution of this paper is the design of a distributed module binder (the dPaster) which achieves better performance than its sequential counterpart (the sPaster) by distributing its actions over the sites of a workstation-server network. The design of the Zuse compiler and dPaster assures that the cost in terms of

* This work was carried out while the author was at the Department of Computer Science, Lund University, Sweden.

translation-time, execution-time, and storage of using an abstract or semi-abstract item will be no greater than if the same item had been concrete. Although the dPaster is designed specifically to support the encapsulation facilities of the Zuse language, it can also be used with other modular and object-oriented languages to facilitate fast inter-modular optimizations.

Considering that all other aspects of language translation have been subject to attempts at parallelization,[2] it is remarkable that this appears to be the first attempt at parallelizing the linking process. It is also the first time distributed techniques have been used to tackle the problems inherent in combining separate compilation with both data abstraction and inter-modular optimization.

2 Background and Motivation

A desirable property of a modular language is that it should allow a module interface to be *fully abstract*, i.e. to contain only the information necessary in order for a client to correctly use the module and for an implementer to correctly implement it. This principle, apart from being basic to sound software engineering practices, is also essential in order to minimize compilation cost. If an interface is not fully abstract, containing extraneous information related more to the module's *implementation* than to its *specification*, any change to this information will affect the interface and hence will trigger the recompilation of the module's clients.

There is, however, an inherent conflict between the two goals of permitting fully abstract module interfaces and separate compilation of interfaces and bodies. The conflict stems from the fact that in many cases an abstract interface will not contain enough information in order for a compiler to assure the static semantic correctness of the module's clients or to generate code for the clients. The conflict is evident in the design of many current modular languages. Ada, for example, requires the implementation of private types to be given in the module interface, in order for the compiler to know the size of the type when compiling a client module. Modula-2, for the same reason, restricts its opaque types to pointers, while Mesa and Oberon-1 [24] require the *size* of opaque types and *public projection* types, respectively, to be given in the interface. Modula-3 instead relies on run-time processing in order to support its opaque and partially revealed object types, as does Albericht [18] – an Oberon descendant – to support its public projection type. Milano-Pascal [5], finally, employs a specialized module linker to share information between separately compiled Pascal modules, specifically sizes of abstract types and values of abstract constants.

It is not only the implementation of abstraction which is hampered by separate compilation but also inter-modular optimizations such as inline expansion and inter-procedural register allocation. C++ [1] and Mesa, for example, require inline procedures to be declared in the module specification, while Ada allows

[2] See, for example, Junkin [13] (compilation), Katseff [14] (assembly), Baalbergen [2] (making).

compilation dependencies between implementation units containing exported in-line procedures. The MIPS compiler-suite [11] solves the inter-modular inlining problem by concatenating the intermediate code of separately compiled modules prior to code-generation and optimization. Similarly, Wall [22] employs a specialized linker in order to do inter-modular register-allocation. While these implementations have some obvious advantages (good global code quality, no unnecessary recompilations) they suffer from the problem of excessive linking times. In this paper we will show that techniques from the field of *distributed compilation* allow full data encapsulation, separate compilation, and cross-module optimization to be achieved without undue translation-time or execution-time overhead.

3 The Language Zuse

Zuse contains the array of basic concepts found in most modular procedural and object-oriented languages and shares with these languages a common view of data and execution. We will focus our attention on the kinds of items Zuse modules may export and the facilities available to a programmer for protecting the integrity of such items.

Zuse borrows its module system from Modula-2: a module is made up of two separately compiled units, a *specification* and an *implementation*. Compilation dependencies generally exist between specification units and their clients, but Zuse (like Modula-2) does not allow dependencies between implementation units.

Zuse shares most of its basic type system with Modula-2. Also supported is a Modula-3-style object type which forms the basis of object-oriented programming, Mesa-style type initializers, structured manifest constants, and inline procedures. Unique to Zuse are its three *modes of export*: a *concrete* item reveals its entire realization in the specification unit, a *semi-abstract* item reveals part of its realization and hides the rest in the implementation unit, and an *abstract* item hides all of its realization in the implementation unit.

3.1 Abstract export

Figure 1 shows a simple complex number package using abstract export. A similar package in Ada would require the realization of the exported type to be revealed in the specification unit. This, apart from being a breach of encapsulation, would require the recompilation of all client modules in the event of a change to the realization. Even worse, since Complex`Create[3] is an inline procedure, most Ada implementations would insert compilation dependencies between the implementation unit of Complex and all of its clients, forcing all clients (and possibly their clients) to be recompiled whenever the implementation unit was changed.

