

DEMAND DRIVEN EXECUTION USING FUTURE GATED SINGLE
ASSIGNMENT FORM

By

Omkar Ulhas Javeri

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Computer Engineering

MICHIGAN TECHNOLOGICAL UNIVERSITY

2020

© 2020 Omkar Ulhas Javeri

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computer Engineering.

Department of Electrical and Computer Engineering

Dissertation Advisor: *Dr. Soner Önder*

Committee Member: *Dr. David Whalley*

Committee Member: *Dr. Saeid Nooshabadi*

Committee Member: *Dr. Zhenlin Wang*

Department Chair: *Dr. Glen E. Archer*

Dedication

To my parents: Chetana Javeri (mother) and Ulhas Javeri (father)

You always encouraged me to follow my dreams and supported me in every way possible to achieve my goals.

To my advisor: Dr. Soner Önder

Who encouraged me to explore new horizons and guided me when I was in a need. You have motivated me to believe in small ideas and to sculpture them into a magnificent master piece. Your belief in me has made it possible for me to architect this work into a realizable design.

To my family and friends

Thank you for always being there for me.

Contents

List of Figures	xi
List of Tables	xv
Acknowledgments	xvii
List of Abbreviations	xix
Abstract	xxi
1 Introduction	1
2 Background	9
2.1 Execution Paradigms	10
2.1.1 Controlflow Computing	10
2.1.2 Data-flow Computing	11
2.1.2.1 MIT Data-Flow Processor	12
2.1.2.2 Monsoon: An Explicit Token-Store Architecture . .	15
2.1.3 Demand-Driven Computing	17
2.1.3.1 Reduction Machines	18

2.2	Future Gated Single Assignment Form	19
3	Controlflow FGSA Assembly Representation	25
4	Demand-driven Execution of Programs	31
5	Programming Language Pragmatics	37
5.1	Addressing Modes Within an Environment	39
5.1.1	Literal Addressing Mode.	40
5.1.2	Frame Direct Addressing Mode.	41
5.1.3	Displacement Addressing Mode.	41
5.2	Use of Addressing Modes	42
5.3	Synchronization in DDE	44
5.3.1	<i>WITH</i> Synchronization Instruction.	47
5.3.2	<i>THEN</i> Synchronization Instruction.	48
5.3.3	<i>EITHER</i> Synchronization Instruction.	48
5.4	Environment, Frames and Mapping of Frames	49
5.5	Deallocating Frames	53
5.6	Passing Arguments to Functions	54
5.7	Procedure Calls	55
5.8	Memory Ordering	57
5.9	Representation of Loops	59
5.9.1	Sequential Unrolling of Loops	63

6	Microarchitecture	69
6.1	Pipeline Overview	70
6.2	Scalar Memory	74
6.2.1	Return Address Storage	76
6.2.2	Reservation Station Storage	77
6.3	Pipeline Details	78
6.3.1	Evaluation Pipeline using RAS	79
6.3.2	Execution Pipeline	82
6.3.3	Send-back-and-commit Pipeline using RAS	84
6.3.4	Frame Allocation Stage	86
6.3.5	Loop Level Frame Allocation Stage	87
6.3.6	Evaluation Pipeline using CAM	88
6.3.7	Send-back-and-commit Pipeline using CAM	90
6.4	Multi-issue Pipelined Implementation	91
7	Simulation of Design, Assembly Language Programming, Debug- ging and Results	95
7.1	Description of Instruction Set Architecture in ADL	96
7.2	Description of Microarchitecture in ADL	97
7.3	Compiling Imperative Programs and Assembly Representation	99
7.3.1	Conversion of Internal Representation of FGSA to Functional Form of FGSA	100

7.3.1.1	Generation of Set for the Loop Environment	100
7.3.1.2	Generation of Set for Outer Environment	102
7.4	Debugging in ADL for Demand-Driven Processor Model	104
7.5	Performance Results	106
7.5.1	Handcoded Evaluated Benchmarks Experimental Parameters	107
7.5.2	Evaluation	109
7.5.2.1	Scalability of DDE Paradigm	110
7.5.2.2	Control-flow Single Issue versus DDE	112
7.5.2.3	Superscalar Processor versus DDE	114
7.5.3	Compiler Generated Benchmarks	119
7.5.3.1	Scalability of DDE Paradigm	121
7.5.3.2	Control-flow Single Issue versus DDE	124
7.5.3.3	Superscalar Processor versus DDE	126
8	Conclusion	131
	References	133
A	Instruction Set Architecture	137

List of Figures

2.1	Future data and control dependency	19
2.2	Program fragment to perform scalar addition	21
2.3	Scalar addition FGSA in graphical form	22
2.4	Controlflow 3-address intermediate representation of FGSA	23
3.1	Controlflow 3-address intermediate representation of FGSA	27
3.2	Controlflow assembly representation of FGSA	28
4.1	Data value state transition in DM	34
5.1	Environment illustration	38
5.2	An environment represented using frame size of four	40
5.3	Demand generation and propagation of values	43
5.4	State transition of data and signals in DM	46
5.5	An environment mapped to a single frame in frame memory	50
5.6	An environment with multiple frame representation in frame memory	52
5.7	Environment deallocation in DDE	54
5.8	Procedure call	56

5.9	Memory dependencies representation using predicate	58
5.10	Memory dependencies predicate available at run time	59
5.11	Virtual loop body tree	61
5.12	Loop unrolling by demanding initiator node	62
5.13	Sequential loop unrolling in DDE	64
5.14	A static instance of loop code for DDE in main memory	66
6.1	Demand driven execution pipeline overview	71
6.2	Scalar memory	75
6.3	Return address storage	77
6.4	Evaluation pipeline using RAS	80
6.5	Execution pipeline	83
6.6	Send-back-and-commit pipeline using RAS	85
6.7	Frame allocation stage	87
6.8	Demand driven execution pipeline	89
7.1	Livermore kernel 1	107
7.2	Livermore kernel 3	108
7.3	Livermore kernel 5	108
7.4	Livermore kernel 11	109
7.5	Livermore kernel 12	109
7.6	1 token dde processor as base with max 16 active outer loop iteration and procedure with max 64 inner loop iterations	111

7.7	1 token dde processor as base with max 64 active outer loop iteration and procedure with max 128 inner loop iterations	112
7.8	MIPS pipelined processor as base with max 16 active outer loop iteration and procedure with max 64 inner loop iterations	113
7.9	MIPS pipelined processor as base with max 64 active outer loop iteration and procedures with max 128 inner loop iterations	114
7.10	Superscalar processor as base with max 16 active outer loop iteration and procedure with max 64 inner loop iterations	115
7.11	Superscalar processor as base with max 64 active outer loop iteration and procedures with max 128 inner loop iterations	116
7.12	Superscalar processor as base without memory disambiguation, with max 16 active outer loop iteration and procedure with max 64 inner loop iterations	117
7.13	Superscalar processor as base without memory disambiguation, with max 64 active outer loop iteration and procedures with max 128 inner loop iterations	118
7.14	Livermore kernel 7	120
7.15	Livermore kernel 9	120
7.16	Livermore kernel 10	121
7.17	1 token dde processor as base with maximum 16 active outer loop iteration and procedure with maximum 64 inner loop iterations . .	122

7.18	1 token dde processor as base with max 64 active outer loop iteration and procedure with max 128 inner loop iterations	123
7.19	MIPS pipelined processor as base with max 16 active outer loop iteration and procedure with max 64 inner loop iterations	124
7.20	MIPS pipelined processor as base with max 64 active outer loop iteration and procedures with max 128 inner loop iterations	125
7.21	Superscalar processor as base with max 16 active outer loop iteration and procedure with max 64 inner loop iterations	126
7.22	Superscalar processor as base with max 64 active outer loop iteration and procedures with max 128 inner loop iterations	127
7.23	Superscalar processor as base without memory disambiguation, with max 16 active outer loop iteration and procedure with max 64 inner loop iterations	128
7.24	Superscalar processor as base without memory disambiguation, with max 64 active outer loop iteration and procedures with max 128 inner loop iterations	129

List of Tables

5.1	Addressing mode with example and meaning	41
6.1	States of EV-tag	76
6.2	States of OP-tag	76
7.1	Number of instruction in a Kernel	130
A.1	Instruction fields	144

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1533828, XPS: Full: FP: Collaborative Research: Sphinx: Combining Data and Instruction Level Parallelism through Demand Driven Execution of Imperative Programs. This work is also supported in part by the National Science Foundation under Grant No. CCF-1450062.

A parallel work on compiler development is being carried out at Florida State University and is supported by National Science Foundation under Grant No. CCF-1533846

Special thanks to Dr. Soner Önder for shaping my efforts in the correct direction making this project what it is today.

Thanks to Zhaoxiang Jin, Tino Moore, Marcus Stojceovich, Gorkem Asilioglu at Michigan Technological University and Ryan Baird, Dr. David Whalley at Florida State University for contributing during research discussions.

Thanks to my parents, faculty members and my friends for their feedback and support.

List of Abbreviations

ADL	Architecture Description Language
CAM	Content-Addressable Memory
DDE	Demand-Driven Execution
DM	Data Memory
FGSA	Future Gated Single Assignment form
Evtag	Evaluation tag
EV-token	Evaluation token
IM	Instruction Memory
ISA	Instruction Set Architecture
Optag	Operand tag
OP-token	Operand token
PC	Program Counter
RAS	Return Address Storage
RTL	Register Transfer Level
SM	Scalar Memory
WB-token	Write-Back token

Abstract

This dissertation discusses a novel, previously unexplored execution model called Demand Driven Execution (DDE), which executes programs starting from the outputs of the program, progressing towards the inputs of the program. This approach is significantly different from prior demand-driven reduction machines as it can execute a program written in an imperative language using the demand-driven paradigm while extracting both instruction and data level parallelism. The execution model relies on an executable Single Assignment Form which serves both as the internal representation of the compiler as well as the Instruction Set Architecture (ISA) of the machine. This work develops the instruction set architecture, the programming language pragmatics and the micro-architecture for the demand-driven execution paradigm.

Chapter 1

Introduction

Existing models based on the Von-Neumann program execution model are not scalable, limited by the compiler's ability to identify dependencies at compile time as well as the cost of analyzing these dependencies at run-time. Currently, extracting parallelism is achieved by using architectures based on Multiple Instruction Multiple Data (MIMD) architectures such as Multi-core Central Processing Units (CPU) and Single Instruction Multiple Data (SIMD) architectures such as Graphic Processing Units (GPU), as well as accelerators such as Digital Signal Processors (DSP)s, Field Programmable Gate Arrays (FPGA), or a combination of more than one. The SIMD and MIMD parallel architectures execute a program based on the controlflow paradigm. Under this paradigm, the extracted parallelism is limited by how well a given program can be analyzed at compile time and mapped to a particular architecture. Further

complexity involves writing a parallel program to extract the desired parallelism in a machine dependent manner. An alternative approach includes dataflow architectures which allow the execution of instructions depending on operand availability [9, 15]. Dataflow architectures naturally provide synchronization of parallel activities, but these architectures primarily have been designed for functional languages and they cannot naturally handle dataflow through memory for imperative programming languages. Thus, dataflow architectures have not been commercially viable.

This dissertation explores and develops a previously unexplored execution model called Demand Driven Execution (DDE). In DDE, the evaluation of an instruction is carried out only when its value is needed. The program execution starts from the outputs of the program which triggers the first instruction to be evaluated. This instruction demands other instructions which are required to generate its operands. Those instructions eventually demand other instructions which are required to generate their results, hence creating a demand tree of instructions. As a result, instructions at the leaves of the tree will have all their operands available and will be among the first ones to be executed. An instruction in DDE is executed when all its operands are available. In DDE, an instruction according to its classification is capable of demanding up to two instructions and an additional instruction if it is predicated. Thus this model leads to significant levels of parallelism. The demand-driven nature of the approach explicitly represents dependencies, allowing for better scalability in terms of both synchronization and communication. We build and explore architectures based

on the DDE paradigm using the Future Gated Single Assignment [7, 8] (FGSA) form.

The DDE architecture is designed keeping general purpose programming in mind. We are able to run programs written in imperative languages such as C which are mechanically translated into the FGSA form and FGSA code is directly executed by the architecture. Doing so, the DDE paradigm exploits instruction and data level parallelism with the goal of achieving higher Instructions Per Cycle (IPC) compared to running the same imperative program on an equivalent superscalar processor.

Generally speaking, we should compare the complexity of a DDE processor in terms of performance and power, to that of controlflow architectures. It takes a similar amount of power to execute any arithmetic, logical and memory instruction like ADD, AND, MUL, LW, SW, whether such instructions are executed in a conventional superscalar processor or a dataflow processor. The DDE paradigm completely eliminates all branch instructions since synchronization is embedded in the paradigm. The difference in performance and power comes in the way instructions are evaluated, scheduled and executed. In a superscalar processor, an instruction is evaluated when the Program Counter (PC) points to it. In a dataflow processor an instruction is evaluated when another instruction generates an operand for that instruction. In a DDE processor an instruction is evaluated when another instruction demands a value from that instruction. In a superscalar processor it is assumed the operands of an instruction are available, or, will be shortly available when the Program Counter (PC)

points to it. In a data flow processor, a instruction is asynchronously scheduled for execution when both of its operands are available, which has its own operand matching mechanism. DDE processors use a unique dynamic operand matching mechanism to asynchronously schedule an instruction when all of its operands are available. The paradigm achieves this by dynamically generating a dataflow graph, starting at the output (i.e., the root) of the graph and starts execution as soon as any of the leaf nodes are reached.

In summary, the primary goal of this dissertation is to develop the instruction set architecture, the programming language pragmatics and the micro-architecture mechanisms necessary for realizing this type of execution efficiently.

Currently, the FGSA form developed by Ding et al. [6] can properly represent imperative programs. Ding’s dissertation which developed the FGSA form has also laid out the theoretical framework necessary for automatic translation of imperative programs into the FGSA form, demonstrated how the form can be used as the internal representation of the compiler and how various forms of optimizations can be performed using the representation. A subsequent publication [7] laid the fundamentals and showed that in a theoretical sense, the form can support all three known execution models, namely, controlflow execution (CFE), dataflow execution (DFE) and demand-driven execution (DDE).

As discussed above, among the three execution models, the DDE paradigm is very

promising in terms of extracting large amounts of instruction and data parallelism. However, before the paradigm can be practically used, many issues need to be addressed and these issues are the focus of this dissertation. In other words, the goal of this dissertation is to develop the DDE paradigm from a theoretical framework into a practical and usable implementation.

In order to develop this paradigm into a practically realizable design, we make the following contributions as we approach the problem as follows:

1. The theoretical FGSA specification lacks the necessary pragmatics, particularly how memory is represented and is accessed. We therefore develop the necessary pragmatics including the program layout, function and loop representations.
2. We define a memory model and machine model to run executable demand-driven code.
3. We develop an understanding of any additional fundamental issues towards constructing an actual DDE machine.

In order to achieve these goals, we develop:

1. The necessary addressing modes for the DDE machine. Addressing modes are necessary to have systematic and efficient accesses to instructions and data;

2. The procedure calling mechanism for the DDE machine, namely, passing of arguments and return of values from procedures;
3. Efficient transformations to enable translation and parallel execution of loops in DDE;
4. A demand-driven ISA representation suitable for efficient execution on such a machine for arithmetic, logical, memory, synchronization, data transfer and special instructions in DDE;
5. A multiple-issue of the microarchitecture for efficient, parallel execution of DDE programs.
6. Necessary simulation infrastructure for evaluating the resulting architecture.

In the rest of the dissertation, we discuss our approach to each of these elements. Chapter 2 gives the basic knowledge necessary to understand the rest of the dissertation. We introduce the new paradigm and connect concepts with the prior art. The section also gives a detailed description of FGSA [7, 8]. Chapter 3 gives a brief description of controlflow assembly representation and show how controlflow intermediate representation of FGSA is converted into controlflow assembly representation. Chapter 4 introduces an abstract view of the memory model and elaborates on instruction synchronization envisioned for the demand-driven processor. Chapter 5

describes the memory model and its layout for generating demand-driven programs. The section also describes a formal method to support high-level language features, such as, procedure calls, loops and parameter passing conventions. Chapter 6 describes the microarchitecture developed for the demand-driven processor. Chapter 7 describes the simulator infrastructure used to develop the demand-driven processor and presents an evaluation of the designs.

Chapter 2

Background

This work builds on prior work which includes the design of the Future Gated Single Assignment (FGSA) form by Ding et al. [6, 7, 8] which is summarized in Section 2.2. Contributions of the dissertation include the controlflow assembly program representation of the FGSA form, demand-driven intermediate representation, the concept of environments and frames, environment and frame addressing, various synchronization primitives in DDE, parameter passing between environments, the development of the instruction set architecture for the demand-driven execution paradigm, and naturalized sequential loop unrolling for exploiting loop-level parallelism under this paradigm.

2.1 Execution Paradigms

Execution paradigms can be broadly classified into three different models of execution based on the flow of control and data. The first model is based on sequential computation or the traditional Von-Neumann program execution model based on incrementing and modifying a program counter and is known as *Controlflow* computing. The second model is based on the availability of data to derive instruction execution. In this model, an instruction computes as soon as its data operands are available. As the computation in this model is driven by the availability of data, it is known as *Data-flow* computing. The third model is driven by the availability of a result and demands computation required to generate its result. As the computation in this model is driven by demand for the result and its computation, it is referred to as *Demand-driven* computing.

2.1.1 Controlflow Computing

Controlflow computing has been the dominant form of computing for decades. The execution model for a controlflow machine is based on updating a *Program Counter (PC)* which decides the execution flow. In this model, the PC holds the address of the instruction to be executed and usually is incremented sequentially. The flow of

program execution is controlled and modified by the use of transfer of control instructions like branch and jump instructions. These transfer of control instructions modify the PC to accomplish a different execution flow instead of the sequential incrementing of the PC. Transfer of control instructions can be conditional or unconditional. Unconditional jump instructions allow the execution of a different control path by unconditionally modifying the PC with a new value, thus transferring the control to a different part of the code. The conditional branch instructions embed a comparison to gate the flow of control. The comparison conditions like less-than, greater-than, equal-to, not-equal-to, less-than-equal-to, greater-than-equal-to are used to activate a control transfer with the help of branch instructions.

2.1.2 Data-flow Computing

The execution in a data-flow machine is data-driven. An instruction in a data-flow machine is enabled for execution when it receives all the required operands (i.e. all the required data is available). Data-flow programs are represented using a directed graph. The nodes in the directed graph act as an operator or a link. These nodes are connected by arcs. The data values in the form of *tokens* are placed on the arcs and transported from one node to another. The instruction at a node is enabled when *tokens* on all the input arcs of the node is available. An enabled operator can fire at any time when all the input *tokens* are available. The operator consumes all the

input *tokens*, performs the required operation and places result output *token* on its output arc. A link can be used to send the result *token* to more than one destination. The link consumes the *token* at its input arc and places copies of the *token* at all of its output arcs. In static data-flow architectures an operator or link can execute only when there is no *token* present on the output arc of that operator or link.

Data-flow computers are classified depending on their communication topology as direct communication and packet communication. In direct communication, the processing elements are directly connected to each other. The Data-Driven Machine #1 (DDM1) is an example of direct communication machine [19, 20]. The packet communication can be further classified as static packet communication machines and dynamic packet communication machines. We visit a few notable data-flow architectures in this chapter.

