Data-Driven and Demand-Driven Computer Architecture

PHILIP C. TRELEAVEN, DAVID R. BROWNBRIDGE, AND RICHARD P. HOPKINS

Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, England

Novel data-driven and demand-driven computer architectures are under development in a large number of laboratories in the United States, Japan, and Europe. These computers are not based on the traditional von Neumann organization; instead, they are attempts to identify the next generation of computer. Basically, in data-driven (e.g., data-flow) computers the availability of operands triggers the execution of the operation to be performed on them, whereas in demand-driven (e.g., reduction) computers the requirement for a result triggers the operation that will generate it.

Although there are these two distinct areas of research, each laboratory has developed its own individual model of computation, stored program representation, and machine organization. Across this spectrum of designs there is, however, a significant sharing of concepts. The aim of this paper is to identify the concepts and relationships that exist both within and between the two areas of research. It does this by examining data-driven and demand-driven architecture at three levels: computation organization, (stored) program organization, and machine organization. Finally, a survey of various novel computer architectures under development is given.

Categories and Subject Descriptors: C.0 **[Computer Systems Organization]**: General— *hardware/software interfaces; system architectures;* C.1.2 **[Processor Architecture]**: Multiple Data Stream Architectures (Multiprocessors); C.1.3 **[Processor Architecture]** Other Architecture Styles—*data-flow architectures; high-level language architectures,* D 3 2 **[Programming Languages]** Language Classifications—*data-flow languages; macro and assembly languages; very high-level languages*

General Terms: Design

Additional Key Words and Phrases Demand = driven architecture, data = driven architecture

INTRODUCTION

For more than thirty years the principles of computer architecture design have largely remained static [ORGA79], based on the von Neumann organization. These von Neumann principles include

(1) a single computing element incorporating processor, communications, and memory;
(2) linear organization of fixed-size memory cells;
(3) one-level address space of cells;
(4) low-level machine language (instructions perform simple operations on elementary operands);
(5) sequential, centralized control of computation.

Over the last few years, however, a number of novel computer architectures based on new "naturally" parallel organizations for computation have been proposed and some computers have even been built. The principal stimuli for these novel architectures have come from the pioneering work on data flow by Jack Dennis [DENN74a, DENN74b], and on reduction languages and machines by John Backus [BACK72, BACK73] and Klaus Berkling [BERK71, BERK75]. The resulting computer architecture research can be broadly classified as either data driven or demand driven. In data-driven (e.g., data-flow) computers the availability of operands triggers the execution of the operation to be performed on them, whereas in demand-driven (e.g., re-

CONTENTS

duction) computers the requirement for a result triggers the operation that will generate it.

Although the motivations and emphasis of individual research groups vary, there are basically three interacting driving forces. First, there is the desire to utilize concurrency to increase computer performance. This is based on the continuing demand from areas such as weather forecasting and wind tunnel simulation for computers with a higher performance. The natural physical laws place fundamental limitations on the performance increases obtainable from advances in technology alone. And conventional high-speed computers like CRAY 1 and ILLIAC IV seem unable to meet these demands [TREL79]. Second, there is the desire to exploit very large scale integration (VLSI) in the design of computers [SEIT79, MEAD80, TREL80b]. One effective means of employing VLSI would be parallel architectures composed of identical computing elements, each containing integral capabilities for processing, communication, and memory. Unfortunately "general-purpose" organizations for interconnecting and programming such architectures based on the von Neumann principles have not been forthcoming. Third, there is the growing interest in new classes of very high level programming languages. The most well-developed such class of languages comprises the functional languages such as LISP [MCCA62], FP [BACK78], LUCID [ASHC77], SASL [TURN79a], Id [ARVI78], and VAL [ACKE79b]. Because of the mismatch between the various principles on which these languages are based, and those of the von Neumann computer, conventional implementations tend to be inefficient.

There is growing agreement, particularly in Japan and the United Kingdom, that the next generation of computers will be based on non-von Neumann architecture. (A report [JIPD81a] by Japan's Ministry of International Trade and Industry contains a good summary of the criteria for these fifth-generation computers.) Both data-driven and demand-driven computer architecture are possible fifth-generation architectures. The question then becomes, which architectural principles and features from the various research projects will contribute to this new generation of computers?