[3] M`t is a reference to the object t exported from module M.

```
SPECIFICATION Complex;                IMPLEMENTATION Complex;
  TYPE T = ;                            TYPE T += RECORD [Re, Im : REAL ];
  CONSTANT Zero : T =;                  CONSTANT Zero : T += RECORD [
  PROCEDURE Create : (                                          Re :== 0.0 ;
     Re, Im : REAL) RETURN T =;                                 Im :== 0.0 ];
END Complex.                            INLINE PROCEDURE Create : (
                                           Re, Im : REAL) RETURN T +=
                                        BEGIN ··· END Create;
                                      END Complex.
```

Fig. 1. Examples of ZUSE abstract export.

A Modula-2 or Modula-3 implementation, on the other hand, while not revealing the realization of the abstract type[4] or introducing any unwarranted compilation dependencies, would be forced to use a pointer-based realization of Complex`T. This has several disadvantages: (1) Every time a complex number is needed a call has to be made to the dynamic allocation routine, which would be expensive for, say, large arrays of complex numbers; (2) when using the built-in assignment operator on variables of Complex`T, the effect is *reference assignment* rather than *value assignment*, the preferred mode for complex values; (3) in Modula-2, which does not have garbage collection, temporary complex variables may never be reclaimed (see Feldman [9, pp. 127-131]).

Hence, ZUSE makes it possible to combine the static allocation, value semantics, and inline expansion of Ada, with the full data encapsulation and compilation independence of Modula-2 and Modula-3.

3.2 Semi-Abstract Export

It may sometimes be desirable to make part of the realization of an exported item known to its clients while keeping the rest private to the implementation. Oberon-1 and Modula-3 provide limited forms of such *semi-abstract* types, but at a cost: In Oberon-1 it is necessary to reveal the size of the full type in the specification unit [24], and in Modula-3 the use of partially revealed object types entails extra run-time analysis. ZUSE, on the other hand, has full support for semi-abstract types, constants, and inline procedures, and the ZUSE translation system assures that this will not involve any extra translation-time or run-time cost. Figure 2 gives some simple examples of ZUSE's semi-abstract export. Note that the ZUSE object type comes in three parts: an optional supertype (preceded by the keyword **WITH**), a sequence of fields (the first set of brackets), and a sequence of methods (second set of brackets).

[4] Modula-2 and Modula-3 do not support ZUSE-style abstract constants and inline procedures.

```
SPECIFICATION SemiAbstract;
  TYPE
     EnumT   = ENUM [Red, Blue, Green];
     RecordT = RECORD [x : CHAR ];
     ObjT    = OBJECT;
     SubObjT= OBJECT
                   [v  : INTEGER ]
                   [R : (x : CHAR)] ;
  CONSTANT
     C : RecordT = RECORD [x :== "R"];
  NOT INLINE PROCEDURE P : (a : EnumT) =;
END SemiAbstract.

IMPLEMENTATION SemiAbstract;
  TYPE
     EnumT   += ENUM [Yellow, Orange];
     RecordT += RECORD [z : CARDINAL ];
     ObjT    += OBJECT [a : INTEGER] [ ];
     SubObjT+= WITH ObjT OBJECT
                    [r  : RecordT ]
                    [U : (z : CHAR)];
  CONSTANT
     C : RecordT += RECORD [z :== 15];
  PROCEDURE P : (a : EnumT) +=
  BEGIN ··· END P;
END SemiAbstract.
```

Fig. 2. Examples of ZUSE semi-abstract export. The module SemiAbstract reveals that the type RecordT is a record with a character field x, but hides the fact that RecordT also has an integer field z. SemiAbstract furthermore reveals that SubObjT is an object type with a field v and a method R, but hides the fact that SubObjT is a subtype of ObjT and declares an additional field r and a method U.

4 The ZUSE Compiler

A consequence of ZUSE's scheme for separate compilation and flexible encapsulation is that the ZUSE compiler does not always have available all the intermodular information necessary in order to completely check the static semantic correctness of the input program and to generate high-quality code. When the compiler determines that this is the case for a particular procedure, code generation and (some of the) semantic checking are deferred to module binding time when all relevant information is available. We list some of the circumstances under which such deferral will be necessary:

