Mapping Reusable Software Components onto the ARC Parallel Processor *

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

Abstract

The ARC parallel processor was designed to efficiently execute software constructed from reusable components. We envision ARC as containing several thousand processing elements (PEs). The PEs have a simple design and are inexpensive, encouraging large scale replication. Each PE is directly connected to several other PEs.

In this paper we show how to automatically map the components of a program onto ARC in a way that exploits its features. Mapping consists of two phases. The first phase determines the maximum amount of parallelism attainable from a program in our model of parallel execution. This is done by mapping program components onto logical processors (of which there is an infinite number). The second phase maps the contents of the logical processors onto physical processors (of which there is a limited number).

1 Introduction

This paper explains techniques for mapping reusable software components onto the ARC (Architecture for Reusable Components) parallel processor [12]. The mapping illustrates the usefulness of ARC for efficiently executing reusable software in parallel. It also helps to dispel the myth that reusable software is inherently inefficient.

Section 2 defines the class of reusable software components supported by ARC. Details of the ARC processor are presented in section 3. The remainder of the paper shows how a software system built from reusable components (in a particular but popular view of such components) can be automatically parallelised for execution on ARC. Specifically, we show how to: (1) identify the distributable components of the system; (2) determine the relevant relationships between the components; (3) model the maximum amount of parallelism attainable with the model of parallel execution used; and (4) use the information from steps 1-3 to map components onto the processor nodes of ARC.

2 Reusable Software Components

The goal of this work is the efficient execution of software constructed from the kind of reusable component commonly written in RESOLVE [6]. RESOLVE [7] is a research language that has much in common with Ada, C++ and Modula-2, so the features supported by ARC are applicable to programs written in many languages. The following is a brief overview of RESOLVE features relevant to this paper.

Programs written in RESOLVE are composed of generic modules exporting ADTs (similar to Ada and Modula-2). Modules have formal specifications and there may be multiple implementations for each specification. RESOLVE has no built-in types, so every variable access is performed by calling an operation (procedure or function). Variables are automatically initialised and finalised (as in C++). Assignment (copying) of one variable's value to another is not a part of the language; instead, swapping the values of two variables is permitted. Operations can access three kinds of data: parameters, local variables, and module variables (static variables associated with a particular module instance and shared among operations exported by that module). Aliased variables are not possible since the data structure representing a variable's value can only be known by one name at any time. The types of variables can be determined statically. Modules can not be instantiated dynamically; instantiations appear outside of the code of module operations, and all instantiations are performed before a program begins execution. The language contains no explicit constructs for expressing parallel execution.

3 The ARC Parallel Processor

The ARC PE is designed to address potential ineffi-

*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".

CH2908-2/90/0000/0499/$01.00 © 1990 IEEE

ciencies of reusable software, including:

- Source code should not refer to architecture-specific features (such as multiple processors), since the implementor of a reusable component may not know how or where it will be used.
- Source code of reusable components may not be available after compilation; thus, parallelism must be extracted from object code or at run-time.
- There are frequent calls to small operations (due to extensive layering in module composition).

Since RESOLVE provides no constructs for expressing parallelism, the automatic detection of parallelism is required. ARC relies on components being mapped statically onto PEs in a way that permits efficient parallel execution. ARC executes RESOLVE programs in parallel by using asynchronous remote procedure calls. When a variable is passed as a parameter to a call, it is automatically "locked." To hide the latency of a remote call, calling code is permitted to continue execution until it attempts to access a locked variable, at which time it waits for a remote call to return before it retries the access to the locked variable. A variable is automatically "unlocked" upon return of the procedure to which it was passed as a parameter. With this approach, parallelism is extracted from object code, so the unavailability of source code is not a problem. Furthermore, the problem of frequent calls to small procedures is addressed by making procedure calls the source of parallelism.

The architectural mechanisms used to implement the model of parallel execution are covered in [12]; however, that paper gives no guidelines on mapping program components onto PEs. The remainder of this paper presents a mapping scheme.