2.1.2.1 MIT Data-Flow Processor

We summarize the description of MIT Data-Flow Processor written by Dennis et al. and Treleaven et al. [4, 5, 19]. MIT Data-Flow Computer uses packet communication architecture with *token* storage. There are two types of packets, control packets and data packets. The Data-flow processor consists of five major segments. An asynchronous protocol is used to transmit packets between these five segments. The five major segments of the Data-flow processor are:

1. The memory of the processor is formed of instruction cells consisting of three register to hold the instruction and its two operands.
2. The arbitration network routes multiple operation packets which are ready with their instructions and operands from instruction cells to the appropriate processing section by decoding the instruction part of the packet.
3. The processing section consist of operation units to perform required operation on the available operands and generate one or more computed values and its target address in the memory.
4. The control network directs control packets from the processing section to the appropriate cells in the memory section.
5. The distribution network section directs the data packets from the processing section to the appropriate cells in the memory section.

Each instruction cell consists of an instruction composed of an operator of the data-flow code, several destination addresses and three registers one to hold the instruction and two other to hold the operands. The instruction cells in the memory section gets enabled for execution when it has the instruction and all the required operands. The operands are received via the distribution network and are written to respective register in the instruction cell. Similarly the instruction is received as a control packets

via the control network and is made available as a instruction by writting to the register in the instruction cell. The enabled instruction along with its operands are sent as an operation packet to the processing section via the arbitration network. The arbitration network directs the packet from the instruction cell to the respective processing units by decoding the operation code of the instruction. The processing units compute one or more result packets which are sent to the instruction cells via control network and distribution network. The result packet is composed of the result value and the destination address derived from instruction processed by the processing unit. The result packet generated is classified into two types as either a control packet or a data packet. The control packet contains a boolean value or an acknowledgment signal and is delivered via the control network. The data packet contains an integer or complex value and is delivered via the distribution network. The result packet is delivered to the instruction cell denoted by its destination address via control network and distribution network. The result packets received by an instruction cell can be an operand or an acknowledgment signal. If all the required result packets by an instruction cell are received, then it can enable that instruction and can generate a new operation packet to be sent to the processing unit.

2.1.2.2 Monsoon: An Explicit Token-Store Architecture

We summarize the description and working of Monsoon machine written by Papadopoulos and Culler [3, 18]. Monsoon is a dynamic packet communication data-flow machine which uses Explicit Token Store (ETS) mechanism that was developed at MIT Laboratory for Computer Science. Explicit token store architecture uses a dynamic storage of fixed size called a “frame”. Frames are allocated dynamically during function invocation and are used to store the tokens of the function. These frames are released on completion of function invocation. Each location in the frame is associated with a presence bit which is initially empty. Dynamic data-flow execution triggering is achieved when the presence bits is set for a particular location. The ETS is capable of directly executing dynamic data-flow graphs. A token in Monsoon machine is comprised of a tag and a value. The tag holds the information for the instruction pointer and the frame pointer where the instruction pointer points to the location of an instruction to be executed and the frame pointer points to an activation frame. The instruction pointed by the instruction pointer holds the information about the instruction to be executed, the offset location in the activation frame allocated for token matching, one or more destination instruction pointers for the result token and additional information for the input port, as “left” or “right” for each destination. A token on arrival checks for the presence bit of its allocated location in the activation

frame. If the location is empty the token is stored in the location by setting the corresponding presence bit as full and no further execution is performed. If the presence bit for the location is *full* the token is read from the location, the presence bit is set to *empty*. The instruction is executed generating one or more result tokens.

The ETS allows parallel calls for activation frames, hence the caller and callee can run in parallel. This mechanism allows allocation of frames for multiple iterations of a loop to run concurrently. It is the responsibility of the compiler to compile the data-flow graph in such a way that reusing a frame is possible only after its previous use is complete. The Monsoon machine is comprised of a group of processing elements connected using a multistage packet switch network and a number of interleaved memory modules. Each processing element has an eight stage pipeline. The format of the message is uniform through the Monsoon machine which is nothing but the flow of tokens thus allowing the hardware to have a uniform inter and intra processor communication. The program compiled to run on a Monsoon machine is a collection of disjoint data-flow graphs and is called a code block which consists of loop body or a function in a high level program. Each code block is dynamically assigned to a processing element. In case of a loop, every iteration is allocated its own frame which can be on separate processing elements. An activation frame and all the tokens within the frame are computed on the same processing element. This approach allows the inter-processor traffic to be minimized and also helps to keep the processing element pipeline full.

Some aspects of the Monsoon architecture and its design principles have been used in the design of our demand-driven processor models.

2.1.3 Demand-Driven Computing

The execution in demand-driven computing is driven by the availability of result. In this model, the demand for a result triggers a demand for the evaluation of an instruction which computes the result. The evaluated instruction demands its operands which can be another instruction or an operand value might be readily available. The flow of demand for the result starts from the outputs of the program and progresses towards the inputs of the program. The output of the program is the first instruction to be evaluated which demands other instructions required to compute its value. The demand chain continues until an instruction demands the input values of the program. The instruction demanding the input values which have its operands readily available will be among the first to be executed. A typical example of a computer capable of executing in Demand-driven computation is reduction machines.

2.1.3.1 Reduction Machines

Reduction machines were designed for executing functional programs [19]. Treleaven et al. classifies reduction machines into two categories, outermost reduction and innermost reduction [19]. In innermost reduction, all innermost arguments of a function to be executed need to be available before the function can execute. The execution starts from the innermost operands computing the value required for the outer instructions, thus making the value available to the outermost instruction to compute the result. In outermost reduction, the outermost instruction demands the instruction which generates its operands which continues to demand other instructions required to generate their value until the demand reaches the innermost instruction which has all its operands available and will be among the first ones to execute and compute a value allowing demand-driven execution.

The demand-driven execution paradigm can be considered to be an outermost reduction engine, where the program to be executed has been translated from an imperative program into the Future Gated Single Assignment form, as discussed in the next section.

2.2 Future Gated Single Assignment Form

Future Gated Single Assignment (FGSA) form developed by Ding et al. [6, 7, 8] is a single assignment representation which can be used by a compiler as its internal representation as well as by a microarchitecture as its instruction set.

In FGSA, every definition is unique and dominates all its uses in a Control Flow Graph (CFG). The representation uses two executable gating functions, PSI (ψ) and ETA (η) controlled by a predicate for two different uses. The ψ function acts as a gating function to control the flow of data to implement selection and has the form $x_{\text{dest}} = \psi_P(\text{arg}_1, \text{arg}_2)$. The flow of data to x_{dest} is controlled using the gating predicate P , which if true, allows the value of the first argument arg_1 to flow into a subsequent use of x_{dest} . Otherwise, the value of the second argument arg_2 flows into a subsequent use of x_{dest} . The η function acts as a gating function to control the flow of data out of loops and has the form $x_{\text{dest}} = \eta_P(y_{\text{arg}})$. This function allows its argument to flow to subsequent uses when its predicate P is true and controls how loops function.

I1: $z = x + y$	I1: $u = v + z_f$	I1: if P_f then $a = a + 1$
I2: $u = v + z$	I2: $z = x + y$	I2: $u = v + z_f$
I3: $P = (u < z)$	I3: $P = (u < z)$	I3: $z = x + y$
I4: if P then $a = a + 1$	I4: if P then $a = a + 1$	I4: $P = (u < z)$
(a) True dependency	(b) Future data dependency	(c) Future data and control dependency

Figure 2.1: Future data and control dependency

FGSA form uses *future dependencies* [16]. An instruction which has an operand

defined by a later instruction in controlflow order is said to have a future dependency.

In Figure 2.1(a), a true data dependence involving the variable z is shown where the value of z is defined by instruction I_1 and is used by instruction I_2 . When the order of the defining instruction and the using instruction is reversed as seen in Figure 2.1(b), the dependency becomes a future dependency such that the variable z is used in instruction I_1 before it is defined in instruction I_2 . In other words, a future dependency allows reversing the data-flow sequence where a true data dependency exists between the instructions and permits an instruction to be hoisted above the instruction which defines its operands. As the use of the variable z is hoisted before its definition, it is written as a future variable z_f and the subscript ‘f’ stands for future. Future dependencies can be employed with control dependencies as well.

In Figure 2.1(a) the instruction I_3 defines the predicate P which guards instruction I_4 . Hoisting I_4 before I_3 yields Figure 2.1(c) where the guarded instruction uses the predicate P before its definition by instruction I_4 . As the guarding predicate use is hoisted before its definition, it is written as a future predicate, P_f . As it can be seen, using future dependencies provides us with the freedom of ordering code without being restricted by data or control dependencies in a controlflow representation. This freedom is necessary to represent imperative programs in the controlflow execution model using FGSA as we illustrate shortly through an example how an imperative program is represented.

Figure 2.2 shows an example program fragment computing a partial vector sum. The

loop start value ‘y’ is dependent on the controlflow path taken based on the value of variable P. Controlflow graph (CFG) FGSA representation of this code is shown in Figure 2.3.

```
main ()
{
    int x = 0;
    int y = 0;
    int n = 100;
    int a[100];
    int P, i;
    if (P > 0)
    { y = 50; }
    for (i = y; i < n; i++)
    { x = x + a[i]; }
    print (x);
}
```

Figure 2.2: Program fragment to perform scalar addition

The ψ function implements the selection between the value 0 (y_1) and 50 (y_2) as seen in Figure 2.3 expressed as $y_3 = \psi_{P_1}(y_{2f}, y_1)$. This selection function must be inserted at a place which dominates all its uses and the only candidate is block B1. As the definition of the value of y_2 falls after the point of use with this only choice, the dependence, needs to become a future dependence and is written as y_{2f} . The following ψ functions, ψ_{Pr_2} and ψ_{Pr_3} , utilize special predicates, Pr_2 and Pr_3 , called *read-once predicates* as seen in block B3. A read-once predicate is a special predicate which return the value when it is read for the first time and for subsequent read returns false value. Thus, ψ_{Pr_2} and ψ_{Pr_3} will allow the flow of data values from their first arguments x_1 to x_2 and y_3 to y_4 respectively when encountered for the first time

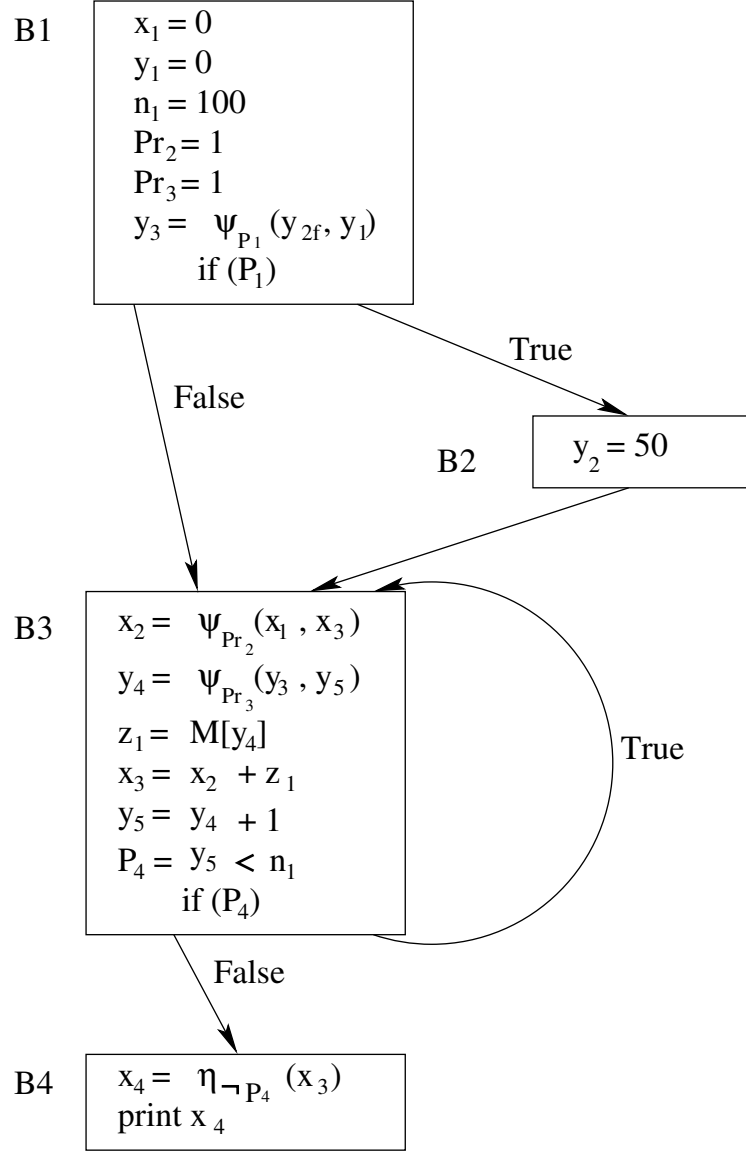


Figure 2.3: Scalar addition FGSA in graphical form

as the predicates Pr_2 and Pr_3 are true. The values of the predicates Pr_2 and Pr_3 will be false for subsequent instances thus allowing the flow of data values from the second argument x_3 to x_2 and y_5 to y_4 respectively, in subsequent instances.

The η function is used to regulate the dataflow out of the loop as shown in block

B4, and is expressed as $x_4 = \eta_{\neg P_4}(x_3)$. The flow of data is controlled by the value of the predicate P_4 . If the value of the gate ($\neg P_4$) is true, i.e., when the value of the predicate P_4 is false, then the loop has terminated and the value of the argument x_3 is allowed to flow into the following uses.

```

      L1:   $x_1 = 0$ 
            $y_1 = 0$ 
            $n_1 = 100$ 
B1       $Pr_2 = 1$ 
            $Pr_3 = 1$ 
            $y_3 = \psi_{P_1}(y_{2_f}, y_1)$ 
           if ( $P_1$ ) goto L3

B2      L2:   $y_2 = 50$ 

      L3:   $x_2 = \psi_{Pr_2}(x_1, x_3)$ 
            $y_4 = \psi_{Pr_3}(y_3, y_5)$ 
            $z_1 = M[y_4]$ 
B3       $x_3 = x_2 + z_1$ 
            $y_5 = y_4 + 1$ 
            $P_4 = y_5 < n_1$ 
           if ( $P_4$ ) goto L3

B4      L4:   $x_4 = \eta_{\neg P_4}(x_3)$ 
           print  $x_4$ 

```

Figure 2.4: Controlflow 3-address intermediate representation of FGSA

The CFG form of an FGSA program can be converted into a linear 3-address intermediate representation for a controlflow processor by following an identical procedure a typical compiler follows when flattening an SSA-CFG form. This process involves a topological visiting of CFG nodes. The resulting 3-address intermediate representation form of the example program is shown in Figure 2.4.

We have provided a brief overview of the background material required to understand this dissertation.

In the next chapter, we present the controlflow assembly representation for FGSA.

Chapter 3

Controlflow FGSA Assembly

Representation

Controlflow assembly program representation for FGSA requires the definition of assembly language equivalents for the FGSA special functions ψ and η , and special read once-predicates. The representation also needs to be able to encode future dependencies. In order to implement the ψ functions, we use two conditional move instructions found in commercial processors [10]. This is because, a conditional move instruction implements the data motion for only one argument. Using two instructions with the same destination accomplishes the desired functionality. Although this may look like a violation of single assignment property of programs, practically only one of the two conditional moves can write. The read-once predicates are implemented by

having a tagged memory and the location is marked as “read-once” by using a special instruction such as “load immediate and set read-once” (lir). The processor then zeroes the location whenever the location is read by any instruction. Alternatively, a non-tagged memory can also be utilized and an atomic “read and set to zero” (rsz) instruction can be utilized to read the location’s value. Similar instructions such as *test-and-set* have been used in commercial processor implementations.

Future dependencies are represented by appending the suffix ‘.f’ to each operand at the assembly level, encoded as a single bit associated with the corresponding operand of the instruction. The processor shelves the instruction with a future operand until the producer instruction is encountered. FGSA construction guarantees that the definition will always be encountered due to its construction and the single assignment property of the representation.

Given these changes, the 3-address intermediate representation of FGSA shown in Figure 3.1 can be converted into assembly representation form for a controlflow processor as shown in Figure 3.2. The program is still in single assignment form as each definition is unique. Execution starts with the first instruction at label L1. The ψ_{P_1} instruction is represented using two conditional move instructions, *move conditional on not zero (movn)* and *move conditional on zero (movz)*, one of which will write, depending on the value of the predicate P_1 which is in register \$7. The ψ_{P_1} instruction also uses a future dependency represented using the suffix ‘.f’ as \$8.f. The controlflow

using the predicate P_1 is defined by the conditional branch instruction *blez \$7, \$L3*. The label L3 is the loop header. The loop iterates until the conditional branch *bgtz \$14, \$L3* is not taken (i.e., the predicate P_4 becomes false). The ψ instructions, using read-once predicates ψ_{Pr_2} and ψ_{Pr_3} are represented using two conditional move instructions, *movn* and *movz* for each of them. This coding allows the flow of the initial values x_1 to x_2 and y_3 to y_4 respectively in the first pass through the loop header. The predicates Pr_2 and Pr_3 become false once read, allowing the value of x_3 to flow to x_2 and y_5 to flow to y_4 in subsequent iterations. The η function becomes a conditional move instruction *movz \$15, \$14, \$12* which moves the value of

```

L1:  x1 = 0
      y1 = 0
      n1 = 100
B1   Pr2 = 1
      Pr3 = 1
      y3 =  $\psi_{P_1}(y_{2f}, y_1)$ 
      if (P1) goto L3

B2  L2:  y2 = 50

      L3:  x2 =  $\psi_{Pr_2}(x_1, x_3)$ 
            y4 =  $\psi_{Pr_3}(y_3, y_5)$ 
            z1 = M[y4]
B3   x3 = x2 + z1
      y5 = y4 + 1
      P4 = y5 < n1
      if (P4) goto L3

B4  L4:  x4 =  $\eta_{\neg P_4}(x_3)$ 
      print x4

```

Figure 3.1: Controlflow 3-address intermediate representation of FGSA

```

$L1:  li      $1,0          #  $x_1 = 0$ 
      li      $2,0          #  $y_1 = 0$ 
      li      $3,100        #  $n_1 = 100$ 
      lir     $4,1          #  $Pr_2 = 1$ 
      lir     $5,1          #  $Pr_3 = 1$ 
      movn    $6,$8.f,$7    #  $y_3 = \psi_{P_1=1}(y_{2_f})$ 
      movz    $6,$2,$7      #  $y_3 = \psi_{P_1=0}(y_1)$ 
      blez    $7,$L3        # if ( $\neg P_1$ ) Jump to L3
$L2:  li      $8,50         #  $y_2 = 50$ 
$L3:  movn    $9,$4,$1      #  $x_2 = \psi_{Pr_2=1}(x_1)$ 
      movz    $9,$4,$12     #  $x_2 = \psi_{Pr_2=0}(x_3)$ 
      movn    $10,$5,$6     #  $y_4 = \psi_{Pr_3=1}(y_3)$ 
      movz    $10,$5,$13    #  $y_4 = \psi_{Pr_3=0}(y_5)$ 
      lw      $11,M($10)    #  $z_1 = M[y_4]$ 
      addu    $12,$9,$11    #  $x_3 = x_2 + z_1$ 
      addiu   $13,$10,1     #  $y_5 = y_4 + 1$ 
      slt     $14,$13,$3    #  $P_4 = y_5 < n_1$ 
      bgtz    $14,$L3       # if ( $P_4$ ) Jump to L3
$L4:  movz    $15,$14,$12   #  $x_4 = \eta_{\neg P_4}(x_3)$ 
      jal     print $4      # print  $x_4$ 

```

Figure 3.2: Controlflow assembly representation of FGSA

x_3 to x_4 when the predicate P_4 , i.e. $\$14$ becomes false. Now the value of x_4 is available, and it is printed. As it can be seen, except the outlined additional instructions and the representation of read-once locations and future dependencies, translation of FGSA programs to executable controlflow assembly is fairly straight-forward and follows general compiler mechanisms of reducing 3-address intermediate form into the assembly language.