– The compiler will not always know the size of a variable which is defined in terms of imported abstract or semi-abstract types or constants. Hence it

will be unable to generate code for any routine which allocates or references such a variable.
– The compiler cannot generate code for a call to an imported abstract procedure since it may be inline, and in this case the code of the procedure is unavailable for expansion.
– The compiler cannot construct the method template for an object type which has a semi-abstract ancestor.
– The compiler cannot always check whether the declaration of an abstract type, constant, or inline procedure which is defined in terms of imported abstract items is part of an illegal recursive declaration (see Figure 3).

```
SPECIFICATION M;          IMPLEMENTATION M;
  TYPE T = ;                IMPORT N;
  PROCEDURE P : () =;       TYPE T += N`T;
END M.                      INLINE PROCEDURE P : () += BEGIN N`P (); END P;
                          END M.

SPECIFICATION N;          IMPLEMENTATION N;
  TYPE T = ;                IMPORT M;
  PROCEDURE P : () =;       TYPE T += RECORD [x,y : M`T ];
END N.                      INLINE PROCEDURE P : () += BEGIN M`P (); END P;
                          END N.
```

Fig. 3. Examples of illegal recursive declarations in ZUSE.

Deferred procedures and context conditions produced by the compiler for a certain module are stored in the module's object code file together with exported inline procedures, binary code produced for non-deferred procedures, etc. The object code file for a module M consists of seven major sections. The *Module Table* is a list of the names of the modules directly imported by M. It is used to determine the modules which will be part of the resulting program. The *Expression Section* is a set of constant expressions (in a DAG format) which organizes all the inter-modular information needed during binding. It stores, among other things, sizes of types, possible static error conditions, the allocation of global variables, and the inline-status and call graphs of procedures. The *Binary Code and Data Sections* contain code and data generated during compilation, and the *Relocation Section* contains relocation information for this code. The *Inline* and *Deferred* sections contain the intermediate code (in an *Attributed Linear Code* format) of inline and deferred procedures.

4.1 The Expression Section

The Expression Section is the central object code structure. The general structure of an expression is $e_{m,i}.a = f(e_{m_1,i_1}.a_1, \cdots, e_{m_n,i_n}.a_n)$, where $e_{m,i}.a$ is the attribute a of the i:th expression in module m, f is a function with n parameters, and the $e_{m_k,i_k}.a_k$:s are references to the values of other expressions. The function f may be an arithmetic operation, an operation to construct structured constants or method templates from their subparts, etc. It is sometimes convenient to think of the Expression Section as a graph where each $e_{m,i}.a$ is a node labeled with the operation f with n outgoing edges to the nodes $e_{m_1,i_1}.a_1, \cdots, e_{m_n,i_n}.a_n$. In most cases the graph will be a forest of expression DAGs, but the following example (the concatenation of the Expression Sections of the modules M and N which were given earlier) shows that the graph may sometimes contain cycles:

$$\begin{array}{lll}
e_{\mathrm{M},1}.size & = e_{\mathrm{N},2}.size & \text{-- M`T} \\
e_{\mathrm{M},2}.inline & = \textbf{TRUE} & \text{-- M`P} \\
e_{\mathrm{M},2}.calls & = [e_{\mathrm{N},3}] & \text{-- M`P} \\
\hline
e_{\mathrm{N},1}.expr & = 2 & \\
e_{\mathrm{N},2}.size & = e_{\mathrm{N},1}.expr * e_{\mathrm{M},1}.size & \text{-- N`T} \\
e_{\mathrm{N},3}.inline & = \textbf{TRUE} & \text{-- N`P} \\
e_{\mathrm{N},3}.calls & = [e_{\mathrm{M},2}] & \text{-- N`P}
\end{array}$$

The illegal declarations of M and N are evident from these expressions: the recursive definitions of M`T and N`T correspond to a cycle between the expressions which express the types' size, and the recursive inline calls can be discerned from the call-graph of the procedures M`P and N`P.

5 The Zuse Sequential Module Binder

In many ways the Zuse sequential binder resembles ordinary systems link editors. They both combine the code produced by a compiler for separately compiled modules into an executable file, and they both run in several phases, the first phase loading definitions and the last phase performing relocations. However, unlike link editors the sPaster is equipped to perform code generation, intermodular optimization, and some static semantic checking. Our present sPaster design runs in four major phases (see Collberg [6] for a more detailed description):

- Phase 1 determines which modules will make up the resulting program, loads the expressions and inline procedures of all modules, and copies the binary code and data sections to the resulting executable file.
- Phase 2 evaluates the expressions, allocates global variables, performs deferred semantic checks, and expands inline calls in inline procedures. Phase 2 also makes a conservative estimate of which deferred procedures may be referenced at run-time.
- Phase 3 reads the intermediate code of every referenced deferred procedure, updates the code with the expression values calculated during Phase 2, expands inline calls, optimizes the intermediate code, generates machine code, and writes the generated code to the executable file.
- Phase 4 performs final relocation.