4 Representation and Mapping

In this paper we show how to automatically map the components of a program onto ARC in a way that exploits its features. Mapping consists of two phases. The first phase determines the maximum amount of parallelism attainable from a program in our model of parallel execution. This is done by mapping program components onto logical processors (of which there is an infinite number). Logical processors and their interconnections are modeled using a directed acyclic graph (DAG). The second phase maps the logical processors onto physical processors (of which there is a limited number).

4.1 Distributable components

Instantiation of a module means fixing the parameters of a module and choosing one (of possibly many) implementations. An implementation of a module is called a *realization*. An instance of a module is called a *facility*. Specifically, a facility consists of the code of the module operations, and (possibly) module data. To declare a variable, one must instantiate the module providing the type the variable is to have. To perform operations on the variable, one makes calls to operations provided by the facility providing the variable's type.

We define a *distributable component* to be the data structure representing a variable's value and the code to access it. Each distributable component can reside on a processor by itself, if there are enough PEs in an ARC computer system.

4.2 Construction of the DAG

In this section we show how to construct a DAG to model the potential parallelism in a software system. The DAG for a particular program shows the relationships among its distributable components, and the maximum amount of parallelism attainable with the model of parallelism used. The graph can be used to map the components onto ARC.

A program is modeled by a DAG, G = (V, E), where:

- $v \in V$ denotes the operations of a facility, f(v), and the representations of one or more variables whose types are provided by f(v);
- $(x, y) \in E$ indicates that the code of facility f(x) calls some operation(s) provided by facility f(y); and
- there exists exactly one vertex in G with indegree 0, representing the facility at the highest level of the abstraction hierarchy.

The DAG representing a particular program can be constructed as follows:

1. Place a vertex in the graph for each facility used in the program.
2. Place an edge in the graph for each call dependency in the program. Only calls between operations of different facilities are represented in the graph.

Each vertex in the graph can be thought of as a logical processor. We assume there are an infinite number of logical processors and that these processors can be connected in an arbitrary fashion.

The amount of parallelism attainable from the graph can be increased by replicating some of the facilities. The goal is to have one vertex in the graph for each distributable component (a variable plus the code to access it); each distributable component of a program is placed on a separate logical processor. This requires the code of a facility to be replicated. A facility's code can be replicated safely iff (1) it has no module data and (2) it provides no operations with two or more parameters of a type that it exports.

Replication of a facility is reflected in the graph by adding a new vertex. Edges are also added to the graph so the new vertex is adjacent to the same vertices as the vertex representing the facility being replicated; i.e., when a vertex is replicated, so are all edges connected to it.

All edges adjacent to a vertex must be replicated when the vertex is replicated, due to the effects caused by swapping the values of two variables. Recall that RESOLVE has no built-in assignment statement; instead, it permits the values of two variables to be swapped. When the values of two variables are swapped, their representations may change PEs. The DAG must represent all possible situations that could arise during execution, so it must contain edges showing all *potential* calls between components. Specifically, if facility 'x' uses a type provided by facility 'y', the DAG must contain edges from all vertices representing copies of 'x' to all vertices representing copies of 'y'.

The DAG for a program shows the parallelism possible if the number of PEs is unlimited and if the PEs can be connected in an arbitrary fashion. The next section discusses issues that arise when the logical processor graph is mapped onto a physical machine.

4.3 Mapping

After the DAG for a program has been constructed, it is used to map the components of a program onto physical PEs. If the number of vertices in the graph (the number of logical PEs) is less than the number of physical PEs in an implementation of ARC, the physical mapping is trivial; place the content of each logical PE onto its own physical PE. This assumes that the interconnection of the logical PEs can be modeled in the physical machine. If the number of logical PEs is greater than the number of physical PEs, existing techniques for embedding graphs onto parallel processors can be employed. The application of such techniques will not be discussed here, since the purpose of this paper is to show how ARC's model of parallel execution is intended to be used.

