Architectural Support for Efficient Execution of Reusable Software Components *

Lonnie R. Welch and Bruce W. Weide
Dept. of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210 †

Abstract

This paper considers architectural features that permit the efficient implementation of a class of reusable software components. The approach is to develop a virtual machine with an instruction set suited to the efficient execution of the class of components. A sequential implementation of the virtual machine is used as a processing element in a distributed memory implementation of it (the virtual machine). The distributed memory computer has a model of parallel execution for which explicit parallelism need not be described by the programmer; furthermore, the model exploits a kind of parallelism that is unique to the class of software components considered.

1 Introduction

As pointed out in [1], reusable software components have a reputation for being inefficient. For this reason, a programmer concerned with efficient execution of software may feel tempted to avoid reusable software. However, a software engineer concerned with controlling development time and maintenance costs would prefer to reuse software components. This paper considers barriers to the efficient execution of reusable software components, so the benefit of increased programmer productivity need not be offset by decreased program execution speed.

The kind of software development considered here is the reuse of modules exporting abstract data types (ADTs). Programs are assumed to be written in RESOLVE [10], a language for producing verifiable and efficient reusable software. The language provides no built-in types—every type is exported by a reusable module; modules are generic, as in Ada; and variables are automatically initialised and finalised, as in C++.

Our approach to improving the execution speed of such programs is as follows: (1) identify potential inefficiencies of such software; (2) develop a model of parallel execution suited to the class of reusable components considered; and (3) design an architecture that addresses the potential inefficiencies of reusable software and supports the model of parallel execution developed.

For the implementation of RESOLVE, we chose to design a RISC-like virtual machine that can be realised efficiently in silicon, and that provides an instruction set suited to the efficient execution of RESOLVE programs. We call the virtual machine ARC, an acronym meaning "Architecture for Reusable Components." Although a compiler generates apparently sequential code for ARC, that code will work on sequential or parallel (distributed) implementations of ARC. If the object code is executed on a parallel machine, the hardware will detect parallelism. A sequential implementation of ARC is used as the processing element (PE) in the parallel implementation of ARC presented in this paper.

Section 2 gives an introduction to the RESOLVE language and discusses potential inefficiencies of reusable software. Section 3 presents ARC and briefly discusses a sequential implementation of it. Section 4 outlines a parallel implementation of ARC. A model of parallel execution is presented along with architectural mechanisms used to support that model. Section 5 compares our work to previous research, and section 6 lists conclusions.

2 Reusable Software

We examine reusable components developed according to the guidelines of Parnas [11]—information hiding and abstraction are used. Software is reused by developing generic modules exporting ADTs.

*Supported in part by the Applied Information Technologies Research Center and by the National Science Foundation (CCR-8802312).

†Authors can be reached at "welch-l@cis.ohio-state.edu".

Programs are written in RESOLVE (REusable SOftware Language with Verifiability and Efficiency). The first section below provides a brief introduction to RESOLVE language and the second section identifies potential inefficiencies of software constructed from reusable components.

2.1 Overview of RESOLVE

RESOLVE is a language currently under development at The Ohio State University. This paper makes no attempt to justify the features of the language (for such a discussion see [5, 6]), but only considers its efficient implementation.

RESOLVE programs are built from (reusable) program components, or modules, where a module has the following major characteristics that are relevant to this discussion:

- It typically exports a type and some operations to manipulate variables of that type.
- It has a formal specification separate from its implementation (realisation).
- It may have multiple realisations for the same specification.
- It may be generic, i.e., parameterised by types, by operations, and even by other modules.

RESOLVE modules provide operations that are subtly different from the procedures and functions in other languages [5]. Below is a list of some features of RESOLVE operations:

- Parameters are passed "by swapping": at operation invocation, the values of the formal parameters are swapped with the values of actual parameters; on operation return, they are swapped again.
- The arguments to a call must be unique, i.e., the same variable may not appear twice in a particular argument list.
- Local variables are automatically initialised upon entry to a block, and are automatically finalised upon exit from a block.

Other important features of RESOLVE for this work are:

- Assignment (copying) of one variable's value to another is not a part of the language; instead, swapping the values of two variables is the only built-in data movement primitive.
- Operations can have no global variables. Instead, they can have three kinds of data:
  1. operation parameters;
  2. local variables; and
  3. module variables—static variables associated with a particular module instance that are shared among operations exported by that instance.
- Aliased variables are not possible, i.e., the data structure representing a variable's value can only be known by one name at any time.
- There are no constructs in the language for expressing parallelism.

The last feature described above (and perhaps others, too) may raise some questions about the potential efficiency of programs written in RESOLVE, especially when executed on a parallel machine. The next section identifies specific characteristics related to reuse that may lead to inefficient execution.