6 The Zuse Distributed Module Binder

The high-level binding required by the Zuse language can be a considerably more expensive operation than conventional link-editing, but fortunately it is a process which is amenable to parallelization. This section will describe the design of the Zuse distributed module binder which distributes its actions over a workstation-server network, or similar loosely-coupled architectures.

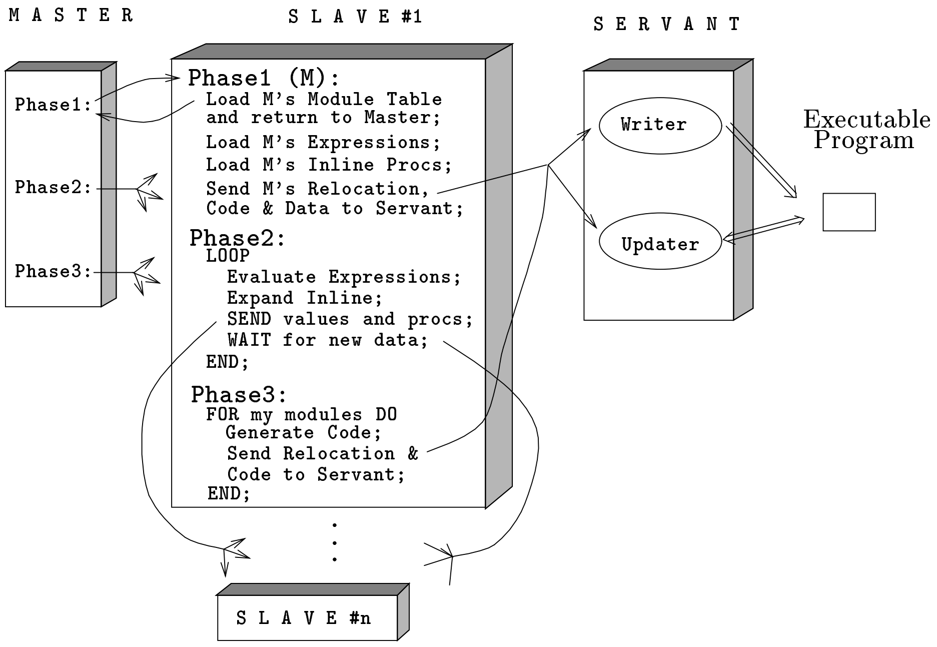

Fig. 4. An overview of the dPaster.

The dPaster can be said to be based on the *data decomposition*[5] model of compiler parallelization, in that each site taking part in the build process is responsible for a subset of the modules which make up the program. The dPaster is made up of three kinds of processes: a *Master*, a *Servant*, and a set of *Slaves* (see the figure above). The Master is the main conductor of processing and is the program invoked by the user. It decides which modules each Slave should process, initiates the various processing phases, and relays any errors back to the user. The Slaves (there is at most one per site) are dormant until awakened by the Master, who assigns them a set of modules to process. The Slaves exchange information about the modules they have been assigned, optimize and generate

[5] See Khanna [15] for a taxonomy of approaches to parallel compilation.

code for their modules, and send the generated machine code and relocation information to the Servant, who (concurrently with the Slave's code generation) performs relocation. Once finished, the Slaves and the Servant return to their dormant state until next contacted by a Master.

The dPaster runs in four main phases (see Figure 4), where the beginning of each phase serves as the primary *point of synchronization*; no Slave starts a phase until all other Slaves have finished the previous one. The rest of this section will present a detailed description of the dPaster, one phase at a time. The algorithms presented assume that inter-process communication is realized by a Birrell and Nelson-type [3] remote procedure call (RPC) protocol, extended with facilities for multicast calls.

6.1 Phase 0

The initial phase awakens the Slaves and the Servant and relays to them global data pertinent to the session, including command line switches, environment variables, object file search paths, etc.

6.2 Phase 1

Finding the modules which are going to be part of the final program entails finding the closure of the module import graph with respect to the main module. During Phase 1 the Master repeatedly selects a module which has yet to be processed (starting with the main program module) and assigns it to a Slave. The assignment is based on an estimated processing time of each module, in an attempt to give the Slaves similar workloads. The Slave loads the module's Module Table, Expression Section, and Inline Section, and passes on the Binary Code, Data, and Relocation sections to the Servant. Finally, the Module Table is sent back to the Master, who uses it to find new unprocessed modules. The process terminates when all modules reachable from the main module have been processed.

6.3 Phase 2

During Phase 2 the Slaves cooperate in order to accomplish three tasks:

1. Evaluate all expressions and replicate the values at the Slaves who will need them during Phase 3.
2. Check deferred context conditions and report any violations back to the Master.
3. Expand inline calls in inline procedures and replicate the resulting code at the Slaves who will need it during Phase 3.