There are important issues to consider when mapping components onto physical processors. The communication distance between the PE containing the code of an operation and the PEs where its local variables are stored can affect performance. The tradeoffs between static and dynamic mapping of local variables should be weighed. Static mapping decides where (on which PEs) to place the local variables of an operation before execution of the program, whereas dynamic mapping decides each time the operation executes. The overhead of dynamically mapping the local variables of an operation should be substantially smaller than the cost of executing the code of the operation. Preliminary results suggest that static mapping yields better results. Care should be taken so that deadlock can not occur (see [13] for specific details).

5 Previous Work

This research extends and refines the model of distributing components presented in [6] to increase parallelism.

A model of distributing data structures on a hypercube can be found in [8]. The approach requires that the programmer specify the distribution. Our approach relies on no programmer input.

Similar work has been done with Ada. APPL [4] is a language for specifying mappings of programs to architectures. The approach permits programs to be designed without taking hardware configurations into account—the same philosophy used in the parallelisation of RESOLVE. Units of distribution of Ada programs are examined in [11]. Lundberg [5] focuses on run-time system support for allocation and migration on a multiprocessor. The model of parallel execution in these papers is influenced by the task construct of Ada. In RESOLVE, a different model arises since there is no explicit notion of parallelism in the language's computational model. ARC's parallelism has a much finer granularity.

Models of parallel execution have been developed for object-oriented languages, too. Dally [2] presents a model and corresponding architecture for Concurrent Smalltalk. Bensley et al. [1] develop a model that allows the run-time detection of data-dependent concurrency. These projects have focused on support for late-binding languages, whereas we are supporting an early-binding language.

Similar work has been done in the area of architectural support. The STARLET computer [3] was engineered to efficiently execute programs that use ADTs. While the goal of the STARLET system is the same as ours, its architecture is quite different. It employs a shared memory processor that attains parallelism

through the pipelining of instructions. Our architecture does not use shared memory, and while it does not preclude the use of instruction pipelining in the processor nodes, parallelism is achieved through a different means.

A similar approach to ARC is taken in the J-Machine [2]. 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. The J-Machine also permits a process to continue execution while awaiting a return from an external message. The ARC processor allows a similar form of continued execution, but differs in that it supports a language with static binding and does not use futures for continued execution.

Tomasulo [9] 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 their operands. Dataflow computing [10] is a generalisation of Tomasulo's technique—the window size is infinite. Although these methods permit more parallelism than our processor does, they introduce large complications into the hardware.

6 Conclusions

The ARC processor, used in combination with the mapping strategy outlined here, encourages the development of layered software by increasing parallelism in correspondence to increases in layering. This approach directly attacks the criticism that reusable software is inefficient.

While our system is designed with the RESOLVE language in mind, it is also useful for other programming languages. If programmers adhere to the style of programming described, the architecture and parallel model apply.

References

[1] E. Bensley et al. An execution model for distributed object-oriented execution. In *OOPSLA '88 Proceedings*, pages 316–322. ACM, Sept 1988.

[2] 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.

[3] 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, Feb 1982.

[4] 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.

[5] L. Lundberg. A parallel Ada system on an experimental multiprocessor. *Software—Practice and Experience*, 19(8):787–800, Aug 1989.

[6] 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.

[7] 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.

[8] L.R. Scott, J.M. Boyle, and B. Bagheri. Distributed data structures for scientific computation. In *Proceedings of the 1987 Hypercube Conference*, pages 55–66, 1987.

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

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

[11] R.A. Volz, T.N. Mudge, G.D. Buzzard, et al. Translation and execution of distributed Ada programs: Is it still Ada? *IEEE Transactions on Software Engineering*, 15(3):281–292, Mar 1989.

[12] L.R. Welch and B.W. Weide. Architectural support for efficient execution of reusable software components. In *The Fifth Distributed Memory Computing Conference*. ACM, April 1990.

[13] L.R. Welch and B.W. Weide. Avoiding deadlock when mapping reusable software components onto the ARC parallel processor. In *ISMM International Conference on Parallel and Distributed Computing, and Systems*. ISMM, Oct 1990.