Although we do not use the controlflow form of FGSA further in this dissertation it forms the basis for the demand-driven representation elaborated later. In any case, it can also be utilized for future controlflow processor implementations which may

require an executable single assignment form.

In the next chapter, we turn our attention to the demand-driven execution and discuss a rather abstract view of the memory model as well as instruction synchronization envisioned for the demand driven execution.

Chapter 4

Demand-driven Execution of Programs

Before we can discuss the implementation of a demand-driven processor, we need to revisit demand-driven execution in an abstract manner to facilitate an understanding behind our approach in designing the architecture. We choose the DDE architecture to be a Harvard architecture embodying an Instruction Memory (IM) and a Data Memory (DM). The IM stores the program code and the DM stores computed scalar values of executed instructions or partially computed values. Similar to P-RISC [15] and Monsoon [18] models, data structures such as arrays and structures reside in heap memory. Hence our architecture embodies a conventional memory with the possibility of incorporating an I-structure [2] like support structure to enable proper synchronization and communication through memory.

In our architecture, an instruction i is of the form $d \leftarrow a \text{ op } b$ where d , a , b are locations in DM. An instruction i in IM has a one to one correspondence with location d in DM such that location d in IM contains instruction i . In other words, instruction i at location d in IM writes to location d in DM. Every location in DM has an additional tag field. This tag field associated with every location in DM is used to represent the current state of data and the possible transitions during program execution. The tag field assumes one of three states for a data value. These states are *Empty* (i.e. data not available), *Partial* (i.e., a data value is available for one of the two operands of an instruction), and *Full* (i.e. data for all source operands are available and the output of the instruction is available).

The execute operation for instruction x in controlflow execution is given by :

$$\begin{aligned}
& \text{execute}(x) : DM[x].value \leftarrow DM[IM[x].op1].value \\
& \quad \text{op } DM[IM[x].op2].value; \\
& \quad DM[x].tag \leftarrow \text{present}.
\end{aligned} \tag{4.1}$$

whereas, the execute operation for instruction x in demand-driven execution is given

by :

$$\begin{aligned}
& execute(x) : DM[x].value \leftarrow evaluate(IM[x].op1) \\
& \quad op \ evaluate(IM[x].op2); \\
& \quad DM[x].tag \leftarrow present.
\end{aligned} \tag{4.2}$$

In other words, the execution of a dyadic instruction involves two function calls, one to *evaluate* the operands and another to perform the operation using the values returned by the functions.

The *evaluate* function is defined as :

$$\begin{aligned}
& evaluate(x) : if DM[x].tag = present \\
& \quad then DM[x].value else execute(x).
\end{aligned} \tag{4.3}$$

Note that, this step recursively evaluates a dependent chain of instructions.

Figure 4.1 illustrates the set of states where each state corresponds to a particular tag. Initially, all tag fields are *Empty*. For instructions with only one operand, the tag field is set to *Full* whenever data arrives. *Partial* represents the state when one of the operands of a dyadic instruction has arrived, but the other operand has not arrived yet. The first arrived operand is stored by stating the partial availability of the result. The tag bit is set to *L-Partial* if the left operand arrives. Otherwise, it

Instruction	Value	Tag
$b = 4$	$\overset{s3}{4}$	Empty $\overset{s3}{Full}$
$c = 28$	$\overset{s3}{28}$	Empty $\overset{s3}{Full}$
$a = b + c$	$\overset{s4}{28} \quad \overset{s5}{32}$	Empty $\overset{s4}{R-Partial}$ $\overset{s5}{Full}$

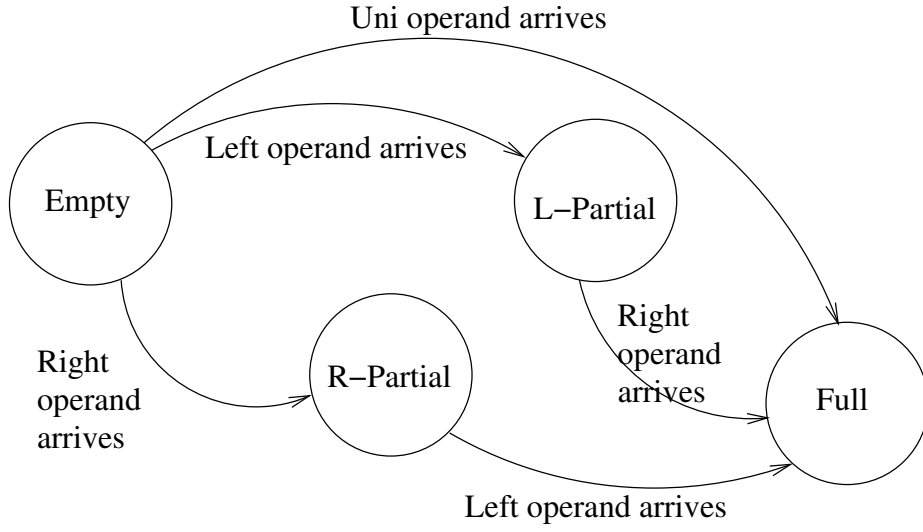


Figure 4.1: Data value state transition in DM

is set to *R-Partial*. When the other operand arrives, the instruction is sent to the execution unit alongside with both operands. Finally, the computed result is stored, the tag field is set to *Full* and the result is returned to requesting instructions.

Figure 4.1 illustrates the execution of instruction $a = b + c$. The demand and memory states for this instruction are explained in steps $s1$ to $s5$ in the figure. In step one, a demand is received for the value of location a and the tag field is *Empty*. In step two, demands for the values of operands b and c are placed. Note that these demand

requests may arrive at their destinations in any order. In step 3, the tag fields, of operand b and c are checked and both are found to be empty. The values of operands $b = 4$ and $c = 28$ are available and the tag fields of both instructions are set to *Full*. In step four, the operand values b and c may arrive in any order. Let's assume the operand value of c arrives before b . Right operand c , being the first operand to arrive, is stored in the location corresponding to a in DM and the tag field is set to *R-Partial*. In step five, the other operand arrives. Left operand b received along with the stored right operand c are now sent to the execution unit. Next, the computed value of $b + c$ is stored in location a and the tag field is set to *Full*.

In order to realize this basic execution model, we need to develop the necessary pragmatics for the demand driven execution.

In a nutshell, the pragmatics of demand-driven execution means we have a memory model on which the program layout can be based, as well as any support that is necessary to enable efficient translation of high-level language features, such as, procedure calls, loops and parameter passing conventions, which are the primary topics of the next chapter.

Chapter 5

Programming Language

Pragmatics

Although the abstract view presented in the previous chapter helps us to understand how the demand-driven execution of programs can be realized, terms such as “demands”, “returns”, “location” remains abstract until we provide actual mechanisms of achieving each of these.

For this purpose, we first discuss the concept of environments, which provides a mechanism of representing program building blocks such as functions, loops and single data locations. We then show that we can implement efficient addressing modes to provide the necessary accesses. In this respect, we consider the sequence of code

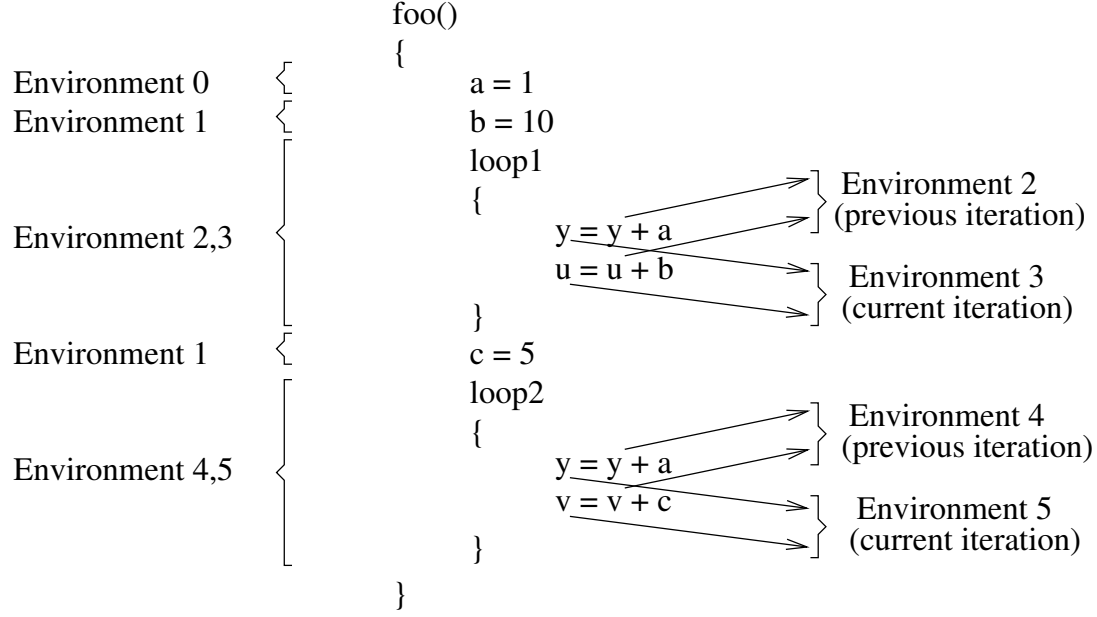


Figure 5.1: Environment illustration

belonging to a functional instance, the code before and after a given loop, the loop body, and the code’s run-time instances all to be “environments”. Following the approach in block-structured languages, we lexicographically number each environment¹, as shown in Figure 5.1. Environment-0 represents a function instance in which the variable ‘a’ has a lifetime through the end of the code. Environment-1 represents code outside the scope of any loop, such as loop1 or loop2. The variable ‘b’ is within the scope of loop1 and the variable ‘c’ is within the scope of loop2. Therefore, the lifetime of these variables coincides with the lifetime of the corresponding loops. The loop1 environment requires two dynamic instances during the execution, namely, Environment-2 and Environment-3. This is because loop carried values need to be

¹The block structured controlflow approach does not distinguish loop instances. In DDE we need to.

kept in the previous instance and communicated to the next instance of the loop environment. For example, the variable ‘y’ and the variable ‘u’ in a loop instance require values from the previous iteration. Thus, the two run-time instances are represented as Environment-2 (i.e., the previous iteration) and Environment-3 (i.e, the current iteration). Similarly, two run-time instances are required for loop2 and they are represented as Environment-4 representing the previous iteration and Environment-5 as the current iteration.

5.1 Addressing Modes Within an Environment

As previously stated, the instructions in IM have a one-to-one relationship with the locations in DM such that the destination ‘d’ of instruction ‘i’ in IM writes to location ‘d’ in DM. The environments from IM needs to have the corresponding environments in DM. To facilitate efficient implementation of environments, we introduce the concept of *frames*.

Similar to paging, we divide an environment into fixed-size continuous blocks of locations called *frames*. To illustrate, Figure 5.2 shows an environment consisting of 11 instructions distributed among frames each with a frame size of 4. Frame 0 and frame 1 have four instructions in each, whereas frame 2 has three instructions and one unused location.

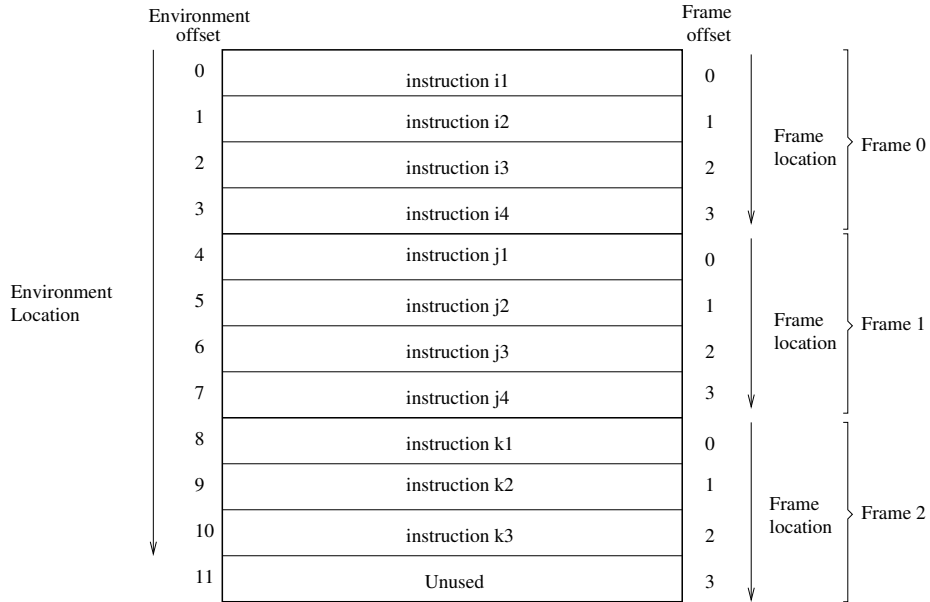


Figure 5.2: An environment represented using frame size of four

Table 5.1 illustrates the three different addressing modes used to address a location within a frame with examples.

5.1.1 Literal Addressing Mode.

The literal (immediate) addressing mode is used in arithmetic operations, comparison operations and to load immediate values. As illustrated in Table 5.1-(1), the literal addressing mode performs the addition of a constant value 8 and the value demanded from location 1 of a frame from DM and stores the result at location 2 of the same frame in DM.

	Addressing mode	Example instruction	Meaning	When Used
(1)	Literal (Immediate)	2: Add 1, #8	$\text{loc}[2] \leftarrow \text{loc}[1] + 8$	for constant
(2)	Frame direct	3: Add 1, 2	$\text{loc}[3] \leftarrow \text{loc}[1] + \text{loc}[2]$	rapid access to a location within a frame
(3)	Displacement	1: Add 2, 2[3]	$\text{loc}[1] \leftarrow \text{loc}[2] + \text{loc}[\text{loc}[3]+2]$	accessing using a pointer for Env location with an offset.

Table 5.1
Addressing mode with example and meaning

5.1.2 Frame Direct Addressing Mode.

The frame direct addressing mode is used to directly reference a location within a frame, hence, it allows rapid access by instructions within a frame. Table 5.1-(2) illustrates the use of the frame direct addressing mode. The example performs the addition of the value demanded from location 1 and the value demanded from location 2 of a frame in DM. It then stores the result at location 3 of the same frame in DM.

5.1.3 Displacement Addressing Mode.

The displacement addressing mode is used to indirectly reference a location from another frame. The location referenced within the frame holds a pointer to another frame. An offset value is added to the pointer value to get the final displacement. The generated displacement gives the pointer to the actual location desired. Table 5.1-(3) illustrates the use of the displacement addressing mode. The example adds the value demanded from location 2 with the value demanded indirectly from another frame

location, address of which is calculated by adding the displacement to the frame offset stored at location 3. Both location 2 and location 3 belong to the same frame in DM. The result is stored at location 1 of the same frame in DM.

5.2 Use of Addressing Modes

Demand-driven execution using the developed addressing modes can be realized by using two pointers, the Instruction Frame Pointer (IFP) which points at the first location of an instruction frame and the Data Frame Pointer (DFP) which points at the first location of the corresponding data frame. All demand requests and returned responses are sent to DM. Therefore, all transactions use data addresses. Data frames establish the necessary association between the instructions and data to fetch the instructions when the frame is created. We abbreviate the DFP for the target environment which receives the demand request as DFP_t and the DFP for the caller environment receiving the return value is represented as DFP_r .

Each demand request is of the form $\langle \text{target address, return address} \rangle$, where the address is a DFP and an offset within that frame. In other words, the demand request is encoded as $\langle DFP_t + \text{offset}, DFP_r + \text{offset} \rangle$. The demand message also includes information indicating whether the operand is a left operand or a right operand. A left operand message is represented as $\langle DFP_t + \text{offset}, DFP_r + \text{offset.left} \rangle$ and the

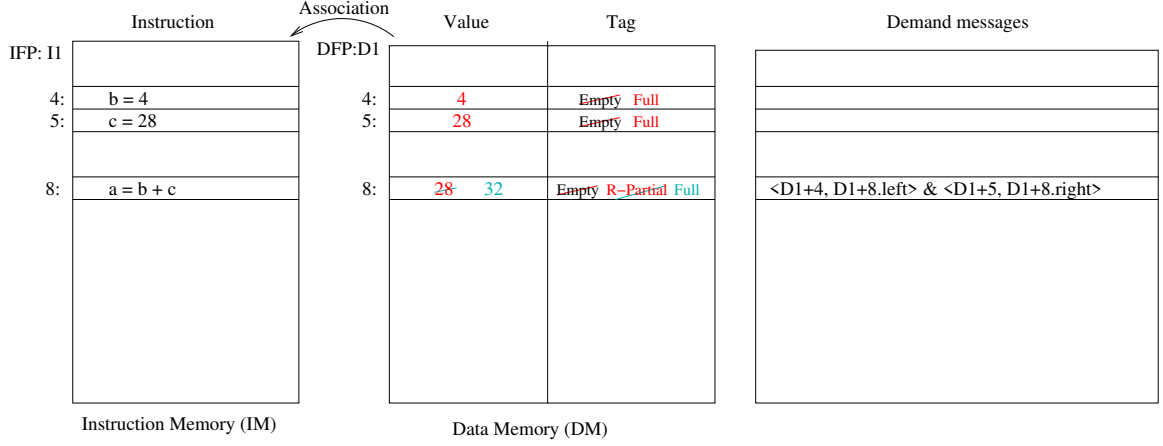


Figure 5.3: Demand generation and propagation of values

right operand message is represented as $\langle DFP_t + \text{offset}, DFP_r + \text{offset.right} \rangle$. Let's go through an example of how the demand requests are generated and how the data value is returned. Figure 5.3 illustrates an example executing the simple expression $a = b + c$. When a demand request for the variable a is received, two demand requests are placed simultaneously:

$$\langle DFP_t + \text{offset}_b, DFP_r + \text{offset}_a.\text{left} \rangle \text{ which is } \langle D1 + 4, D1 + 8.\text{left} \rangle \quad (5.1)$$

$$\langle DFP_t + \text{offset}_c, DFP_r + \text{offset}_a.\text{right} \rangle \text{ which is } \langle D1 + 5, D1 + 8.\text{right} \rangle \quad (5.2)$$

Initially, the tag for variables b and c are *Empty* in DM. The instructions at offset_b i.e. location 4 and offset_c i.e. location 5 in IM are fetched. The value of variables b and c are written to DM and corresponding tags are set to *full*. Now the values of variables b and c are returned to a . The operand values b and c may arrive in any

order. Let's assume the operand value of c arrives before b . Right operand c being the first operand to arrive is stored in the location corresponding to a in DM and the tag field is set to *R-Partial*. Now the other operand arrives. Left operand b received along with the stored right operand c are now sent to the execution unit. Next, the computed value of $b + c$ is stored in a and the tag field is set to *Full*.