2.2 Potential Inefficiencies

When software components are designed in a way that permits their reuse in as many situations as possible, inefficiencies can arise at execution time. Programs have many procedure and function calls. Every access to a variable whose type is provided by a reusable component is a call to a module operation. Since the language has no built-in types, almost every statement in a program is a call. Each operation is usually small, too, because of extensive layering in module composition.

In addition to frequent calls and small procedures, programs built from reusable components have another feature that can make them inefficient: they are not designed with a particular architecture in mind. If information hiding is practiced, the author of a software component knows nothing about the context in which it will be used. The author does not know whether the component will be used on a parallel machine or on a sequential machine, and if used on a parallel machine what the granularity of parallelism will be. Programming statements that refer to such information should therefore be avoided within reusable components [9, 8]. This results in simplified design of and increased reuse of the components, but means that parallelism must be detected automatically.

Another potential inefficiency of reusable software is that the source code of components may not be available. In a mature reusable components industry, it is likely that vendors will only provide specifications and object code. The absence of source code

also suggests that parallelism should be extracted from object code or at run-time.

It would appear, then, that the large number of procedure calls would slow the execution of programs. The performance would appear to be further degraded by the inability to write software which is designed with a certain architecture in mind and by the lack of source code on which to perform optimisations. However, this need not be the case. We will show how a simple model of parallel execution and architectural support can combat these potential problems.

3 The Virtual Machine

Efficient execution of RESOLVE programs is achieved by designing a virtual machine, ARC, suited to their execution and by efficiently implementing it. The design and implementation of ARC are discussed below.

3.1 Overview of Features

ARC is formally specified using RESOLVE; the specification is useful in the formal verification of the compiler, but is not further discussed here.

The instruction set of ARC was designed to be useful to compiled RESOLVE programs. Furthermore, instructions were chosen on the basis that they permit efficient implementation of ARC. Instructions not suited to pipelined execution were rejected. In general, design of the instruction set followed RISC principles [12].

The major memory components of ARC that are relevant to this discussion are: the parameter stack, the local data stack, indexed memory, and facility memory (see Figure 1). The parameter stack (PS) is contained in the operation processor and permits operations and their callers to communicate. Parameters are pushed onto the PS before an operation is called; they are popped from the PS when an operation begins execution and returned to the PS before the operation returns; they are finally removed from the PS by the caller.

The local data stack (LDS) contains addresses of the representations of the local variables and parameters of an operation. Upon entry to an operation, the value of the LDS top-pointer is increased by the number of parameters to the operation plus the number of local variables declared in the operation. Entries on the LDS are referenced relative to the top of the stack.

The representations of variables are stored in indexed memory (IM). In addition, IM stores the static data (module variables) of component instances.

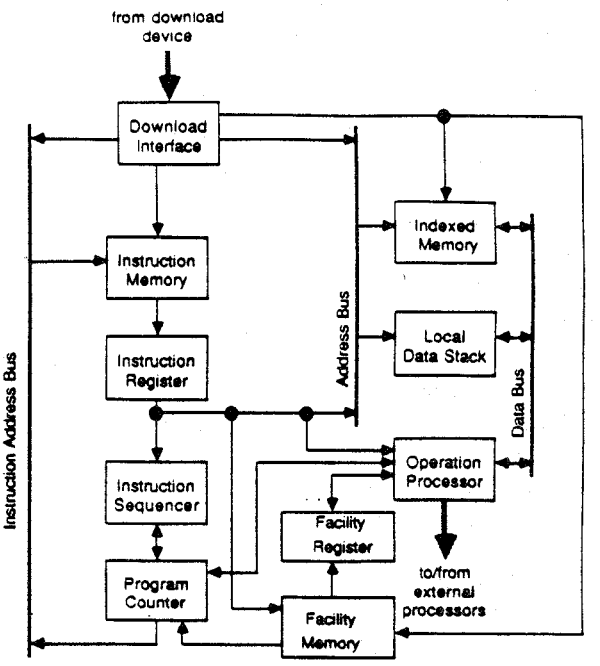

Figure 1: Possible organisation of a sequential implementation of ARC.

Facility memory (FM) and the facility register (FR) permit code sharing among instances of the same module. The linker constructs tables that contain the addresses of operations; these tables are stored in FM. At run-time, the CALL machine instruction uses the FR to access the tables and obtain addresses of operations.

3.2 Sequential Implementation

We have designed a sequential implementation of ARC. The processor contains a large on-chip memory, including storage for the PS, the LDS, IM, and FM. Instruction execution is pipelined; to support this, the processor has the following features:

1. instructions are of uniform length;
2. instructions execute in a single cycle; and
3. delayed branch instructions are provided.

The sequential processor is used as the processing element (PE) in a distributed memory implementation of ARC. The rest of this paper is devoted to the distributed memory implementation.