In the following we will concentrate on algorithms for expression evaluation (point 1 above), since the algorithms for point 2 and 3 are simple variants.

At the start of Phase 2 the expression graph has been partitioned (as a result of Phase 1) into s parts, one part for each Slave. The edges of the graph are either *internal* to the Slaves, connecting nodes residing on the same site, or *external*, connecting nodes on different sites. While the graph will most often be acyclic, illegal recursive declarations do introduce cycles.

Distributed expression evaluation can be approached in several different ways. The most intuitive method may be for each Slave to evaluate local subgraphs depth-first, and to retrieve the value of an external node by querying the Slave which owns the node. This method has three problems: (1) Each Slave has to know the owner of every node in the graph; (2) Cycles in the graph can lead to deadlock situations which either have to be avoided or detected; (3) The message complexity can be prohibitive (the number of messages is proportional to the number of external edges in the graph, which in turn grows with the number of Slaves). To reduce the message complexity we use a different algorithm which makes heavy use of multicast calls. It is a conceptually simple iterative method where each Slave evaluates as much as it can, distributes the new values to the other Slaves, and waits for the arrival of new values. The number of messages for this algorithm becomes proportional to the *height* of the graph formed by the external edges in the expression graph, rather than to the number of edges themselves, as in the previous algorithm. Our algorithm assumes the existence of a set of auxiliary boolean attributes of Table 1.

Attribute	Semantics	Initial Value
$e_{m,i}.\mathit{known}$	**TRUE** on Slave S_k if $e_{m,i}$ has a known value.	**TRUE** if $e_{m,i}$ was computed at compile-time.
$e_{m,i}.\mathit{global}$	**TRUE** if $e_{m,i}$ represents an exported entity.	Set at compile-time.
$e_{m,i}.\mathit{referenced}$	**TRUE** if $e_{m,i}$ represents a procedure reachable from the main module.	**TRUE** for main module bodies.

Table 1. Auxiliary attributes used by the expression evaluation algorithm.

The $e_{m,i}.\mathit{referenced}$ attribute is true for procedures which may actually be called at run-time. The $e_{m,i}.\mathit{global}$ attribute is used to avoid having to replicate expression values which are local to a module. During Phase 2 each Slave repeatedly executes the following algorithm:

1. Evaluate as many expressions as possible, i.e. all expressions whose arguments are known, and let E_{out} be the subset of these expressions which are global.
2. Compute R_{out}, the set of all global procedures $e_{m,i}$ reachable from other reachable procedures. I.e., $e_{m,i}$ is in R_{out}, iff for some $e_{n,j}$, $e_{m,i} \in e_{n,j}.\mathit{calls}$

and $e_{n,j}$.*referenced*=**TRUE**.
3. Multicast (E_{out}, R_{out}) to all other Slaves.
4. Wait for new incoming values. When a tuple (E_{in}, R_{in}) arrives from another Slave, store the incoming values of E_{in}, set $e_{n,j}$.*referenced*=**TRUE** for all $e_{n,j}$ in R_{in}, and set $e_{n,j}$.*known*=**TRUE** for all $e_{m,i}$ in E_{in} and R_{in}.

The algorithm terminates when no Slave can perform any more evaluations. There are two possible reasons for this: either all expressions have been evaluated, or there exists a cycle among the expressions. To differentiate between these situations, the Master employs a distributed termination detection algorithm which alerts it when all Slaves are idle. If at that point some Slaves still hold unevaluated expressions, the Master will know that the expressions contain a cycle and can issue the appropriate error message. The termination detection algorithm can be made particularly simple since all communication is in the form of multicast remote procedure calls:

1. A Slave is either *active* or *passive*. A Slave may enter a passive state only after having determined that all remote calls it has initiated have returned, all incoming calls have been returned, and that no further local processing is possible.
2. Let each Slave S_i maintain a count C_i of all multicast messages sent or received during the execution of the underlying algorithm.
3. Let the Master send a multicast remote procedure call **DETECT** to $S_1, \ldots, S_s$. On receipt of a **DETECT** call Slave S_i continues processing until a passive state is reached. At this point a reply is sent to the Master including the current value of C_i.
4. If all Slaves return the same value C, the Master can conclude that the underlying algorithm has terminated. Otherwise, the process is repeated with a new **DETECT** call.

Theorem 1. *The algorithm above detects termination iff the underlying computation has terminated.*

Proof. See [6, pp. 184–186]

The inline expansion part of Phase 2 uses an algorithm similar to the one just described for expression evaluation. One difference is that inline procedures (which may be large and expensive to transmit and store) are only replicated at the Slaves which actually need them, whereas all global expression values are always replicated at all Slaves.