5.3 Synchronization in DDE

The DDE paradigm naturally provides instruction synchronization. Every demand for a value generated by an instruction returns the value to the demanding instruction as soon as it becomes available. The paradigm implements the necessary synchronization through tags attached to memory cells. However, in addition to this implicit synchronization mechanism, we have special cases where we need to control how synchronization occurs. This kind of conditional synchronization is observed when we need to trigger the demand for two operands but are interested in only the value of the first operand. Similarly, there are cases where the order of demand is important. For example, we may need to trigger the demand of two operands in a specific order. In this case, the first operand is demanded and the second operand is demanded after the value of the first operand becomes available. Yet another case is when we need to trigger completion of two operands but we need to consume only either one of these operands.

To properly describe the synchronization in DDE, we adapt the programming notation by Andrews Gregory et al. [1]. In this notation, the symbol $\parallel$ is used to represent concurrent operations. The symbol $;$ is used to represent an ordering. Thus, a simultaneous demand to a and b is represented as $\mathbf{a}\parallel\mathbf{b}$. Similarly, the ordering of b *after* a is represented as $\mathbf{a};\mathbf{b}$.

To implement these synchronization primitives, we need to distinguish between the completion of a demand request and the returned data. Although in most cases they coincide, for synchronization purposes we may be interested in the completion of the request separately from the arrival of actual data. As previously described, the abstract DDE machine sets the tag field to *Full* when the requested data is computed. This implies the actual completion of the demand request. However, there are cases where the data which accompanies the completion signal may be irrelevant. Several of the synchronization constructs are in this category. We introduce a new tag, *complete* to distinguish between the completion of a demand signal from the arrival of actual data and provide separate means to target each. *Complete* means the demand request is complete, but no data value has been written to the word and the demand request returns only a completion signal. The demand signal is generated using *eval.s* instruction. This instruction returns the demand completion signal and sets the tag to *Complete*. In contrast, a data value is demanded using *eval.d* instruction. This instruction returns the actual data and sets the tag field to *Full* when the data becomes available.

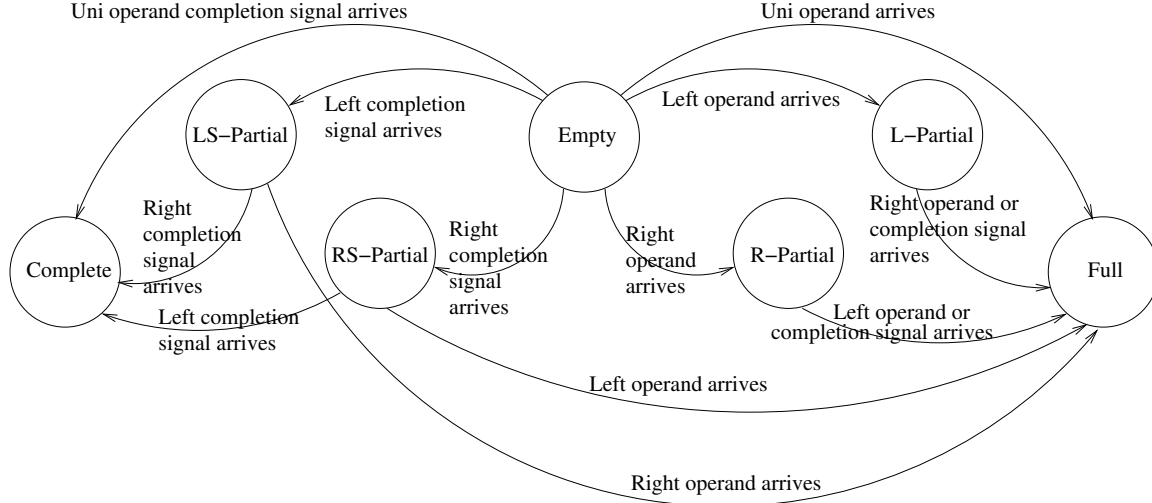


Figure 5.4: State transition of data and signals in DM

Figure 5.4 illustrates the state transitions for a DDE machine in terms of data and signals. We add states to Figure 4.1 to incorporate demand signals alongside the available data value. Initially, all tag fields are set to *Empty*. For instructions with only one operand, the tag field is set to *Full* whenever the data arrives or set to *Complete*, if a signal arrives. *S-Partial* represents the state when one of the completion signals of a dyadic instruction has arrived, but the other completion signal or data has not arrived yet. The first arrived signal indicates the partial completion of the result. The tag is set to *LS-Partial* if the left signal arrives. Otherwise, it is set to *RS-Partial*. If the synchronization involves two signals, when the second completion signal arrives, the state is changed to *Complete*. In the case where the synchronization involves a signal and a value, the arrival of the value for the second operand changes the state to *Full*.

In the following instruction descriptions, we adopt field names and the symbols for the left operand as *lop* and for the right operand as *rop* representing frame locations accessed through some addressing mode and *dest* represents DM location corresponding to the current instruction. Therefore *dest* is implicit and is not encoded with the instruction.

5.3.1 **WITH** Synchronization Instruction.

The *with* instruction is defined as *WITH lop, rop*. This instruction acts as a synchronization fork instruction by assigning the contents of *lop* to *dest* on the completion of the demand request which is simultaneously issued to both *lop* and *rop* :

eval.d (lop) || eval.s (rop) ;

when lop.tag = full & rop.tag = complete: dest $\leftarrow$ lop

The *with* instruction is useful when we want to explicitly demand two values where we are interested in only the first value and completion of computation for the second value. This makes the computed second value readily available to future demands for the value and thus can be used to shorten the critical path for other demands.

5.3.2 ***THEN*** Synchronization Instruction.

The *then* instruction is defined as *THEN lop, rop*. This instruction acts as a synchronization fork instruction by assigning the content of *lop* to *dest* on completion of the demand request for both *lop* and *rop*. The demand for *lop* and *rop* is an orderly demand sequence where 'lop' is demanded first and 'rop' is demanded next :

```
eval.d (lop) ;  
when lop.tag = full :  
eval.s (rop) ;  
when rop.tag = complete: dest ← lop
```

The *then* instruction is useful when there is a need for an ordered demand between instructions. For example in loops it will be useful to know the loop carried value is consumed before the instance of an iteration is freed.

5.3.3 ***EITHER*** Synchronization Instruction.

The *either* instruction is defined as *EITHER lop, rop*. This instruction acts as a synchronization fork instruction by assigning the content of *lop* or *rop* to *dest* on completion of the demand request for either one of the two values, i.e., *lop* or *rop*. *dest* gets the result of the first completed request and the value generated by the second request is ignored.

For demanding signal :

```
eval.s (lop) || eval.s (rop) ;
```

```
when lop.tag = complete: dest.tag ← complete
```

```
when rop.tag = complete: dest.tag ← complete
```

For demanding value :

```
eval.d (lop) || eval.d (rop) ;
```

```
when lop.tag = full: dest.tag ← full
```

```
when rop.tag = full: dest.tag ← full.
```

The *either* instruction is useful when it is required to demand two instructions but can consume whichever value is received first. This is useful in an instance such as dynamically unrolling of loops or an event to trigger a side effect.

5.4 Environment, Frames and Mapping of Frames

As defined previously an *Environment* is a sequence of code belonging to a functional instance, the code before and after a given loop, the loop body, or a dynamic instance of a loop iteration. An environment can contain an arbitrary number of instructions that is mapped to one or more fixed size frames by the compiler. Figure 5.5 illustrates

an example in which an environment has been mapped to a single frame with instructions $I_1, I_2, \dots, I_{64}$ and DM stores computed scalar values of executed instructions or partially computed values represented as $D_1, D_2, \dots, D_{64}$ by those instructions.

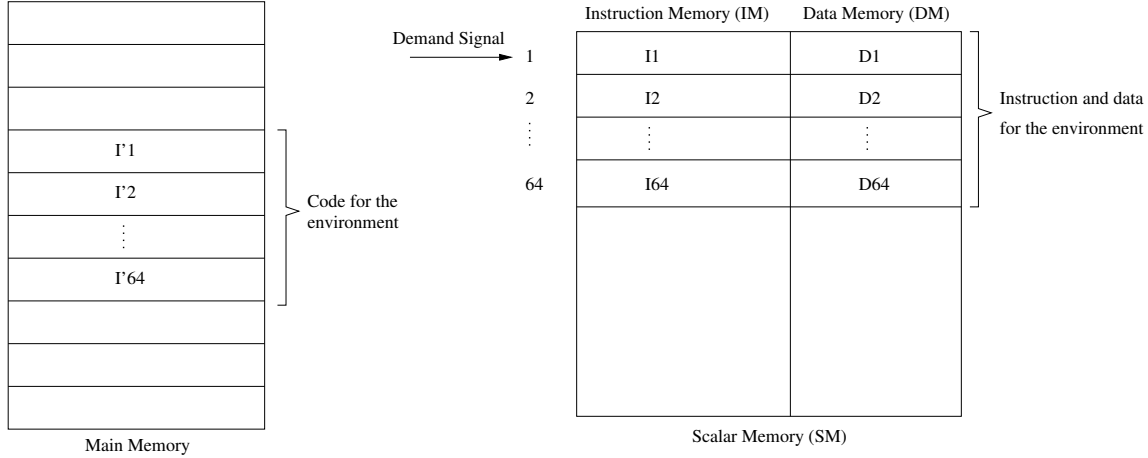


Figure 5.5: An environment mapped to a single frame in frame memory

A *frame* is allocated in SM by using an explicit instruction *newf* whose syntax is *newf* $\langle label, arg_blk_src, arg_blk_dest \rangle$. In this instruction, the *label* is an instruction address pointing at the beginning of the code belonging to that frame. The *arg_blk_src* field specifies the location of the argument block in the current frame as an offset. The *arg_blk_dest* field specifies the offset in the new frame where the address of argument block is stored on creation of the new frame. The *newf* instruction when demanded checks to see if it already has executed. If it did not, it creates a new frame and writes the start address of the argument block at the indicated location in the created frame. Creation of new frame copies *size* many instructions from main memory starting at *label* to IM of a new frame.

It is possible to introduce an additional *tag* field which is a constant field specifying whether a new frame should be created or be permitted to link to an existing frame. We can then use a hardware structure to store the information for IM - DM linked pairs created using a *newf* instruction. In this case, *newf* instruction with a tag bit *set* will return the address of an existing frame, instead of allocating a new one. A search in this hardware structure can be performed using the supplied instruction address to link to an existing frame.

As previously stated, an environment is distributed across multiple frames when the number of instructions in the environment exceeds frame size. Figure 5.6 illustrates an *environment* of size 84. For a *frame size* of 64 locations, the environment when allocated, will be allocated as two frames of size 64. The second frame can be allocated for a smaller size but for simplicity and efficiency of implementation, frames should possibly be allocated in fixed sizes. A demand for a value at a location in frame 0 of the environment will lead to the allocation of a frame in SM. During the allocation, the instructions $I'_1, I'_2, \dots, I'_{64}$ of frame 0 of the environment are copied from the main memory to IM. Another reference to a location in frame 1 of the same environment will lead to the allocation of another frame in SM and copying of the instructions $I'_{65}, I'_{66}, \dots, I'_{84}$ from the main memory to IM. Locations 85 to 128 are filled with NOPs. As explained in Section 5.1, we use displacement (base + offset) addressing mode in order to refer to a value from another frame location. In the rest of this dissertation we use the syntax for displacement addressing mode as *offset(base)*, where the *base*

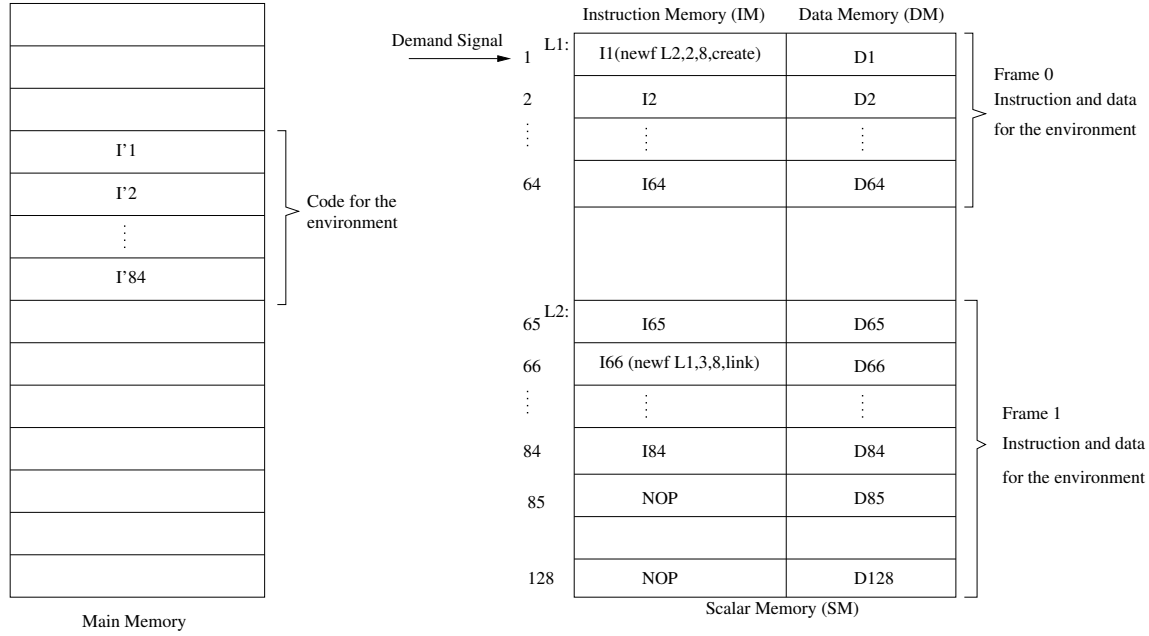


Figure 5.6: An environment with multiple frame representation in frame memory

is a location in the current frame containing a pointer to the start (*location 0*) of the target frame and the *offset* is the displacement within that frame from *location 0*. A value from a different frame can be demanded by using only the displacement addressing mode.

As seen in Figure 5.6, an instruction in frame 0 of a given environment can access any location in frame 1 of the same environment by using the *newf* instruction located at *location 0* of its frame which creates frame 1 for that environment. Similarly, an instruction in frame 1 of the environment can access any location in frame 0 of the same environment by using the *newf* instruction located at *location 1* of its frame which creates a link to an existing frame 0.

5.5 Deallocating Frames

The frames allocated by the *newf* instruction need to be deallocated once the use of those frames is complete. We introduce the delete frame *delf* instruction which frees a frame. The syntax for *delf* instruction is *delf* $\langle left_{operand}, right_{operand}, predicate_{operand} \rangle$. The $left_{operand}$ demands the necessary computation in the procedure and the $right_{operand}$ is used to locate the frame to be freed. The $predicate_{operand}$ can be used to embed additional control decisions. For instance, it can be set when it is safe to free a previous loop iteration in a loop. A frame must be deallocated only after all output values of the frame are read.

Typically, the call to a frame location in the *main* procedure starts with a demand for frame deallocation instruction. The frame deallocation demands the allocation of the frame and the output value from the frame to be freed. We illustrate environment allocation and deallocation with an example as shown in Figure 5.7. The demand starts with a demand for value of d in the *main* procedure which is the deallocate frame instruction. The deallocate frame instruction demands P which is the output value from the called frame and $q(z)$ which will return the address of the frame to be deallocated, which is available at offset q of *foo* environment. The instruction at location z will lead to the creation of a new frame for the *foo* environment. The evaluation of instruction at location P will lead to a demand of value at location a

of the *foo* environment. The value of *a* when available is returned to *P* which is the output value from the *foo* environment. The value of instruction at location *q* of the *foo* environment, when available is returned to *d*, which is the address of the frame to be deleted. When both the value of *P* and *q* are returned to *d*, the *delf* instruction has both of its operands and can now deallocate the frame for the *foo* environment.

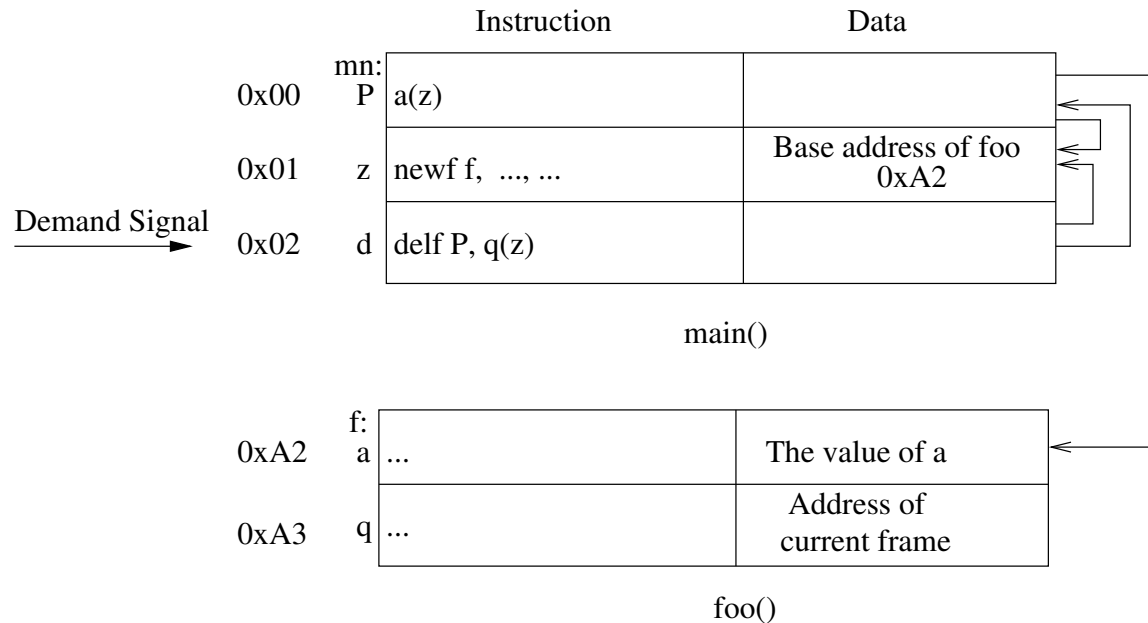


Figure 5.7: Environment deallocation in DDE

5.6 Passing Arguments to Functions

Arguments to called functions are grouped into an *argument block* such that instructions computing an argument value are in consecutive locations in the procedure argument order. For example, for the call *foo(a,b)*, the two instructions *a=...* and

$b=...$ which compute the argument values are grouped together and form an argument block. The argument values now can be demanded by any instruction from another environment by using a pointer to the argument block through the displacement addressing mode. The callee and caller environments can be separately compiled. Therefore, a function environment should be able to consume different arguments used at different call sites. This may necessitate the use of copy instructions to generate the proper argument order.

5.7 Procedure Calls

An example procedure call is shown in Figure 5.8. For simplicity, we do not include the *delf* instruction in our example. The execution begins upon receiving a demand for value at location d of *main* procedure. This demand in turn triggers the demand of the value at location z in *main* procedure. Being the first demand for z , the value of z is not available and the *newf* instruction is evaluated. The execution of *newf* instruction leads to the creation of an environment for *foo* and a pointer to the first instruction of the argument block *ab* is stored at the supplied offset n in the frame created for *foo* environment.