4 A Parallel Implementation of ARC

The distributed memory implementation of ARC supports a model of parallel execution suited to the kind of reusable components commonly written in RESOLVE. Following a presentation of that model, the architectural mechanisms supporting it are discussed.

4.1 Model of Parallel Execution

The parallel implementation of ARC executes RESOLVE programs in parallel by using continued execution after remote calls. It works as follows:

1. Code for each procedure is statically mapped onto one or more PEs.
2. Each variable is internally represented on some PE as the address of a data structure that represents the variable's abstract value. Data structures representing variables are not moved around. The elements of a conglomerate data structure (e.g., a list or queue) may reside on different PEs.
3. Variables are manipulated by making (possibly remote) procedure calls.
4. A variable is "locked" when it is passed as a parameter to a call and unlocked upon return of the call to which it was passed.
5. A procedure continues execution while awaiting the return of a remote call, until it accesses a locked variable.
6. The PE waits when an attempt is made to access a locked variable; during this time it is idle and cannot accept incoming calls.
7. A PE waiting to access a locked variable retries the access when a remote call returns.

To see how this model of parallel execution works, consider a keyframe animation system. An artist specifies "key" frames of an animation and a computer draws the intermediate frames (those occurring between the key frames). The process of filling in the intermediate frames is referred to as "inbetweening." Inverse kinematics is often applied to generate the in-between frames of jointed figures (such as humans, animals and robots).

A software system that uses inverse kinematics to perform inbetweening might model a human as a *record* with four fields: one field for each of the arms and legs. Each arm and each leg could be represented as an *list* of *links*. Figure 2 shows the "uses" hierarchy among ADT components.

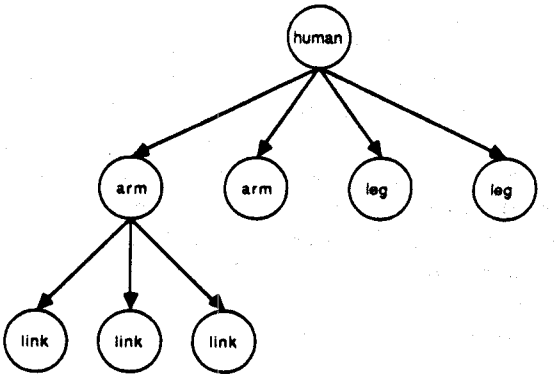

Figure 2: The "uses" hierarchy among ADT components in the keyframe animation system.

Parallelism is obtained on the parallel realisation of ARC by mapping each data structure, plus the code to access it, onto a separate PE. In this example, the data structure representing the human is mapped to one node, each leg and each arm is mapped to its own node, and each link occupies its own node. Parallelism arises by allowing operations on each of the arms and legs to proceed simultaneously, with their positions in a particular frame being computed concurrently. Furthermore, operations on the links constituting a particular arm or leg proceed in parallel.

Note that this is only one possible mapping. Allowing more than one data structure to reside on the same PE is permitted, but it limits potential parallelism. Modules having module data or dyadic operations (operations with two operands of the same type) can also limit potential parallelism. Due to the model of parallel execution, care must be taken when mapping the components onto processors so that deadlock cannot occur. Fortunately, for RESOLVE a static analysis can determine a safe mapping.

4.2 Architectural Mechanisms

The implementation of ARC provides architectural mechanisms to support the continued execution model of parallel execution. To permit conglomerate data structures to have their component elements represented on remote machines, each data

structure is simply represented with a pointer. To facilitate this representation on a distributed memory processor, data addresses are composed of a processor id and a local address.

Remote procedure calls are supported by dedicating a portion of each PE to the task of interfacing with other processors. This part of the PE is responsible for creating call packets containing the arguments to an operation and the address of the operation, and for sending and receiving call packets. Remote calls are also supported by two-part instruction addresses, composed of a processor id and a local address.

Variable locking is supported as follows:

1. A *busy-bit* is associated with each variable.
2. Pushing a variable onto the parameter stack turns its busy-bit on.
3. Popping a value from the parameter stack into a variable's local memory location turns that variable's busy-bit off.

Continued execution of code that has one or more outstanding remote calls is accomplished by implementing the data access (push) instructions to check the busy-bits of items to be pushed, and to pause the processor when a "busy" datum is accessed. Automatic checking of busy-bits prevents "old" values of variables from being passed as parameters to procedures.

The continued execution scheme works because no data structure in a RESOLVE program is known by more than one name, i.e., there is no aliasing. It is not necessary to limit programming to declarative, functional, or single-assignment languages in order to use continued execution. However, it must be impossible to write unsafe programs, i.e., no data accessible within the scope of a call should be accessible outside that scope by another name.