The problem of distributed expression evaluation arises in two other areas related to language implementation, namely parallel attribute evaluation [16] and parallel evaluation of functional programs [12]. Since in general the expressions to be evaluated are known prior to evaluation, research in these areas emphasizes the development of algorithms which compute the optimal expression-to-site mapping. In our case, however, little is known about the structure of the expression graph before evaluation, and hence the expression-to-site mapping has to be done more or less at random. Our algorithm for expression evaluation is also reminiscent of *replicated blackboards* used in AI systems [7] and the parallel topological sorting algorithm of Er [8].

6.4 Phase 3

After Phase 2 all Slaves have local access to any inter-modular information they may need during Phase 3. This phase is therefore uncomplicated. Each Slave executes the following algorithm for each of the modules it owns:

1. Read the intermediate code of all referenced deferred procedures.
2. Update the code with the computed expression values and expand calls to inline procedures.
3. Optimize and generate machine code.
4. Send the code, relocation information, and new addresses of generated procedures to the Servant.

6.5 The Servant

It would be elegant if the Slaves could write their generated code directly to a shared global file. However, client file caching in network file systems makes concurrent access to global files problematic. Having a dedicated process handle all accesses to the executable file avoids this problem, and also makes it possible to perform relocations concurrently with code generation. In the current implementation the Servant process receives generated code and data, relocation information, and addresses of procedures and variables from the Slaves. Two lightweight processes (*Writer* and *Updater*) within the Servant operate concurrently on these data: *Writer* writes the code and data to the executable file, and *Updater* updates the file as relocation information and procedure addresses become available.

7 Evaluation

The current implementation of the ZUSE translation system runs on and produces code for Sun 3 workstations. For programs with a large amount of encapsulation and inlining the DPASTER achieves a speedup factor between 2.6 and 3.5.

Figure 5 shows the execution times (clock-time in number of seconds) for the SPASTER and the DPASTER, when binding five versions of the SPASTER itself. Table 2 gives the amount of encapsulation and inlining in each version. As a comparison the timing for the SUN Modula-2 linker *m2l* (a pre-linker which calls the UNIX systems linker *ld* as part of the linking process) for the same program is also given. Figure 5 also shows the results for four large artificially generated programs. Figure 6 shows how the speedup of the DPASTER varies with the amount of encapsulation and inlining.