Once the environment is created, the *newf* instruction demands the return value of that function. The return value in this example is produced by the instruction at

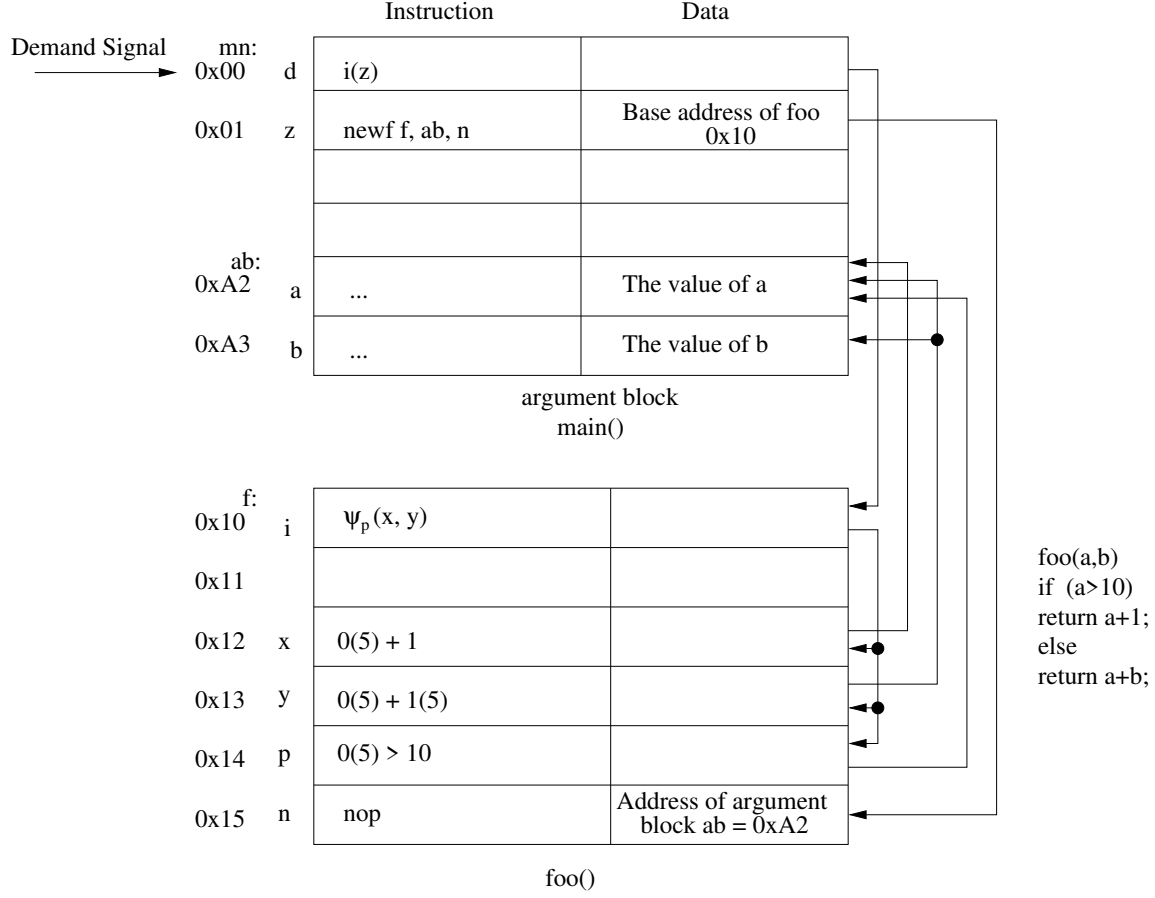


Figure 5.8: Procedure call

location i of the `foo` environment. Demanding the return value at location i results in the evaluation of $\psi_p(x, y)$. The ψ instruction evaluates the predicate $p = 0(5) > 10$. The value of argument a is read by using the base address stored at location n of `foo` environment with an offset zero, which is the address of argument a . Once the value of a is available, the value of predicated p is computed. Depending on the value of predicate p either value of x or value of y will be demanded. When the demanded value return, the result of ψ is computed and returned to d .

Procedure calls in DDE selectively demand only the computation which is required for the execution as opposed to conventional procedure calls in an imperative programming language such as *C*. In a conventional procedure call, both the value of argument *a* and *b* are forwarded to *foo* without taking into account of whether these arguments will be used in *foo*. In our example, depending on the value of predicate *p* the value of *b* will be demanded only if the predicate *p* is false.

A new function call to the *foo* environment which uses different argument values such as *g* and *h* can be performed by passing the instruction address of a second argument block, say *gh*. The instruction for the new call will be another *newf* instruction which will look like *newf* $\langle foo, gh, offset \rangle$.

5.8 Memory Ordering

The memory dependencies which are not resolved statically at compile time are represented using predicates. Let's see through an example how predicated memory dependencies can be used to order the accesses to memory. In our example, the addresses of *sw z*, *sw y*, *lw x* are not known at compile time. It is possible that the two store and the load instructions can reference the same memory location. The execution order between these memory instruction can be maintained by using predicates to represent dependencies between them.

	Address demanded/ present		Predicate available
a	1	sw z, mem, true	
b	1	sw y, mem, a	
c	1	lw x, mem, b	

Instruction Memory
Data Memory

Scalar Memory

Figure 5.9: Memory dependencies representation using predicate

The memory dependencies represented using predicate is illustrated with an example as seen in Figure 5.9. The *lw x* uses predicate *b* to maintain dependency with *sw y*. The *sw y* uses the predicate *a* to maintain dependency with *sw z*.

We illustrate the execution of memory instruction and the resolution of their dynamic dependencies in Figure 5.10. The demand for *lw x* at location *c* triggers the computation of address for load and the demand for predicate *b*. The *sw y* demands its operands and predicate. This leads to the computation of the address of *sw y*. The operands for *sw y* leads to the return of the value which will be committed by *sw y* to memory later when the store is ready to commit when its predicate is available. This leads to the demand for location *a*. The *sw z* demands its operands and predicates.

	Address demanded/ present		Predicate available
a	1	sw z, mem, true	true
b	1	sw y, mem, a	a
c	1	lw x, mem, b	b

Instruction Memory
Data Memory

Scalar Memory

Figure 5.10: Memory dependencies predicate available at run time

It computes its address and demands its value. It has its predicate true and is ready to store the value as soon as it has its address and value available. When *sw z* stores its value to the memory, the predicate *a* is made available to *sw y*. The *sw y* can now go ahead and store the value to memory and then make the predicate *b* available to *lw x*. The *lw x* can now go ahead and load the value from the memory.

5.9 Representation of Loops

In DDE, loops are treated as tail recursive procedure calls with a modification that only the last procedure call in the chain returns the computed value directly to the

very first called procedure. The loop iteration in DDE are executed forward and the execution within a loop iteration progresses in demand driven manner.

The environment for a dynamic instance of a loop iteration is allocated dynamically using a *newf* instruction and can be freed when the use of the iteration is completed. The necessary instructions in the loop iteration environment will be triggered and executed to compute information such as loop carried values or internal iteration results. A single demand for a value in an loop iteration environment is received from another environment which initiates the evaluation of an instruction and triggers a chain of demands in the iteration environment. We refer to every instruction in a loop environment as a *node*. The exit node in an loop iteration environment is defined as the *initiator* node. The *initiator* node initiates the evaluation of an instruction and demands the *root* node which triggers a chain of demand for other *nodes* in the loop iteration environment. An instruction in DDE is capable of demanding up to two other instructions and an additional instruction if the current instruction has a predicated operand. The chain of demand starting from the *root* node expands by demanding up to two other instructions (*nodes*). This expansion can differ for every loop, but always starts from the *root* node. One such random expansion is shown in Figure 5.11. The demand for *nodes* initiated by the *initiator* node demands the *root* node which expands into a tree structure. This virtual tree structure conceptually shows how the demand sequence expand and propagates for a loop body starting from the *root* node. We refer to this virtual tree structure as *Loop Body Tree (LBT)*.

As the *root* node in an LBT converges triggering of all the computation in a loop to single *node*, we refer to this *root* node as *Root of Loop Body Tree (RoLBT)*.

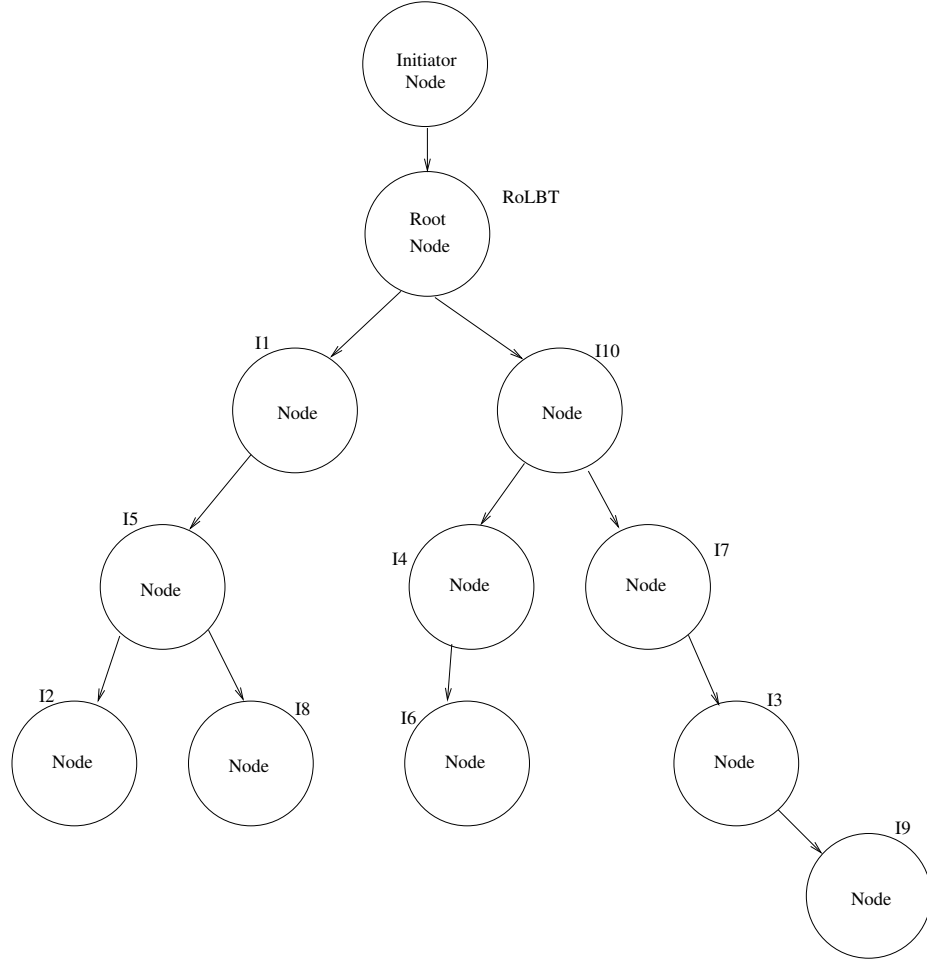


Figure 5.11: Virtual loop body tree

A loop in DDE is initiated by an *initiator* node which expands by demanding the RoLBT. Every iteration environment has its own RoLBT. A demand for an *initiator* node from an outside environment triggers the computation for the first iteration environment of the loop. In order to sequentially unroll the loop, the loop environment demands the value of the *initiator* node in a new instance of an iteration environment.

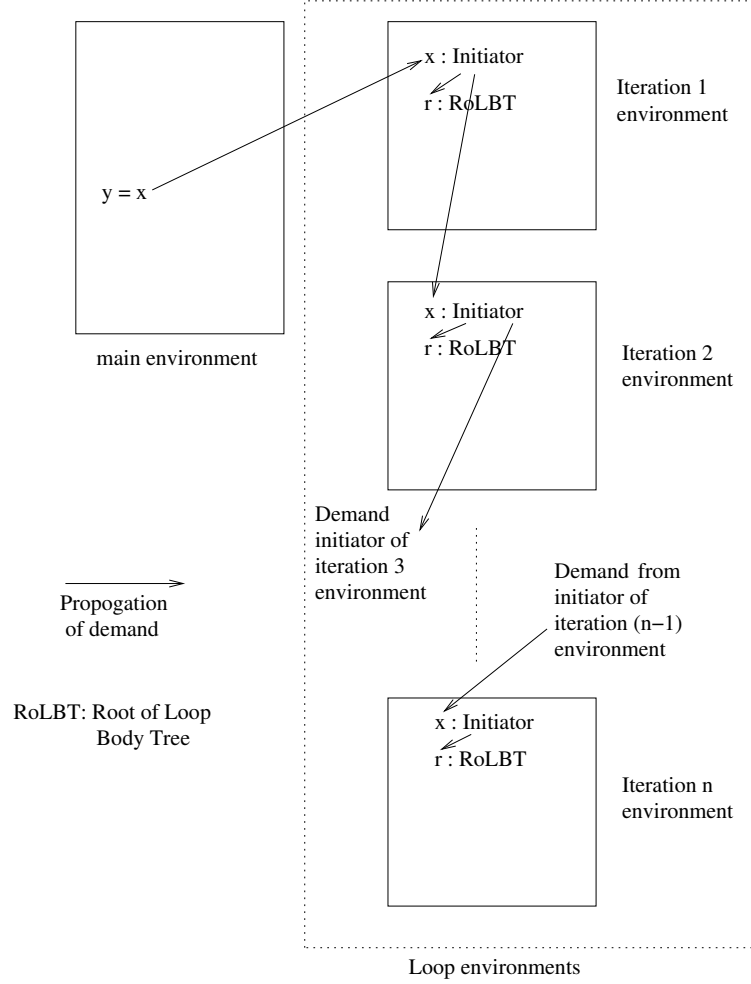


Figure 5.12: Loop unrolling by demanding initiator node

Figure 5.12 illustrates sequential loop unrolling in DDE by a demand for an *initiator* node. A Demand for *initiator* node at location x is received from a *main* procedure environment which triggers RoLBT triggering all the computation for loop iteration 1 environment. The *initiator* node of current iteration environment demands its own instance of *initiator* node in a new environment. The recursive demand of *initiator* node by itself in a new environment allows sequential unrolling of loops in DDE.

5.9.1 Sequential Unrolling of Loops

We first discuss sequential unrolling of loops in DDE without speculation. RoLBT converges triggering of all the computation in a loop to single *root* node and forms a tree structure. DDE tries to explore fine granularity at each iteration level by a demand place for an *initiator* node which demands RoLBT. DDE dynamically allocates an iteration environment and is self-contained in freeing the iteration environment when the use of the iteration is completed.

We introduce several new instructions, called *next*, *first* and *delf* to aid with the loop unrolling in DDE. The *next* instruction is used to implement FGSA η function using recursion. It's of the form *next lop, rop, pop*, where the *lop* argument corresponds to η argument, *pop* corresponds to η predicate and *rop* is used to recursively unroll the loop. Hence, the *next* instruction demands *lop* and *pop* at the same time just the way η does. If the *pop* is true, which indicates loop termination, then the instruction stores *lop* into its location and returns the value to the demanding instruction. Otherwise, it demands *rop*, i.e. demanding the RoLBT in a new environment. While demanding *rop* it also appends the return address of the demand as the address of the instruction which demanded *next* instead of its own return address. The *first* instruction demands all its operands but selectively uses only its first operand and is of the form *first lop, rop*. The *first* instruction simultaneously places a demand for both of its operands

lop and *rop* but waits and uses the value of only its first operand *lop* and discards the second operand *rop*. The *delf* instruction deletes/frees a frame in which it is contained. The *delf* instruction is of the form *delf lop, rop, pop*, where *lop* is used to demand the necessary computation, *rop* points at the address of the environment to be deleted/freed, and *pop* can be used to embed additional decision making information.

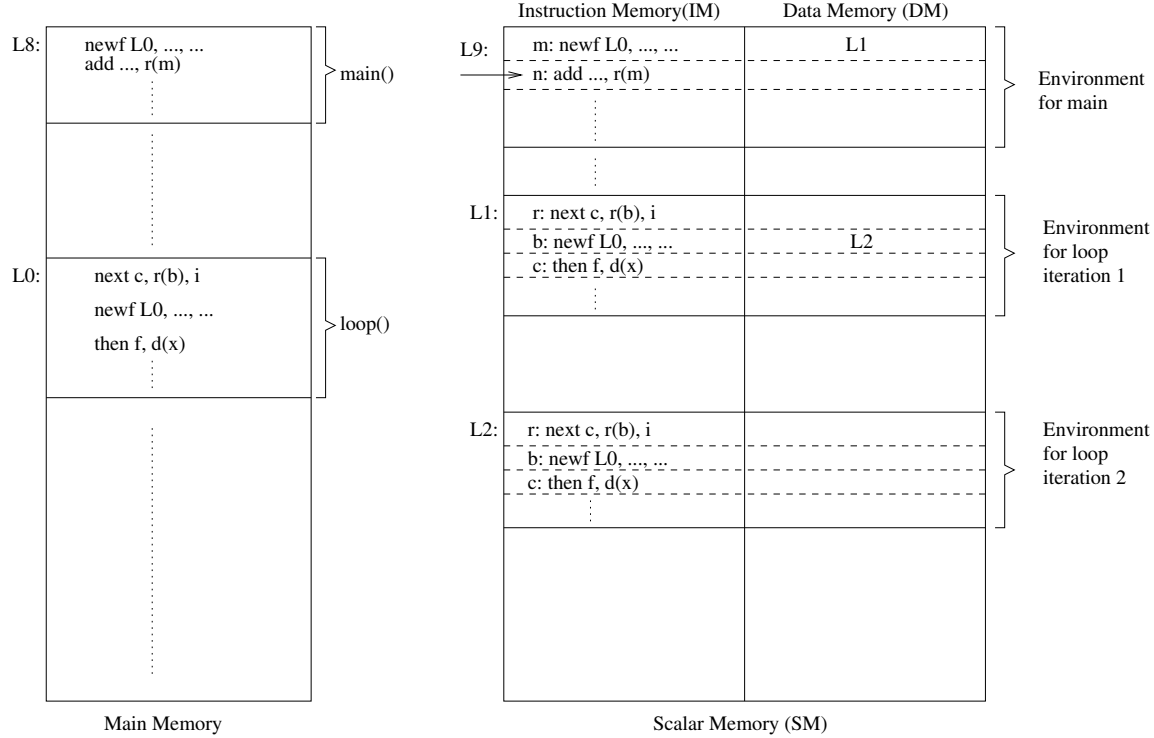


Figure 5.13: Sequential loop unrolling in DDE

A static instance of code in main memory unrolls into a dynamic instance of code in SM for execution of loops in DDE. We use Figure 5.13 to illustrate the sequential loop unrolling in DDE. The main memory holds the static instance of code. Creation and allocation of environments in SM for static environments of *main* procedure and loop from the main memory is done dynamically during execution. An instruction

from the *main* procedure demands the *initiator* node of the loop to trigger sequential loop unrolling. A dynamic instance of the *main* procedure and *iteration 1* of the loop is created in SM, represented by labels *L9* and *L1* respectively. The locations in SM are labeled as *r*, *b*, *c*, The instruction *next* at location *r* is the *initiator* node of the loop. The instruction *next* being the *initiator* node demands RoLBT, which is its operand *c*. RoLBT triggers a chain of computations for an iteration environment by demanding its operands. The instruction *next* on receiving its predicate *i* as false performs a call to itself in a new environment by demanding its operand *r(b)*. The operand *r(b)* uses displacement addressing mode and with help of a *newf* instruction at location *b*, demands a value at location *r* in a new iteration environment. Thus instruction *next* at location *r* performs a recursive call to itself in a new iteration environment leading to sequential unrolling of loops. Two such instance of iterations of the loop are represented using labels *L1* and *L2* in SM. As the *main* procedure has created loop *iteration 1* environment and the loop *iteration 1* has created loop *iteration 2* environment, the DM for the *main* procedure and loop *iteration 1* holds labels *L1* and *L2* of the created environment. The instruction at location *c* is RoLBT, which uses a *then* instruction. It triggers a *delf* instruction by demanding its operand *d(x)* carried from the previous environment after the availability of its operand *f*, indicating all loop values from the previous iteration have been used and it is safe to free the previous iteration environment.