Work on data-driven and demand-driven architecture falls into two principal research areas, namely, data flow [DENN79b, GOST79a] and reduction [BERK75]. These areas are distinguished by the way computation, stored programs, and machine re-

sources are organized. Although research groups in each area share a basic set of concepts, each group has augmented the concepts often by introducing ideas from other areas (including traditional control-flow architectures) to overcome difficulties. The aim of this paper is to identify the concepts and relationships that exist both within and between these areas of research. We start by presenting simple operational models for control flow, data flow, and reduction. Next we classify and analyze the way computation, stored programs, and machine resources are organized across the three groups. Finally, a survey of various novel computer architectures under development is given in terms of these classifications.

1. BASIC CONCEPTS

Here we present simple operational models of control flow, data flow, and reduction. In order to compare these three models we discuss each in terms of a simple machine code representation. These representations are viewed as instructions consisting of sequences of arguments—operators, literal operands, references—delimited by parentheses:

(arg0 arg1 arg2 arg3 ··· arg$n$ − 1 arg$n$).

However, the terms "instruction" and "reference" are given a considerably more general meaning than their counterparts in conventional computers. To facilitate comparisons of control flow, data flow, and reduction, simple program representations for the statement a = (b + 1) * (b − c) are used. Although this statement consists of simple operators and operands, the concepts illustrated are equally applicable to more complex operations and data structures.

1.1 Control Flow

We start by examining control flow, the most familiar model. In the control-flow program representations shown in Figure 1, the statement a = (b + 1) * (b − c) is specified by a series of instructions each consisting of an operator followed by one or more operands, which are literals or references. For instance, a dyadic operation such as + is followed by three operands; the first two, b and 1, provide the input data and the last, t1, is the reference to the shared memory cell for the result. Shared memory cells are the means by which data are passed between instructions. Each reference in Figure 1 is also shown as a unidirectional arc. Solid arcs show the access to stored data, while dotted arcs define the flow of control.

In traditional sequential (von Neumann) control flow there is a single thread of control, as in Figure 1a, which is passed from instruction to instruction. When control reaches an instruction, the operator is initially examined to determine the number and usage of the following operands. Next the input addresses are dereferenced, the operator is executed, the result is stored in a memory cell, and control is passed implicitly to the next instruction in sequence. Explicit control transfers are caused by operators such as GOTO.

There are also parallel forms of control flow [FARR79, HOPK79]. In the parallel form of control flow, shown in Figure 1b, the implicit sequential control-flow model is augmented by parallel control operators. These parallel operators allow more than one thread of control to be active at an instance and also provide means for synchronizing these threads. For example, in Figure 1b the FORK operator activates the subtraction instruction at address i2 and passes an implicit flow of control on to the addition instruction. The addition and subtraction may then be executed in parallel. When the addition finishes execution, control is passed via the GOTO i3 instruction to the JOIN instruction. The task of the JOIN is to synchronize the two threads of control that are released by the addition and subtraction instruction, and release a single thread to activate the multiply instruction.

In the second parallel form of control flow, shown in Figure 1c, each instruction explicitly specifies its successor instructions. Such a reference, i1/0, defines the specific instruction and argument position for the control signal, or *control token*. Argument positions, one for each control signal required, are represented by empty bracket symbols ( ), and an instruction is

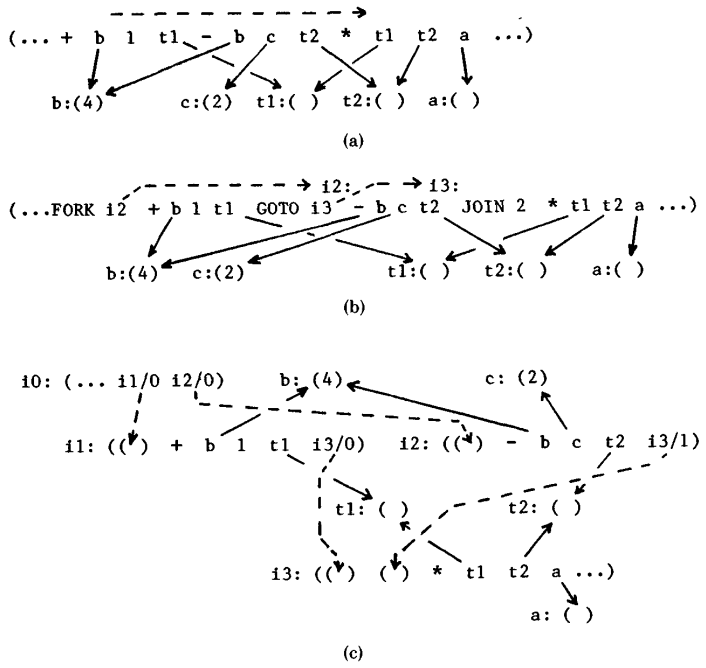

Figure 1. Control-flow programs for a = (b + 1) * (b − c): (a) sequential, (b) parallel "FORK–JOIN"; (c) parallel "control tokens."

executed when it has received the required control tokens. The two parallel forms of control flow, illustrated by Figures 1b and 1c, are semantically equivalent; FORKS are equivalent to multiple successor instruction addresses and JOINs are equivalent to multiple empty bracket arguments.