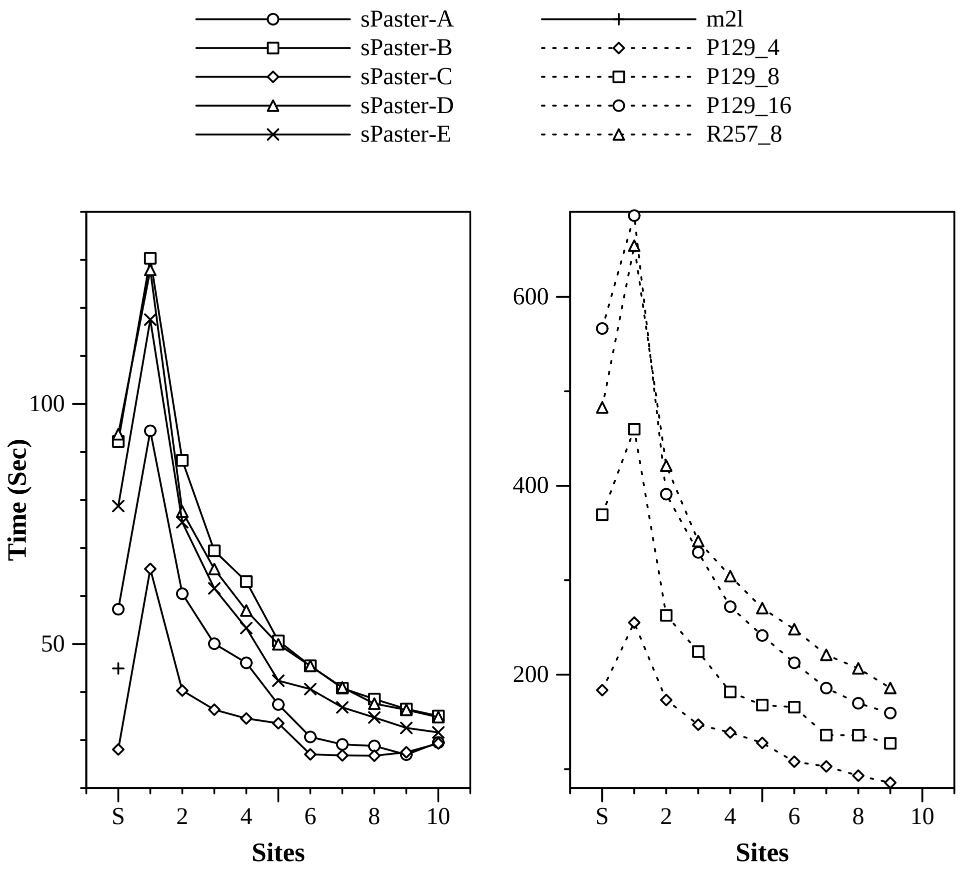

Fig. 5. On the left is shown the running time of the sPASTER (point S) and the DPASTER running on 1–10 sites when binding 5 versions of the sPASTER itself. The timing for the SUN Modula-2 linker `m2l` linking the sPASTER is also given. To the right is shown the total running time of the sPASTER and the DPASTER when binding 4 artificial programs.

8 Conclusions and Future Work

The main contributions of the ZUSE language and translating system is summarized by the following points:

Regular and flexible encapsulation ZUSE is more uniform than other similar languages in its ability to hide or reveal any part of any exported item.

Less dynamic allocation Since ZUSE's abstract types can be allocated statically there is less stress on the dynamic allocation system. Statically allocated abstract types also make it easy (compared to Modula-2 and Modula-3 which

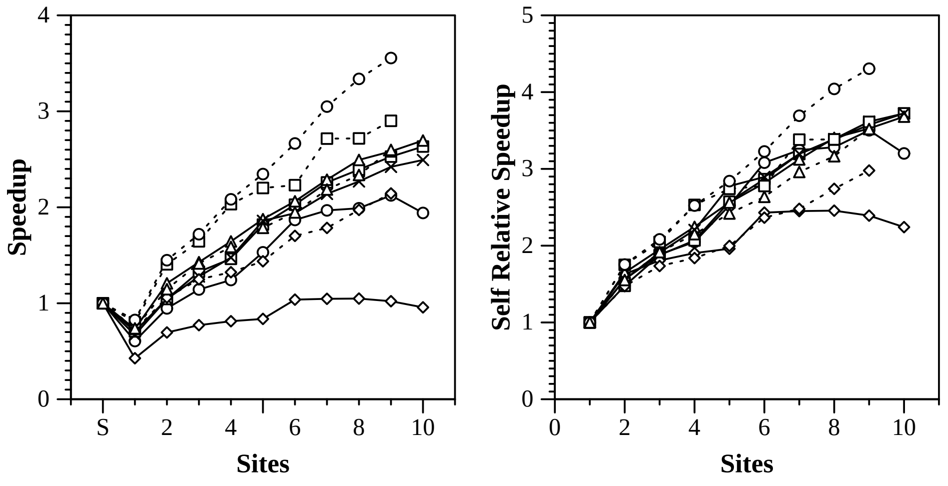

Fig. 6. Speedup of the dPASTER relative the sPASTER (left), and self-relative speedup of the dPASTER (right).

PROGRAM	M	LOC	S	P_t	T_a	C_a	I_a	I_2	I_3	I_c	P_d	S_d	E
sPaster-A	108	36	14	2024	28	34	0	0	0	0%	879	5	533
sPaster-B	108	36	14	2024	28	33	431	41	1325	55.6%	953	15	578
sPaster-C	108	36	14	2024	0	0	0	0	0	0%	0	0	55
sPaster-D	108	36	14	2024	0	0	431	41	1325	55.6%	760	10	99
sPaster-E	108	36	14	2024	16	27	202	10	724	34.8%	1024	15	497
P129_4	129	51	62	1905	471	448	653	133	1815	N/A	986	0	7542
P129_8	129	55	64	1905	471	448	653	268	3101	N/A	1129	0	8645
P129_16	129	56	64	1905	471	448	653	361	4040	N/A	1222	0	8360
P257_8	257	167	203	5158	203	800	775	647	2907	N/A	3889	0	13629

Table 2. Source code statistics regarding 5 versions of the sPASTER and 4 artificially generated programs. Abbreviations: M = Number of modules; LOC = Total size in thousands of lines of code. S = Total number of statements (in thousands). P_t = Total number of procedures. T_a, C_a, I_a = Number of abstract types, constants, and inline procedures; I_2, I_3 = Number of inline expansions performed during Phase 2 and 3; I_c = Percentage of dynamic calls expanded, for typical input. P_d = Number of referenced deferred procedures. S_d = Number of deferred context conditions. E = Number of constant expressions, excepting call-graph attributes.

only allow dynamically allocated abstract types) to implement abstract data types with value semantics.