We illustrate a loop example in Figure 5.14. Only the static instance of the code in

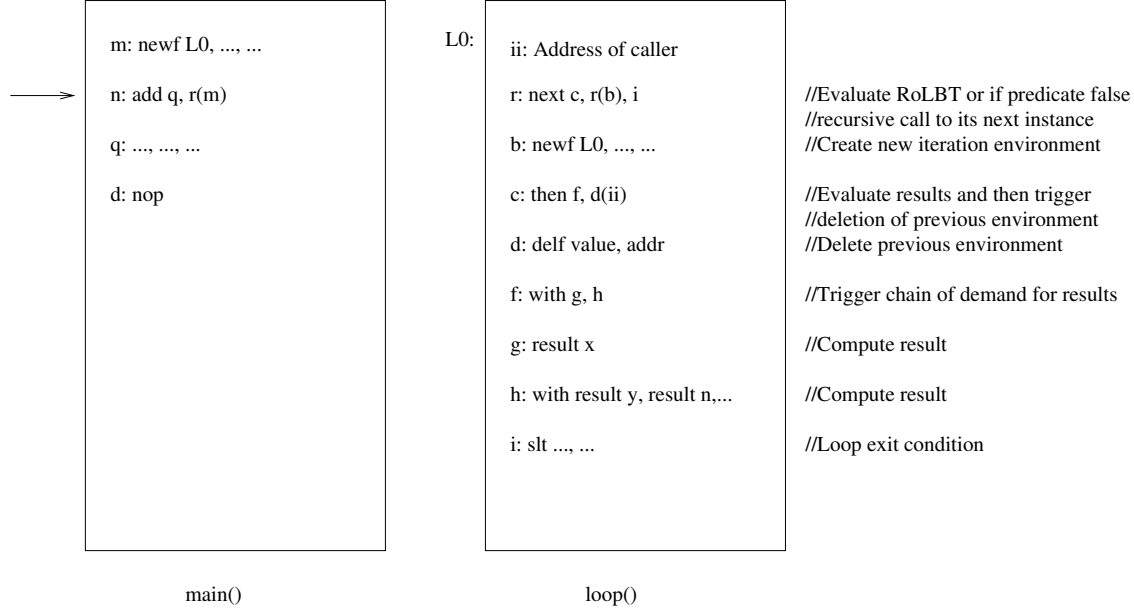


Figure 5.14: A static instance of loop code for DDE in main memory

main memory is used to illustrate the example. We show how a demand sequence propagates through the code and how environments are created and freed dynamically during the execution of the code. The execution starts when a demand for value n in *main* procedure is received. The value of n being unavailable triggers the demand for its operands q and $r(m)$, where q is a location in *main* procedure and $r(m)$ is a location in the loop at label r . A new iteration environment is dynamically created for iteration 1 of the loop by using *newf* instruction specified at location m . The demand received for location r for iteration 1 of the loop is the *initiator* node which demands RoLBT. RoLBT triggers the evaluation of all the computation in an iteration. The instruction r triggers the demand for the result via c and loop exit predicate i . The instruction c is the *then* instruction which demands the results in the iteration by demanding f . When the value of f is available, it is certain that all loop

carried dependencies have been used. It then triggers the demand sequence for the deletion of the previous environment by demanding $d(ii)$. This allows deleting/freeing individual iteration environments dynamically as soon as the use of an iteration is completed. The instruction c returns the result to r when it becomes available. The value of the predicate i is also returned to r . If the value of predicate returned is false, r will demand its own instance r from the next iteration. The instruction at location r is a *next* instruction that only waits for the value of the predicate to be returned before it demands its *rop*, if the value of predicate returned is false. A new iteration environment is created for the next iteration using a *newf* instruction specified at location b . Location r being the *initiator* node of the next iteration, it will trigger a chain of computation by demanding RoLBT in the next iteration. The process of dynamic creation of a new iteration environment for every iteration of a loop continues for n iterations by the instruction at location r , by recursively calling its own instance in the next environment until the predicate returned is true. In *iteration* n , the value of predicate i becomes true and the value of r from the n^{th} iteration is returned to the location n in *main* procedure.

In the next chapter, we present our work on the microarchitecture of single issue and multi-issue demand-driven processor designs.

Chapter 6

Microarchitecture

The microarchitecture describes the core-architecture of the demand driven processor. The core-architecture performs three major tasks for demand driven instruction execution: (1) An instruction is evaluated by a demand for its result; (2) The instruction is executed when it has all its available operands; (3) The generated result is then returned to all the consumer instructions which are waiting for the result.

This basic functionality of demand driven execution forms the basic building blocks of a demand driven processor. A demand driven processor can then be constructed using these functional blocks. The amount of parallelism that can be extracted is dependent on how these blocks are used and organized. The two primary questions are: (a) How are the processor internal blocks timed, and (b) How is the communication between

each of the blocks constructed?

Given how these questions are answered, it is possible to realize a demand driven processor as a simple in-order core, a pipelined implementation, a multi-issue pipelined processor, a multi-core processor and as well as a many core processor. We illustrate a pipelined implementation of a demand driven processor in this chapter. We further demonstrate a multi-issue pipelined implementation of our demand driven processor.

6.1 Pipeline Overview

The Demand Driven Execution paradigm functions based on a demand for the result of an instruction, leading to the evaluation and execution of additional instructions which are required to produce the result. Hence, we divide the DDE pipeline into three separate pipelines, namely the *evaluation pipeline*, the *execution pipeline* and the *send-back-and-commit pipeline*. The communication among these pipelines may be provided through the use of “tokens” implemented through message passing. For example, the evaluation pipeline may send a message to the execution pipeline embodying the operand of an instruction. An illustration of a general demand driven execution pipeline is shown in Figure 6.1. We refer to various points in Figure 6.1 to illustrate the functioning of various blocks in this section.

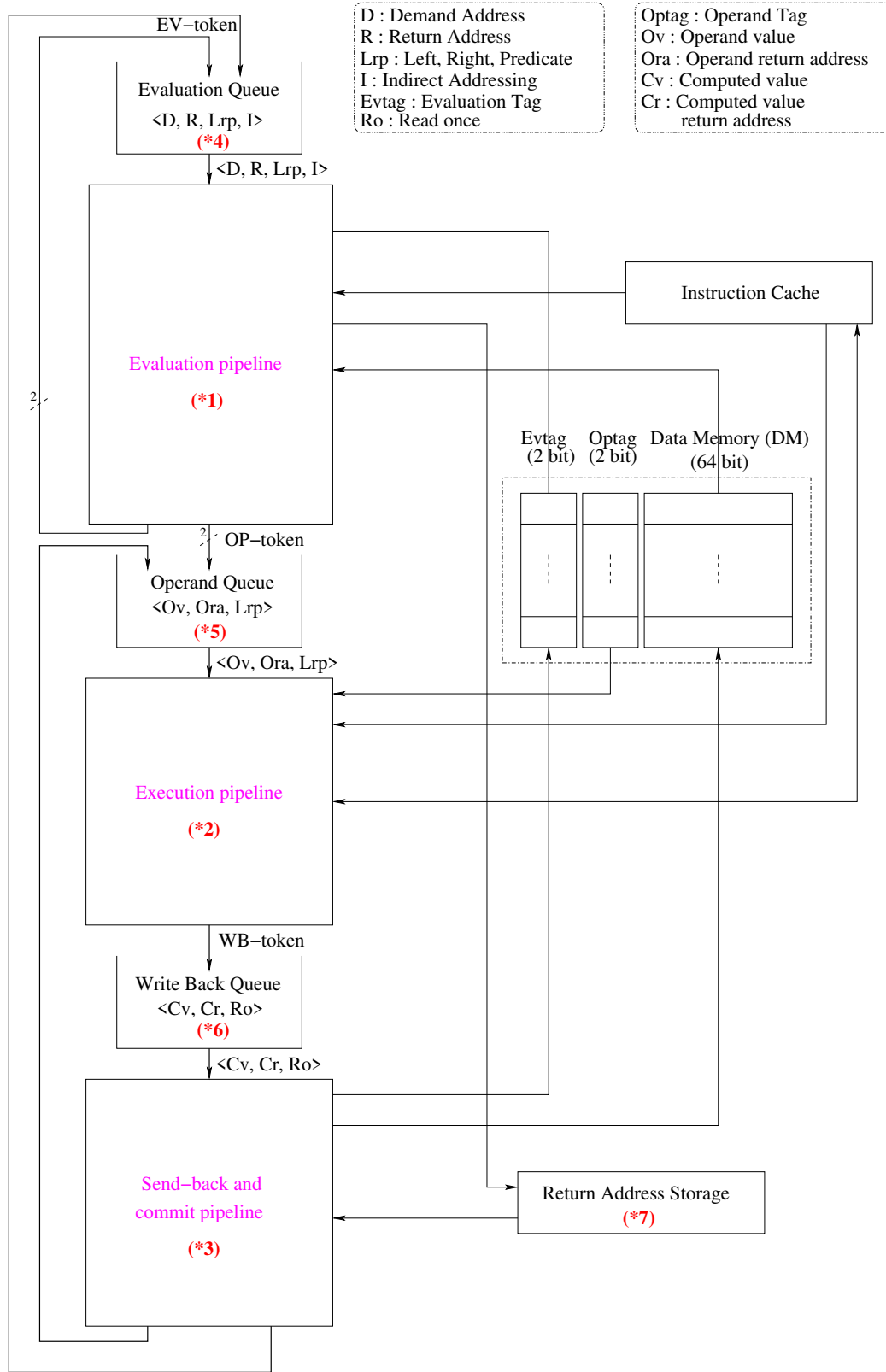


Figure 6.1: Demand driven execution pipeline overview

In this design, the evaluation pipeline (*1) facilitates the demand process. This process includes demanding the value at a location, issuing subsequent demands for the operands of the corresponding instructions if that value is not available, and storing the return address. If the value is available the value is sent instead to the execution pipeline.

The execution pipeline (*2) is responsible for computing the demanded value by executing the corresponding instruction. If the receipt of a data value does not enable the execution of a new instruction, then this instruction is still awaiting some other data value to be returned to it. The data value received from the evaluation pipeline is stored into the scalar memory while waiting for the other data or operand to arrive and the instruction at this location is effectively shelved until that time. This pipeline also commits store values to the memory.

The send-back-and-commit (*3) pipeline generates return tokens for the result value, returning it to all the addresses waiting for that value. As a result, new operand tokens may trigger new instructions to be executed.

Communication between the pipelines is one-way and it proceeds from evaluation to execution and then to send-back-and-commit pipeline. In a message-driven pipeline, the rate at which messages are generated and consumed may differ significantly. Therefore, we introduce queues at the beginning of each pipeline to buffer messages.

The queue at the beginning of the evaluation pipeline is referred to as the (*4) *evaluation queue* (*ev-queue*), and the queues in the other pipelines are named similarly. The incoming demand requests in the form of evaluation token (*EV-tokens*) are buffers by the ev-queue and are consumed every cycle.

An evaluation token $\langle D, R, Lrp, I \rangle$ consists of the location being demanded (D), the location of the requester along with its *port information* (R, Lrp) and demand information including a *indirect* demand (I). The destination of a returned value is referred to as a port and can be the left, or right operand of a dyadic instruction, or, the predicate operand of a predicated instruction. Similarly, an operand token (*OP-token*) $\langle Ov, Ora, Lrp \rangle$ consist of the operand value being returned (Ov) and the address and the port of the target instruction (Ora, Lrp). A Write-back token (*WB-token*) $\langle Cv, Cr, Ro \rangle$ consists of the computed value being returned (Cv) and the location to which it is being returned (Cr) along with the information if the value being returned is read-once (Ro). If the demanded location can be directly referenced and is in the vicinity of the returning instruction, it is referred to as a *direct demand*. If an address computation using a pointer value is necessary in order to demand a location, we refer to this demand as an *indirect demand*. Naturally, the pointer value in this case needs to be accessed using a direct demand.

A pipelined implementation is capable of handling one demand request every cycle and may issue up to two new demands by constructing and placing new *EV-tokens* back

into the *ev-queue*. If the demanded location already has the computed value available, then the data value is packaged into an *OP-token* and sent back to the requester by inserting this token into the (*5)*op-queue*. If the corresponding instruction loads only an immediate value, then the data value is packaged into an *WB-token* and sent back to the requester by inserting this token into the *write-back queue* (*6)(*wb-queue*). Similarly, the execution pipeline can consume one *OP-token* every cycle. This pipeline removes one *OP-token* from the operand queue and either executes the corresponding instruction or shelves the available data value if the instruction is not yet ready to execute. If the instruction is able to execute, then a new *WB-token* is generated and inserted into the *wb-queue*. Similarly the send-back-and-commit pipeline consumes a *WB-token* every cycle from the *wb-queue*. This pipeline will generate return tokens for all the consumers waiting for the computed value. Either an *OP-token* or an *EV-token* is generated using the information available for the waiting consumer. Multiple tokens needed by multiple requests may take multiple cycles depending on the number of consumers for the computed value and the organization of the send-back-and-commit pipeline.

6.2 Scalar Memory

In order to exploit a larger amount of parallelism, the demand driven processor needs to rapidly access a large amount of instructions and data. For our processor, we

Evtag (2 bit)	Optag (2 bit)	Instruction memory (IM) (64 bit)	Data Memory (DM) (64 bit)	Return link (1 bit)	In use (1 bit)
⋮	⋮	⋮	⋮	⋮	⋮

Figure 6.2: Scalar memory

envision a very fast memory which can store both instructions and data. We refer to this memory as the *Scalar Memory* (SM). As illustrated in Figure 6.2, SM is used to store scalar values with their states and the instructions to be executed. SM is divided into an *Instruction Memory* (IM) which stores the instructions, and a *Data Memory* (DM) to store the computed scalar values. It also provides additional fields to store a *tag*, and a *return link*. The return link can be used to distinguish the instance stating the end of the execution for a DDE machine. The *in-use* field specifies if the current location is valid and is usable. The tag field holds the information about the state of the data and its evaluation. This field is divided into two separate fields, one that keeps track of the evaluation state of an instruction and the second one keeps track of the execution state of an instruction. The former is called the *Evaluation tag* (*Evtag*). The latter is called the *Operand tag* (*Optag*). Table 6.1 and 6.2 summarize the states of the *Evtag* and *Optag* which are explained in the rest of this paragraph. If a data value is not available for an instruction being evaluated, then the state of

the *Evtag* is *Empty & Unlocked*. If the evaluation of the operands required for the generation of data is in progress, then the state in the *Evtag* is *Empty & Locked*. If the computed data value is available, then the state of the *Evtag* is *Full*. If the operand value required for the computation of the data is not available, then the state in the *Optag* is in *Empty*. For a dyadic instruction, if one of the operand values required for the computation of the data is available, then the state in the *Optag* is *Partial*. Similarly, for a predicated instruction, if the predicate operand value is available, then the state in the *Optag* is *Predicate-Partial*.

Empty & Unlocked	The data is not available
Empty & Locked	The data is in process of being computed
Full	The data is available

Table 6.1
States of EV-tag

Empty	The data is not available
Partial	The data is in process of being computed
Predicate-Partial	The data is available

Table 6.2
States of OP-tag

6.2.1 Return Address Storage

Return addresses can be stored using a linked list of all the addresses to which the computed value must be returned to. A pointer in SM, the *Return link* field, can serve as the head of the linked list. Additional entries need to be assigned to a dynamically allocated area, Return Address Storage (RAS) as seen in Figure 6.3. We need to

Return Address Frame (32 bit)	Return Address Displacement (32 bit)	Next link (32 bit)	Return port (2 bit)	Indirect (1 bit)	In use (1 bit)
⋮	⋮	⋮	⋮	⋮	⋮

Figure 6.3: Return address storage

expand *Return link* field in SM accordingly to accommodate the address space to access the RAS. Each RAS entry is nothing but a return token concatenated with a next-link and an in-use bit as shown in Figure 6.3.

6.2.2 Reservation Station Storage

An alternative approach to a linked list of return addresses is to use a Content-addressable memory (CAM) storage, similar to reservation stations in conventional architectures. In a CAM implementation, the CAM storage would have a layout identical to the RAS, with the exception that there would not be a need for the next-link field. Instead, a tag field that is the CAM key is used to search for all the entries requesting this data.

6.3 Pipeline Details

We present two demand driven pipeline implementations, one based on return queue storage, and another that is based on reservation station storage. These two microarchitecture implementations use the same pipeline functionality, the only difference being the storage used to store the return request.

As previously described, the DDE pipeline is composed of three pipelines and additional frame allocation stage. There are three types of tokens generated for the pipelines classified as: (1) *EV-token* for demands and for additional frame allocation stage; (2) *OP-token* for execution; (3) *WB-token* for committing values to SM and returns.

The evaluation pipeline, which is the first pipeline, facilitates handling of a demand request. Depending on the availability of the requested data value, different tokens are generated. An available data item or an instruction with no operands will immediately generate an *OP-token*, whereas an unavailable data value will demand the operands of the instructions at that location, hence generating new *EV-tokens*. Such is the case with indirect instructions. Instructions with only immediate operands generate a *WB-token*.

The second pipeline, the execution pipeline is responsible for the execution of an

instruction when all of its operands are available, temporarily storing of an operand when only one of two operands of a dyadic instruction or predicate operand for a predicate instruction is available, and committing store values after execution.

The third pipeline, namely the *send-back-and-commit pipeline*, is responsible for writing back the results to SM, and generating return tokens.

The frame allocation stage has its own queue where it buffers *EV-tokens* designed for frame creation. The frame allocation stage is designed for processing frame creation instructions and accessing the instruction cache. Each token embeds the necessary information to be consumed by the pipeline or the frame allocation stage.

6.3.1 Evaluation Pipeline using RAS

The *evaluation pipeline* is divided into four stages as shown in Figure 6.4. The major functionality of each of the stage is as follows. (1) *s_Eval* stage evaluates an instruction and makes a decision based on reading the necessary *Evtag* related to that instruction. (2) *s_Fetch* stage fetches the instruction from IM and a decision is taken based on the current state of the *Evtag*. An entry is made in return queue storage for the return address to return the value when it becomes available. The instruction will be allowed to proceed to the next stage if it is being evaluated for the first time. (3) *s_Decode* stage decodes the instruction and makes a decision about how to proceed with the

Let us look at a detailed flow of instructions in the *evaluation pipeline*. Demand requests are queued in the evaluation queue. A new *EV-token* can be taken out every cycle from the *ev-queue*. In the *s_Eval* stage the *Evtag* and data are read from the location in SM pointed by the demanded address. The token is passed to the *s_Fetch* stage of the pipeline or a new *OP-token* is generated according to the value of the *Evtag*. If the state of the *Evtag* is *Full*, then the data value is available and had become ready at an earlier stage. An *OP-token* is generated using the available data value, *return address* and the return port information. The token is then inserted into the *op-queue*.