5 Previous Work

Here we compare ARC to similar work been done in the areas of hardware support for programming languages, continued execution during remote calls, hardware detection of parallelism, and distributed memory architectures.

The STARLET computer [4] was engineered to efficiently execute programs composed of ADT modules. While the goal of the STARLET system is the same as ours, its architecture is different. It employs a shared memory processor that attains parallelism through the pipelining of instructions. A basic data type is provided by the architecture and all user-defined data types are mapped onto it. This enables the architecture to be tailored to pipeline the operations of the basic data type. Our architecture differs in several ways. First, it does not use shared memory. Second, it employs two types of parallelism: pipelining within each PE, and manipulation of more than one data structure at a time (where each of the data structures being manipulated is stored on a different PE).

A similar approach is taken in the J-Machine [2, 3], a machine designed to support Concurrent Smalltalk. Whenever a call is made to a method that is not local, or when an operand of a method is not local, a message is sent to the node where the external item is stored. A process is permitted to continue execution while awaiting return of an external message; the "futures" model of continued execution is used. The ARC parallel processor allows a similar form of continued execution. However, the J-machine differs in its model of programming. It assumes an object-oriented paradigm in which messages are sent to objects, whereas ARC supports a more traditional, procedural approach.

Tomasulo [13] uses the hardware detection of parallelism to permit instructions to execute as soon as their operands become available. All instructions within a finite window are permitted to execute in parallel, subject to the availability of the operands to the instructions. Dataflow computing [14] is a generalisation of Tomasulo's technique—the window size is infinite. Although in principle these methods may permit more parallelism than our processor does, they were not employed in ARC since they would seem to introduce large complications into the hardware. (We are, however, reinvestigating this preliminary conclusion).

An example of a RISC distributed memory processor is the Transputer [7]. Both the Transputer and the J-Machine have on-chip RAM; our machine also uses this approach. ARC also uses distributed memory and has an interconnection network that could be similar to the Transputer, or could be a hypercube, for instance.

6 Conclusions

Architectural features were presented to address potential inefficiencies associated with a class of reusable software components. The following lists the potential inefficiencies and shows how each was addressed:

position.

- Procedure calls are the source of parallelism, not the source of inefficiency.
- Layering is encouraged because it increases potential parallelism.

2. Architecture-specific features (such as multiple processors, shared vs. distributed memory) should not be referred to when writing reusable components.
   - Parallelism is automatically extracted at run-time, not described by the programmer.
3. Source code of components is not available.
   - Parallelism is extracted from machine instructions by the hardware.

The potential parallelism of the continued execution model remains to be evaluated. We are pursuing investigations of this question and other issues before building actual hardware for ARC.

While this system is designed with the RESOLVE language in mind, it is also applicable to other programming languages if programmers adhere to the style of programming described here.

References

[1] F. Bastani, W. Hilal, and S.S. Iyengar. Efficient abstract data type components for distributed and parallel systems. *Computer*, 20(10):33–44, Oct 1987.

[2] W. J. Dally, L. Chao, A. Chien, et al. Architecture of a message-driven processor. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 189–196. ACM, June 1987.

[3] W.J. Dally. Fine-grain message-passing concurrent computers. In *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 2–12. ACM, Jan 1988.

[4] W.K. Giloi and R. Gueth. Concepts and realisation of a high-performance data type architecture. *International Journal of Computer and Information Sciences*, 11(1):25–54, 1982.

[5] D. Harms and B.W. Weide. Types, copying, and swapping: Their influences on the design of reusable software components. Technical Report OSU-CISRC-3/89-TR13, The Ohio State University, Mar 1989.

[6] W.A. Hegazy. *The Requirements of Testing a Class of Reusable Software Modules.* PhD thesis, The Ohio State University, Columbus, Ohio, June 1989.

[7] INMOS Limited. *IMS T424 Transputer Reference Manual*, 1984.

[8] R. Jha, J.M. Kamrad II, and D.T. Cornhill. Ada program partitioning language: A notation for distributing Ada programs. *IEEE Transactions on Software Engineering*, 15(3):271–280, Mar 1989.

[9] S. Muralidharan and B.W. Weide. On distributing programs built from reusable software components. Technical Report OSU-CISRC-11/88-TR36, The Ohio State University, Nov 1988.

[10] W.F. Ogden, B.W. Weide, and S.H. Zweben. Design, specification, and implementation of reusable software components using RESOLVE. Technical report, The Ohio State University, April 1989.

[11] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058, Dec 1972.

[12] D.A. Patterson. Reduced instruction set computers. *CACM*, 28(1):8–21, Jan 1985.

[13] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:297–306, Jan 1967.

[14] A.H. Veen. Dataflow machine architecture. *Computing Surveys*, 18(4):365–396, Dec 1986.