No unnecessary recompilations Unlike Ada and C++, ZUSE's encapsulation and inter-modular inlining facilities do not create any unwarranted compilation dependencies, and hence will not generate any unnecessary recompilations.

Fast inter-modular optimization ZUSE's distributed module binder provides fast turn-around times even for large programs with much encapsulation and many inline calls.

No unwarranted run-time processing Unlike Modula-3 method lookup, object-type field accesses, and method template construction in ZUSE do not entail any extra run-time processing.

The inter-modular information discussed in this paper is of two different kinds: that which results from the use of encapsulation in modular and object-oriented programming languages and that which results from a desire to perform inter-modular optimizations. The distributed module binder presented here represents a framework in which both kinds of information can be computed at link-time, with little or no apparent extra translation-time cost compared to conventional link editing.

Our present work focuses on the use of better load balancing algorithms in the distributed binder. We are also investigating whether inter-procedural optimization techniques [20] other than inlining (such as inter-procedural register allocation [22] and constant propagation [4]) can be incorporated into our distributed binder. We are furthermore considering applying incremental techniques [19] to the distributed binder to further speed up binding when only small and local changes have been made between two consecutive binds.

References

1. Accredited Standards Committee X3, Information Processing Systems, The American National Standards Institute (ANSI), CBEMA, 311 First St. NW, Suite 500, Washington, DC 20001. *Draft Proposed American National Standard for Information Systems — Programming Language C++*, x3j16/91-0115 edition, September 1991.
2. Erik H. Baalbergen. Design and implementation of parallel make. *Computing Systems*, 1:135–158, 1988.
3. Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
4. David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium On Compiler Construction*. ACM, June 1986.
5. A. Celentano, P. Della Vigna, C. Ghezzi, and D. Mandrioli. Separate compilation and partial specification in Pascal. *IEEE Transactions on Software Engineering*, 6:320–328, July 1980.
6. Christian S. Collberg. *Flexible Encapsulation.* PhD thesis, Lund University, December 1992.

7. R. S. Engelmore and A. J. Morgan. *Blackboard Systems*, chapter 30. Addison-Wesley, 1988. ISBN 0-201-17431-6.
8. M. C. Er. A parallel computation approach to topological sorting. *The Computer Journal*, 26(4):293–295, 1983.
9. Michael B. Feldman. *Data Structures with Modula-2.* Prentice-Hall, 1988. ISBN 0-13-197666-4.
10. Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite. Early experience with Mesa. *CACM*, 20(8):540–553, August 1977.
11. Mark Himelstein, Fred C. Chow, and Kevin Enderby. Cross-module optimizations: Its implementation and benefits. In *Proceedings of the Summer 1987 USENIX Conference*, pages 347–356, June 1987.
12. Simon L. Peyton Jones. *Parallel Graph Reduction*, chapter 24. Prentice-Hall, 1987. ISBN 0-13-453325-9.
13. Michael D. Junkin and David B. Wortman. The implementation of a concurrent compiler. Technical Report CSRI-235, Computer Systems Research Institute. University of Toronto, December 1990.
14. Howard P. Katseff. Using data partitioning to implement a parallel assembler. *SIGPLAN Notices*, 23(9):66–76, September 1988. ACM/SIGPLAN Parallel Programming: Experience with Applications, Languages, and Systems.
15. S. Khanna and A. Ghafoor. A data partitioning technique for parallel compilation. In *Proceedings of the Workshop on Parallel Compilation*, Kingston, Ontario, Canada, May 1990.
16. Eduard F. Klein. Attribute evaluation in parallel. In *Proceedings of the Workshop on Parallel Compilation*, Kingston, Ontario, Canada, May 1990.
17. Greg Nelson. *Systems Programming with Modula-3.* Prentice Hall, 1991. ISBN 0-13-590464-1.
18. Jukka Paakki, Anssi Karhinen, and Tomi Silander. Orthogonal type extensions and reductions. *SIGPLAN Notices*, 25(7):28–38, July 1990.
19. Russel W. Quong. *The Design and Implementation of an Incremental Linker.* PhD thesis, Stanford University, May 1989.
20. Stephen Richardson and Mahadevan Ganapathi. Code optimization across procedures. *Computer*, pages 42–49, February 1989.
21. M. W. Rogers, editor. *Ada: Language, Compilers and Bibliography.* Cambridge, 1984. ISBN 0-521-26464-2.
22. David W. Wall. Experience with a software-defined machine architecture. *ACM Transactions on Programming Languages and Systems*, 14(3):299–338, July 1992.
23. Niklaus Wirth. *Programming in Modula-2.* Springer Verlag, second edition, 1983.
24. Niklaus Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988.

This article was processed using the LaTeX macro package with LLNCS style