In *s_Fetch*, if the state of the *Evtag* is *Empty & unlocked*, then the necessary instruction is read from IM and the *Evtag* is updated to become *Empty & locked*. A new link is created with the head of the linked list storing the return address, so the value can be returned when it is available. If the state of the *Evtag* is *Empty & locked*, then the demand for the operand of the current instruction has already been generated by another demand. Instead of generating another demand, the current token retires after appending its return address to the head of the return address linked list for the value being demanded.

If the instruction proceeds to *s_Decompile* stage, then it is decoded. Instructions with only an immediate value create new *WB-tokens*. This process essentially bypasses the *execution pipeline* and results in a direct insertion into *wb-queue*. If the instruction is

a new frame creation instruction, then the token is sent to the *frame allocation queue*. All other instruction types are sent to *s_Demand* stage.

s_Demand stage generates up to two new *EV-tokens* if the operand value is not available or if the demand address type is *indirect*. It creates up to two new *OP-tokens* when the operand value is available with a *direct* demand address, or, if the instruction has no operands.

6.3.2 Execution Pipeline

The execution pipeline is divided into three stages as shown in Figure 6.5. The major functionality of each of the stage is: (1) *Read_SM* stage reads a decoded instruction from IM along with *OP-token*. (2) *Pre_EX* stage makes a decision based on the *OP-token* and the available operand about how to proceed with the execution of the current instruction; (3) *EX* stage performs the actual execution. This pipeline has four types of execution units to handle simple arithmetic and logical instructions, load instructions, multiply and divide instructions, and store instructions.

Let us look at a detailed flow of instructions in the execution pipeline. An available *OP-token* is removed every cycle from the operand queue. In *Read_SM* stage, *Optag* and IM are read from the corresponding location in SM pointed by the return address. The instruction is passed to the next stage of the pipeline.

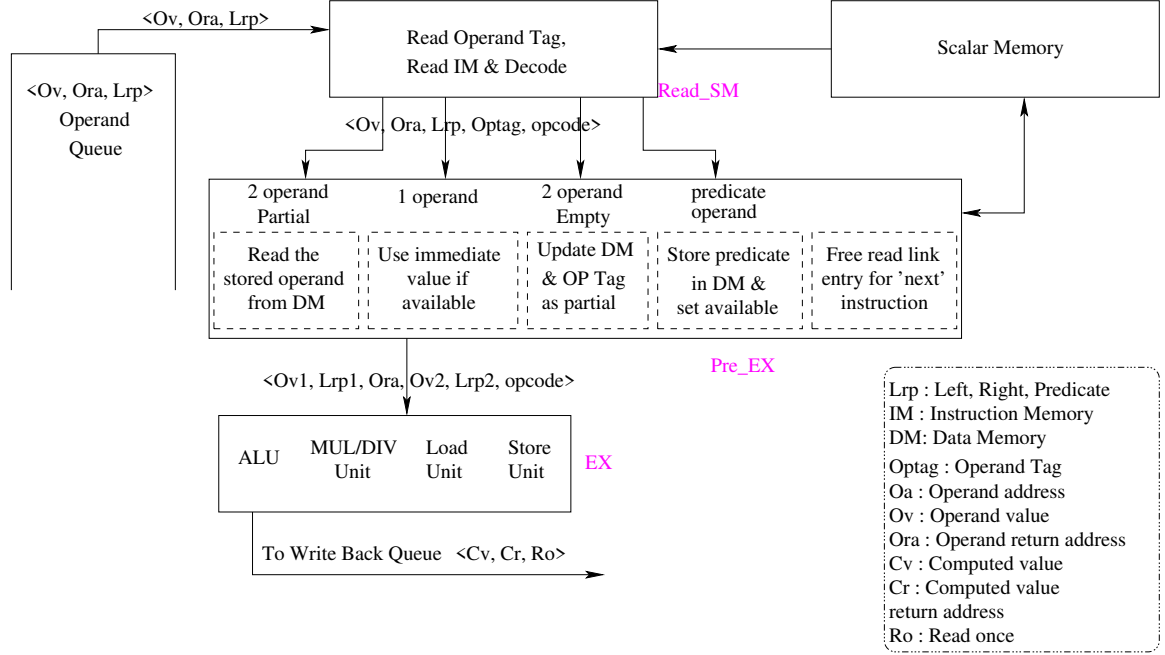


Figure 6.5: Execution pipeline

The *Pre_EX* stage then makes a decision about how to proceed with the execution. It accounts for the number of operands needed and how many operands are currently available based on information of decoded instruction and the *Optag* value. The following decisions are made: (a) For a uni-operand instruction, the token is sent to the execution unit; (b) For a dual operand instruction, if the state of the *Optag* is *Empty*, then this is the value of one of the operands and is stored in DM for later use. The *Optag* is updated to become *Partial*; (c) For a dual operand instruction, if the state of *Optag* is *Partial*, then the stored operand from DM is read and is sent to the execution unit along with the available value in *OP-token*; (d) If the available operand is a predicate value for a predicated instruction, then the predicate value is stored in DM and *Optag* is updated to *Predicate Partial*; (5) The entry of the return

address can be freed if required.

If the *EX* stage receives the required operands, then the instruction is allowed to execute. Once the value is computed, the execution unit sends the data to the *wb-queue* by generating a *WB-token*. If an instruction is a store instruction, then the value computed is also stored in the memory.

6.3.3 Send-back-and-commit Pipeline using RAS

The send-back-and-commit pipeline is divided into three stages as shown in Figure 6.6. The major functionality of each of the stage is: (1) *WB-update* stage writes back the available result in SM and updates the corresponding tag; (2) *RL* stage reads a linked list associated with the list of return address waiting for the available value; and (3) *Gen-token* stage generates an *EV-token* or an *OP-token* based on the available information.

Let us have a closer look at each of the stages of this pipeline. A *WB-token* can be read every cycle from the *wb-queue*. The *WB-update* stage commits the available result value from the token to DM and updates the *Evtag* to *Full*. The value is then sent to the *Read Link* stage.

The *Read Link* stage reads head of the linked list, which points to a list of frame

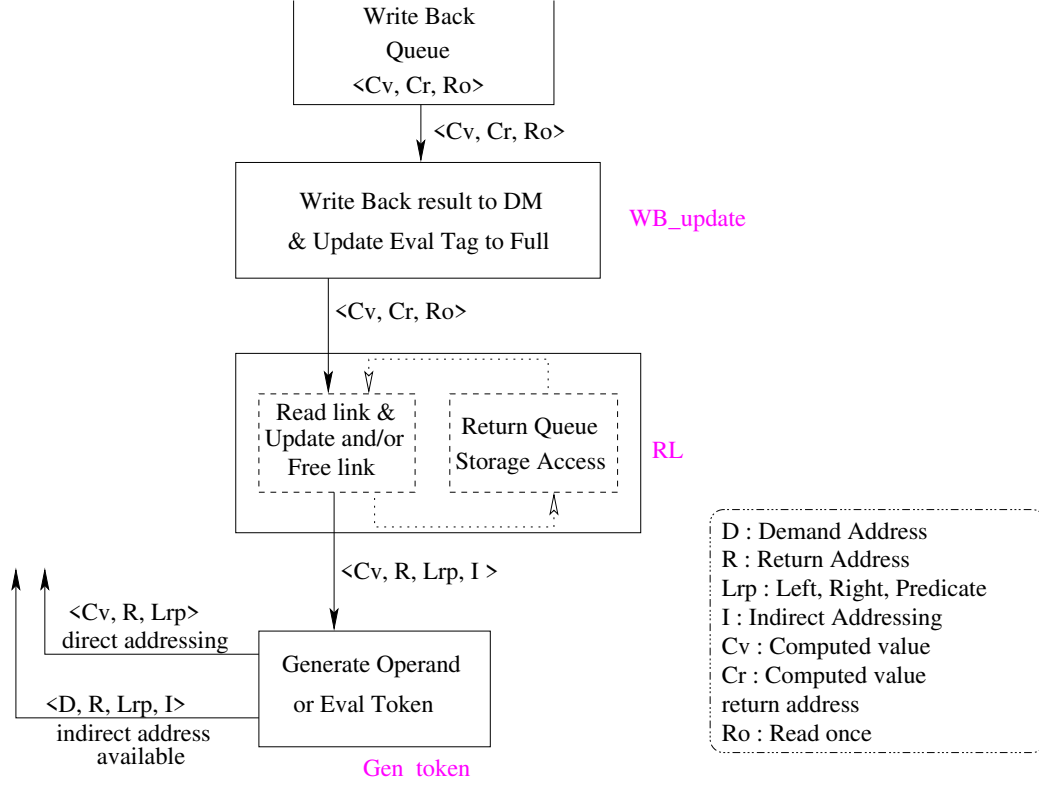


Figure 6.6: Send-back-and-commit pipeline using RAS

addresses to which the value needs to be returned. A linked list entry is consumed every cycle which provides the information for a token generation and is sent to the next stage. Finally the list is freed after the last entry in the list is read.

The *Gen.token* stage generates a new *OP-token* or a new *EV-token* by combining the result value and the address information available from the previous stage. The information about the indirect bit is used to decide whether an *OP-token* or *EV-token* needs to be created. If the indirect bit is false, it will lead to generation of a new *OP-token*. If the indirect bit is true it will lead to generation of a new *EV-token* by using the result value as the demand address.

6.3.4 Frame Allocation Stage

The frame allocation stage shown in Figure 6.7 is responsible for processing frame creation instructions. The stage accesses the instruction cache and reads instructions in burst mode. A *frame allocation queue* buffers tokens consisting of instruction designed for frame allocation. A token can be read from the frame allocation queue every cycle if the current frame allocation stage is not busy. The token read is decoded and broken down into individual elements. This information is used to access the instruction cache. The number of instructions equivalent to the frame size of the architecture are read from the instruction cache in burst mode. A free frame location in the scalar memory is consumed and the instructions read from the instruction cache are written in the IM segment of the scalar memory. The information available from the token is used to store the argument pointer at a specific location in the allocated frame. This information is stored in the specific data memory location and can be later used by other instructions in the frame to demand the required arguments from other frames.

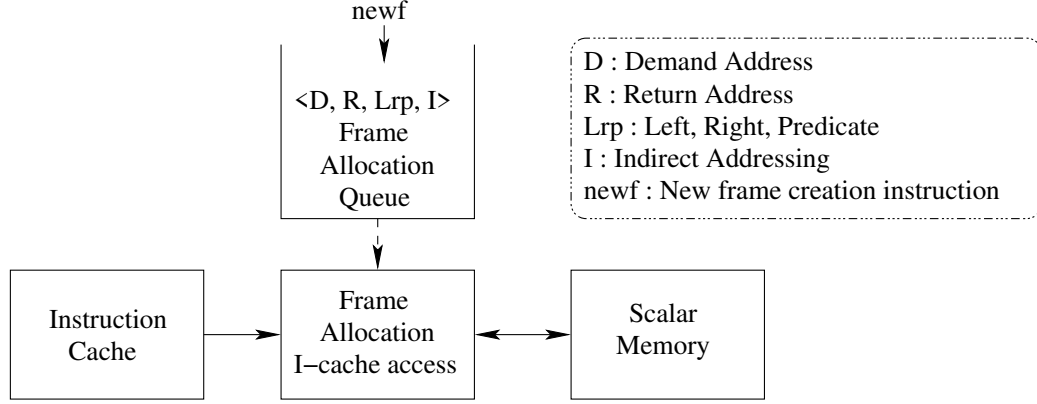


Figure 6.7: Frame allocation stage

6.3.5 Loop Level Frame Allocation Stage

In order to control the achievable parallelism in a demand driven machine, we implement separate pool of frames for function calls, loops and innermost level of nested loops. In this design, one pool of frame serves procedure calls which are simple functions as well as the outer level of nested loops. The design uses a second pool of frames assigned for the innermost loops in nested loops. This approach allows us to control the number of active procedures and active innermost loop iterations running at a given time on the machine. Loop level frame allocation stage has the same functionality as the frame allocation block. It has its own separate queue called *frame allocation queue level 1*. As of now, we only control the dynamic unrolling of innermost loop iterations using a separate pool of frames. It is also possible to control other levels of multilevel nested loops by having individual pool of frames for each level, and each of them having individual queues.

6.3.6 Evaluation Pipeline using CAM

The *evaluation pipeline* is divided into four stages as shown in Figure 6.8. This pipeline has a similar functionality as the evaluation pipeline based on the return queue storage described in Section 6.3.1. The only difference is in the *s_Fetch* stage as described below.

The *s_Fetch* stage fetches an instruction from IM and a decision is taken based on the current state of the *Evtag*. A entry is made in reservation station storage for the return address so as to return the value when available. The instruction will be allowed to proceed to the next stage if it is evaluated for the first time.

In *s_Fetch*, if the state of the *Evtag* is *Empty & unlocked*, then the necessary instruction is read from IM and the *Evtag* is updated to *Empty & locked*. A new entry is allocated in the reservation station storage and information of the demand address is stored as the tag field. Also information about return address, port and indirect address are stored. If the state of the *Evtag* is *Empty & locked*, then the demand for the operand of the current instruction has already been generated by another demand. The current instruction is retired after creating an entry in the reservation station storage. A new entry is allocated in the reservation station storage and the demand address is stored as the tag field. Also, the return address, the port and the indirect address are stored.

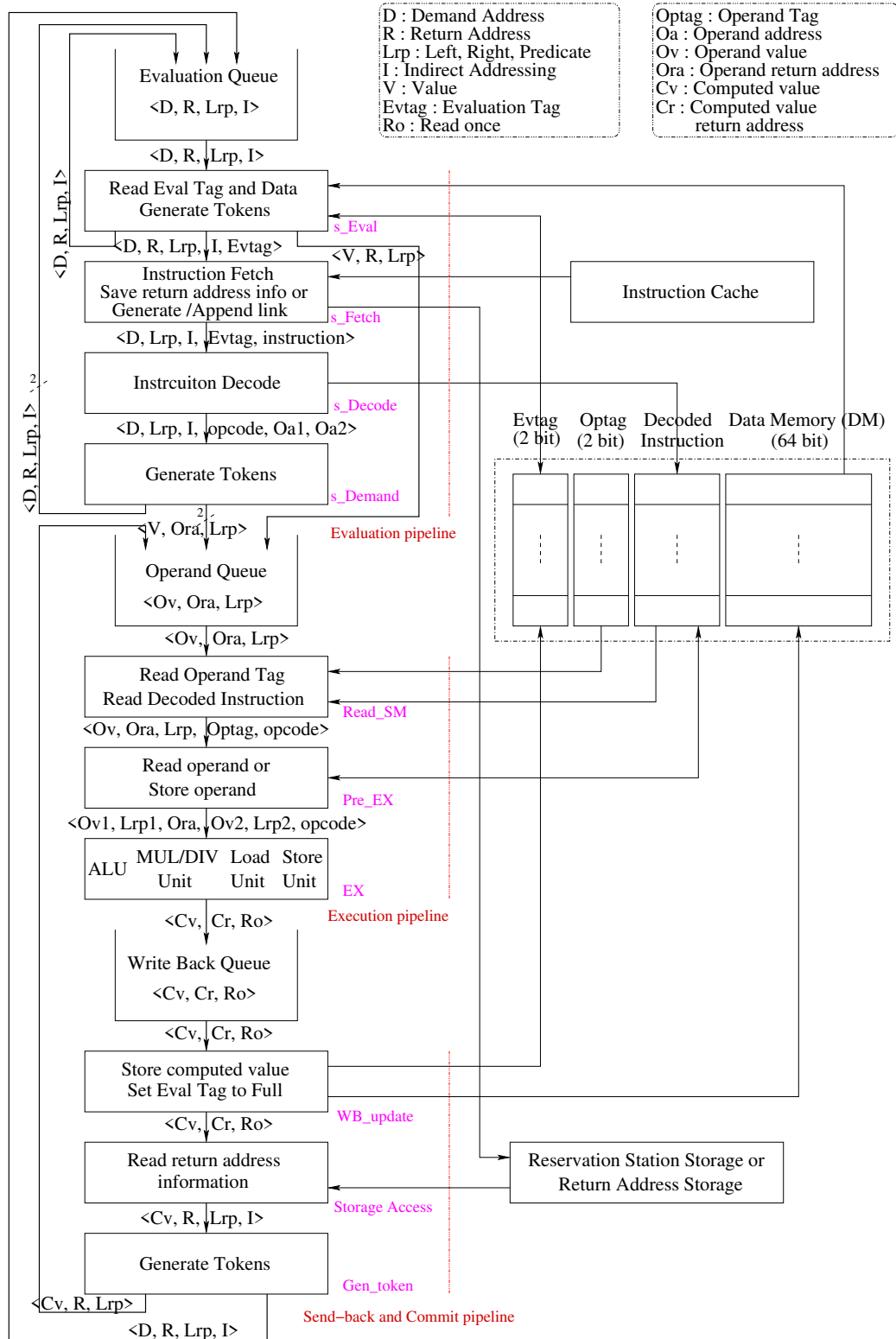


Figure 6.8: Demand driven execution pipeline

6.3.7 Send-back-and-commit Pipeline using CAM

The pipeline is divided into three stages as shown in Figure 6.8. The major functionality of each of the stage is: (1) The *WB_update* stage writes back the available result in SM and updates the corresponding tag; (2) The *Access_res* stage reads the return address by performing an associated search using a key for all the entries waiting for the available value; (3) The *Gen_token* stage generates an *EV-token* or an *OP-token* based on the available information.

Let us have a closer look at each of the stages of this pipeline. A *WB-token* can be read every cycle from the *wb-queue*. The *WB_update* stage commits the available result value from the token to DM and updates the *Evtag* to *Full*. The value is then sent to the *Access_res* stage.

The *Access_res* stage compares the tag field of all the locations in the reservation station storage in parallel. An entry matching the tag provides the information for the return address required for the generation of a new token. The matching entry is read and that location in the reservation station is freed. The read data along with the available value are sent to the next stage. Multiple entries can be read from the reservation station and sent to the next stage in parallel.

The *Gen_token* stage is capable of receiving multiple tokens in parallel and can generate multiple new *OP-tokens* or new *EV-tokens* in parallel. Each token is created by combining the result value and the address information available from the previous stage. The indirect bit is used to decide whether an *OP-token* or an *EV-token* needs to be created. If the indirect bit is false, it will lead to the generation of a new *OP-token*. If the indirect bit is true it will generate a new *EV-token* by using the result value as the demand address.

6.4 Multi-issue Pipelined Implementation

We expand the pipelined implementation based on reservation stations from Figure 6.8 to a DDE multi-issue pipeline. The DDE multi-issue pipeline is also composed of three pipelines and additional frame allocation stages. Each of the individual pipeline segments, *evaluation pipeline*, *execution pipeline*, *send-back-and-commit pipeline* can be “n-wide” and it is possible to have a different width for each of the three segments. The frame allocation stage can also be expanded to become “n-wide”.

In this design, the *evaluation pipeline* handles parallel demand requests and generates new tokens as required. If there are more than one parallel demand request for the same location in SM, then the stage schedules them by updating the *Evtag* for this demand using a methodology similar to the use of test-and-set instructions commonly

found in processors. This approach allows only one of the demand request for the same location to proceed to further computation if the computation was not initiated. The other demand requests for the same location will be retired after they have created entries in the reservation station storage.