The sequential and parallel control-flow models have a number of common features: (1) data are passed indirectly between instructions via references to shared memory cells; (2) literals may be stored in instructions, which can be viewed as an optimization of using a reference to access the literal; (3) flow of control is implicitly sequential, but explicit control operators can be used for parallelism, etc.; and (4) because the flows of data and control are separate, they can be made identical or distinct.

1.2 Data Flow

Data flow is very similar to the second form of parallel control flow with instructions activated by tokens and the requirement for tokens being the indicated ( ) symbols. Data-flows programs are usually described in terms of directed graphs, used to illustrate the flow of data between instructions. In the data-flow program representation shown in Figure 2, each instruction consists of an operator, two inputs which are either literal operands or "unknown" operands defined by empty bracket ( ) symbols, and a reference, i3/1, defining the specific instruction and argument position for the result. A reference, also shown as a unidirectional arc, is used by the producer instruction to store a *data token* (i.e., result) into the consumer. Thus data are passed directly between instructions.

An instruction is enabled for execution when all arguments are known, that is, when all unknowns have been replaced by partial results made available by other instructions. The operator then executes, removing the inputs from storage, processing them according to the specified operation,

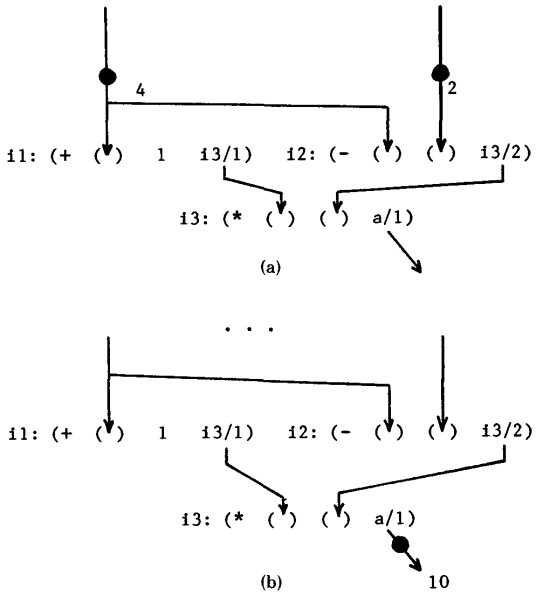

Figure 2. Data-flow program for a = (b + 1) * (b − c) (a) Stage 1; (b) Stage 4.

and using the embedded reference to store the result at an unknown operand in a successor instruction. In terms of directed graphs, an instruction is enabled when a data token is present on each of its input arcs. During execution the operator removes one data token from each input arc and releases a set of result tokens onto the output arcs.

Figure 2 illustrates the sequence of execution for the program fragment a = (b + 1) * (b − c), using a black dot on an arc to indicate the presence of a data token. The two black dots at Stage 1 in Figure 2 indicate that the data tokens corresponding to the values of b and c have been generated by predecessor instructions. Since b is required as input for two subsequent instructions, two copies of the token are generated and stored into the respective locations in each instruction. The availability of these inputs completes both the addition and the subtraction instruction, and enables their operators for execution. Executing completely independently, each operator consumes its input tokens and stores its result into the multiplication instruction "i3." This enables the multiplication, which executes and stores its result corresponding to the identifier "a," shown at Stage 4.

In the data-flow model there are a number of interesting features: (1) partial results are passed directly as data tokens between instructions; (2) literals may be embedded in an instruction that can be viewed as an optimization of the data token mechanism; (3) execution uses up data tokens—the values are no longer available as inputs to this or any other instruction; (4) there is no concept of shared data storage as embodied in the traditional notion of a variable; and (5) sequencing constraints—flows of control—are tied to the flow of data.

1.3 Reduction

Control-flow and data-flow programs are built from fixed-size instructions whose arguments are primitive operators and operands. Higher level program structures are built from linear sequences of these primitive instructions.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.