In addition to the previously explained responsibilities of the *execution pipeline*, the execution pipeline also processes multiple operands in parallel. If there is more than one operand available for the same instruction, then they are consumed one at a time. If dyadic instructions operands were allowed to proceed in parallel it could lead to a deadlock. This is because, each operand would see itself as the first, causing it to be shelved. Hence in order to prevent such deadlocks, we adopt a simple mechanism of having multiple operand queues for the execution pipeline, which allows operands belonging to the same instruction to be inserted in the same queue, effectively serializing them.

The *send-back-and-commit pipeline* has the same responsibility as previously explained. The only extra feature is that it can process multiple *WB-tokens* in parallel. All *WB-tokens* are independent of each other, as they have a computed value for a unique instruction. The number of *WB-tokens* which can be processed in parallel is limited by the physical capabilities of the reservation station storage, in essence the number of ports of the reservation station.

We have presented the multi-issue demand-driven execution pipeline. Extending the

multi-issue pipeline to a multi-processing element (multi-PE) design can be easily be done by tagging messages by processing element identifier. Due to several restrictions in the ADL compiler, we have not evaluated multi-PE version of the microarchitecture.

In the next chapter we discuss the simulator infrastructure we developed to simulate our microarchitecture designs.

Chapter 7

Simulation of Design, Assembly Language Programming, Debugging and Results

We use the *Architecture Description Language (ADL)* framework designed by Önder et al. [17] to implement and evaluate our processor design. ADL is a domain-specific language which allows specification of instruction set architecture, microarchitecture, assembly language syntax and binary representation of a new architecture. A description written in ADL is compiled using the ADL compiler to automatically generate a cycle-accurate simulator. In addition to the simulator the compiler also generates an assembler and a disassembler. The cycle-accurate simulator respects timing at

the Register Transfer Level (RTL). ADL also has features for automatic generation of statistical data which can be used for performance analysis. ADL also supports a special set of commands to invoke a debugger and display monitoring information.

7.1 Description of Instruction Set Architecture in ADL

For our demand-driven processor, we developed a description of our instruction set architecture (ISA) in ADL. In the ISA, we define the instruction, its assembly language syntax and binary representation. Our ISA description encodes the instruction description for a control-flow processor as well as demand-driven processor. Control-flow instructions are 32-bits wide. Since we require more than 32-bits to encode all the information for our demand-driven machine, the demand-driven instruction set has a special instruction which act as an extension for all other demand-driven instructions to encode the remaining information which cannot fit into a single 32-bit instruction. As a result, our demand-driven instructions become 64-bits wide. An instruction encodes an opcode and the mode of execution as *control-flow* or *demand-driven*. The instruction also encodes additional information as required regarding its operands, immediate values, and floating-point instruction expansion. We define different attributes for an instruction as follows:

- a) Instruction class (*i_class*) as integer, floating-point, or multi-cycle integer;
- b) Number of instruction cycles (*i_cycles*) as single cycle or multi-cycle;
- c) Instruction type (*i_type*) as ALU, branch, load, or store;
- d) Designated execution unit (*exu*) as load unit, store unit, integer unit, floating-point add unit, or floating-point multiply unit;
- e) Branch condition category (*c_what*) as equal, not equal, greater than, greater than or equal to, less than, or less than or equal to;
- f) Branch type (*c_detail*) as conditional, unconditional, direct, indirect, direct link, or indirect link;
- g) Destination and operand type (*dest_type*, *lop_type*, *rop_type*) as float_register, integer_register, double_register, special_input, cpc_register, or lo_hi_register;
- h) Annotation (*l_annotation*) for performing instruction fusion and special features.

7.2 Description of Microarchitecture in ADL

A description of the micro-architecture for our demand-driven processor has been developed in ADL. For the micro-architecture, we define the demand-driven pipeline and a control-flow functional implementation. We define different artifacts such as registers, buffers, and structures to store and process instructions and data. All of these artifacts are supported by ADL as built-in types. The semantics of each processing stage has been specified using the schematics of the stage to generate the

RTL statements. For the demand-driven pipeline we define different processing stages. Each processing stage has its own context on which it can operate in a given clock cycle. N processing stages are cascaded to form an n -stage demand-driven pipeline. There is an ordering among pipeline stages. The context moves from a preceding processing stage to the next stage at the end of the clock cycle. The context from stage 1 of the pipeline gradually proceeds towards stage n , which will take n or more clock cycles. The context is processed by each stage before it is sent to the next stage. Sometimes a particular pipeline stage takes more than one cycle to operate on its context. This stalls the current stage and all preceding stages of the pipeline. It is possible to write a description for more than one pipeline where each pipeline independently operates.

The defined instruction set architecture and microarchitecture are used to implement four different simulators. 1) A functional implementation of the demand-driven processor using return queue storage structure; 2) A pipelined demand-driven processor using return queue storage; 3) A pipelined demand-driven processor using reservation station storage; 4) A multi-issue demand-driven pipeline using reservation station storage.

A generated description was compiled using the compiler designed for the ADL language by Önder et al. [17].

7.3 Compiling Imperative Programs and Assembly Representation

We use our defined assembly language syntax and binary representation to write assembly programs. The assembly programs have instructions defined for demand-driven execution in blocks of frame size. Each block is referenced using a label. In a single assembly generated program instructions for control-flow and demand-driven code can be mixed. The instructions for the demand-driven code are always composed in a pair, where the first instruction provides the opcode information for the functionality of the instruction and the second instruction acts as an extension to the first instruction.

Before we can code the program in assembly language, we take the imperative program written in C language and convert it into the internal representation of FGSA for imperative programs. We use the algorithm defined in Section 7.3.1 to convert the internal representation of FGSA for imperative programs to the functional form of FGSA. We then use this functional internal representation of FGSA to generate the demand-driven code in assembly. Since the compiler development progressed in parallel with the architecture development, we needed to hand-translate several programs to generate assembly code for our processor. Currently, the compiler development is being undertaken by Florida State University.

7.3.1 Conversion of Internal Representation of FGSA to Functional Form of FGSA

We define a loop environment as the set of instructions which are repeatedly executed until a condition is satisfied. We define the outer environment as the set of instructions that are not repeatedly executed, but provide some of the operands of instructions within the loop or consume output values from the loop.

7.3.1.1 Generation of Set for the Loop Environment

The algorithm we use to generate code is defined below. We define USE to be an array of sets where $USE[id]$ is the set of destination names of the set of instructions which use the FGSA name id . We define sets for the destinations of η function, ψ instructions and *read-once predicates*.

We first generate a USE set for every unique use of an FGSA name using Algorithm 1. We assume code generation which will have a single $USE[R]$ where R is a read-once predicate for all ψ instructions in a given loop. We use the $USE[R]$ to identify the occurrence of a loop for a program. Every element of R will generate a loop environment using Algorithm 2.

Data: S
Result: USE[S]
S: set of instructions of the program.
while $S \neq \emptyset$ **do**
 i = S.remove();
 USE[i.lop] = USE[i.lop] $\cup$ USE[i.dest];
 USE[i.rop] = USE[i.rop] $\cup$ USE[i.dest];
end

Algorithm 1: Generation of USE set for every unique use of an FGSA name

Data: USE[S], R
Result: elements in a loop environment
E: set of η function destinations.
P: set of ψ instruction destinations.
R: set of read-once predicates.
LE: set of elements in a loop environment.
let wl = worklist of instructions to be traversed
let wl = $\emptyset$
let LE = $\emptyset$
while $R \neq \emptyset$ **do**
 r = R.remove() //process a loop
 wl = {r}
 while $wl \neq \emptyset$ **do**
 op = wl.remove();
 foreach *use of* USE[op] **do**
 if *use of* USE[op] $\notin E$ **then**
 wl = wl $\cup$ USE[op];
 LE = LE $\cup$ op;
 end
 end
 end
end

Algorithm 2: Generation of set for a loop environment

7.3.1.2 Generation of Set for Outer Environment

We use Algorithm 3 to generate a set for the outer environment. We start with a call to a loop environment using an η instruction. An η instruction is inserted into the set of elements for the outer environment group. Using elements of the set for the loop environment, we identify all other instructions which will be grouped into a set for the outer environment. We traverse all instructions in the set for the loop environment to check if the use of each of its operands are defined in the set for the loop environment. If an operand is not defined in the set for the loop environment, the instruction generating the operand is placed in the set for the outer environment. Each instruction in the set for the outer environment checks if its operands are leaf nodes. If not, the instructions generating these operands are added to the set for the outer environment.

Data: loop environment
Result: outer environments
E: set of η function destinations.
LE: set of elements in a loop environment.
OE: set of elements in outer environment.
let worklist = set of all instruction to be included in set for outer environment
let worklist = $\emptyset$
while $E \neq \emptyset$ **do**
 e = E.remove() //process eta
 OE = OE $\cup$ e
 if *operand of e* $\in \{LE\}$ **then**
 foreach *instruction i in the LE* **do**
 foreach *operand o of instruction i* **do**
 if *Definition(o)* $\notin LE$ **then**
 worklist = worklist $\cup$ operand o of instruction i;
 end
 end
 end
 while *worklist* $\neq \emptyset$ **do**
 w = worklist.remove() ;
 OE = OE $\cup$ w;
 foreach *operand o of w* **do**
 if *operand o* $\neq$ *leaf node* **then**
 worklist = worklist $\cup$ operand o of w;
 end
 end
 end
 end
end

Algorithm 3: Generation of set for outer environment

7.4 Debugging in ADL for Demand-Driven Processor Model

The ADL generated disassembler is used for debugging and monitoring information in real time. The disassembler automatically disassembles the memory image and invokes two disassembled windows when an error condition is encountered. The first window shows the assembly instructions and highlights the current instruction whose address is in the instruction pointer. The second window gives a detailed view and the values in (1) all the registers in the current machine cycle; (2) all the pipelines and their stages and the context of individual stages showing the current instruction in it, if it has no context, or if there is a pipeline bubble; (3) the current address in the instruction pointer; (4) user-defined counters; (5) the machine cycle and the number of useful and stall cycles; and (6) the state of the minor machine cycle dividing machine cycles into a minor stage as prologue, intermission, or epilogue. It is also possible to explicitly invoke the disassembler by adding a special attribute when the user runs the program on the simulator or by having a specific invocation statement in the simulator itself.

Once the disassembled memory image is generated, it is possible to single step through machine cycles, including minor machine cycles. A GNU debugger (GDB) can be

attached to the process running the ADL disassembler. Using GDB along with the disassembler allows a user to (1) print the details of any variable or an RTL statement from the simulator; (2) print the errors with a pointer to the simulator code generating the error and probable cause; (3) add one or more watch points for variables or RTL statements; (4) single step in the generated simulator code; (5) change the assigned value for a variable or a register; and (6) many other things which GDB is capable of.

A user can add print statements in the micro-architecture description written for the simulator, or, in the written instruction set architecture. This approach can be used to print variables, RTL values, register values, memory addresses or values. This approach is more useful for debugging while developing the micro-architecture or the instruction set architecture in ADL.

We also add special print statements and code to generate a graphical layout of actual demand sequence among instructions. We print each demand request as a node when it arrives as a packet in the evaluation queue and generate a *dot* file as written in *DOT* (a graph description language) [21, 23]. The written description of these nodes are used to generate a graph using *graphviz* [22]. A visual graph provides a lot of information about the actual demand sequence and has been found to be very useful for debugging. We have also written a description to be used by the debugger which generated a graph for all the demand requests which are shelved, waiting for a

value. This description allows us to see a graphical representation of waiting demand sequences and accordingly allows us to see if a live lock or a dead lock is occurring due to the dynamic sequences of demands.

We also manually plot the demand graph of the static instance of the assembly code to verify and validate a hand-written assembly code for our processor. This approach helps to eliminate possible deadlocks. For example, it is important to see what is the critical path for a particular demand sequence, and how it is handled in a loop. This information can be used to visualize unrolling of loops. It is also useful to see the possible allocation of frames from the frame memory for a given sequence of code.

7.5 Performance Results

We use the Livermore loops written by Francis H. McMahon [12, 13, 14], which are programs written for parallel computers as our benchmark suite. These kernels were used to benchmark computers running scientific code at Lawrence Livermore National Laboratory. Each loop in Livermore loops is written for a mathematical kernel and measures the numerical computation for a spectrum of structures related to a processor. Some of the Livermore loops are vectorizable. We use these kernels to test the fine-grain instruction-level parallelism on our processor as we change the issue-width and the number of ports while keeping the remaining hardware parameters

constant. For each of the individual kernels, we record the time it takes to run it on a single-issue demand-driven processor and use it as our baseline for that kernel. We measure the time it takes to run each of these individual kernels with a different issue width. Kernel 1 and kernel 12 are vectorizable whereas kernel 3, kernel 5, and kernel 11 are non-vectorizable.

7.5.1 Handcoded Evaluated Benchmarks Experimental Parameters

1) Kernel 1 is a *hydrodynamics fragment* used for computations related to the study of liquids in motion. The kernel computes vector-vector, vector-scalar multiplication and vector-vector, vector-scalar addition. The code for the kernel is depicted in Figure 7.1.

```
for ( l=1 ; l<=100 ; l++ )
{
    for ( k=0 ; k<200 ; k++ )
    {
        x[k] = q + y[k]*( r*z[k+10] + t*z[k+11] );
    }
}
```

Figure 7.1: Livermore kernel 1

2) Kernel 3 is an *inner product* used in linear algebra, and adds more information to the collection of vectors. The kernel performs vector-vector multiplication and

vector-scalar addition operations. The code for the kernel is given in Figure 7.2.

```
for ( l=1 ; l<=100 ; l++ )
{
    q = 0;
    for ( k=0 ; k<1000 ; k++ )
    {
        q += z[k]*x[k];
    }
}
```

Figure 7.2: Livermore kernel 3

3) Kernel 5 is a *tri-diagonal elimination, below diagonal* used in numerical linear algebra to solve a system of linear equations by representing them as matrices. The kernel computes vector-vector multiplication and a vector-vector subtraction as shown in Figure 7.3.

```
for ( l=1 ; l<=100 ; l++ )
{
    for ( i=1 ; i<1000 ; i++ )
    {
        x[i] = z[i] * ( y[i] - x[i-1] );
    }
}
```

Figure 7.3: Livermore kernel 5

4) Kernel 11 is a *first sum* used in statistics. The kernel computes vector-vector addition as shown in Figure 7.4.

```

for ( l=1 ; l<=100 ; l++ )
{
    x[0] = y[0];
    for ( k=1 ; k<1200 ; k++ )
    {
        x[k] = x[k-1] + y[k];
    }
}

```

Figure 7.4: Livermore kernel 11

5) Kernel 12 is a *first difference* used in statistics. The kernel computes vector-vector subtraction as shown in Figure 7.5.

```

for ( l=1 ; l<=100 ; l++ )
{
    for ( k=0 ; k<200 ; k++ )
    {
        x[k] = y[k+1] - y[k];
    }
}

```

Figure 7.5: Livermore kernel 12

7.5.2 Evaluation

DDE is a new paradigm from architecture to program representation to its system software. Since the compiler infrastructure is still under development, our evaluation of the paradigm has been limited by the availability of code that can be tested on the simulated architecture. Currently, several handcoded kernels and compiler compiled kernels make up the benchmark base. Therefore, the evaluation in this section should be taken only as a preliminary evaluation of the paradigm. As well, these preliminary

data presented in the section indicate great promise and we still were able to demonstrate the strengths of the paradigm by comparing it with simple pipelined MIPS processors as well as a very idealized superscalar processor model.

7.5.2.1 Scalability of DDE Paradigm

The first set of experiments study the scalability of the architecture itself. For this purpose, we first evaluate the architecture using a model which can process one token per clock cycle and then vary the width of the architecture using the single token architecture as the baseline. For every kernel, time taken to run it on single token architecture is taken as a baseline 1 for that kernel. We then measure the speedup in terms of execution time for individual kernels as we vary the width of the architecture.

As mentioned earlier we have a separate pool of frames to control the achievable parallelism. One pool of frames is used for simple functions as well as the outer levels of nested loops and a second pool of frames is used for the innermost level of nested loops. Figure 7.6 illustrates that a maximum pool of 16 frames were available for simple functions as well as the outer level of nested loops. A maximum pool of 64 frames were made available for the innermost loop iterations. This pool is used to dynamically schedule innermost level of nested loops as per the availability of frames. As the figure illustrates, the performance flattens around 16 tokens due to lack of further loop-level parallelism to utilize the machine's capacity.

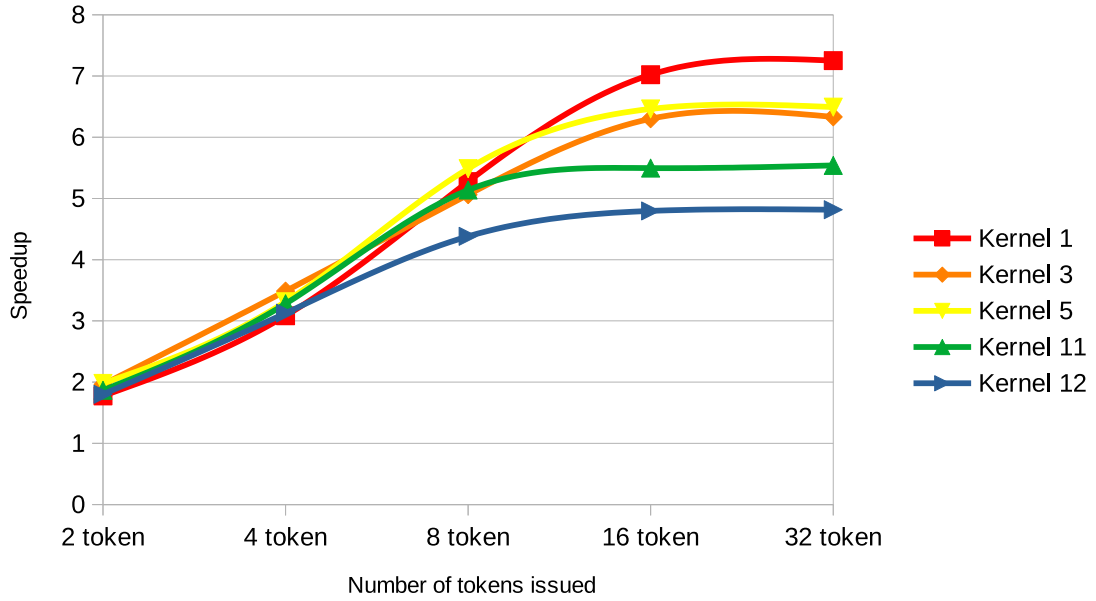


Figure 7.6: 1 token dde processor as base with max 16 active outer loop iteration and procedure with max 64 inner loop iterations

Figure 7.7 illustrates a maximum pool of 64 frames were available for simple functions as well as the outer level of nested loops. The innermost poolsize was set at 128 frames. As it can be seen, there is a significant performance gain compared to running the same set of benchmarks which were throttled using a maximum pool of 16 frames for simple functions as well as the outer level of nested loops. These experiments clearly show that increasing the pool size allows the processor to dynamically spawn more outer level of nested loops as per the availability of frames and the paradigm can exploit the additional available parallelism. Future designs need to concentrate on making a large number of frames available, which can only be done using multiple processing elements due to the large number of ports that will be necessary, if only the width of the architecture is increased.

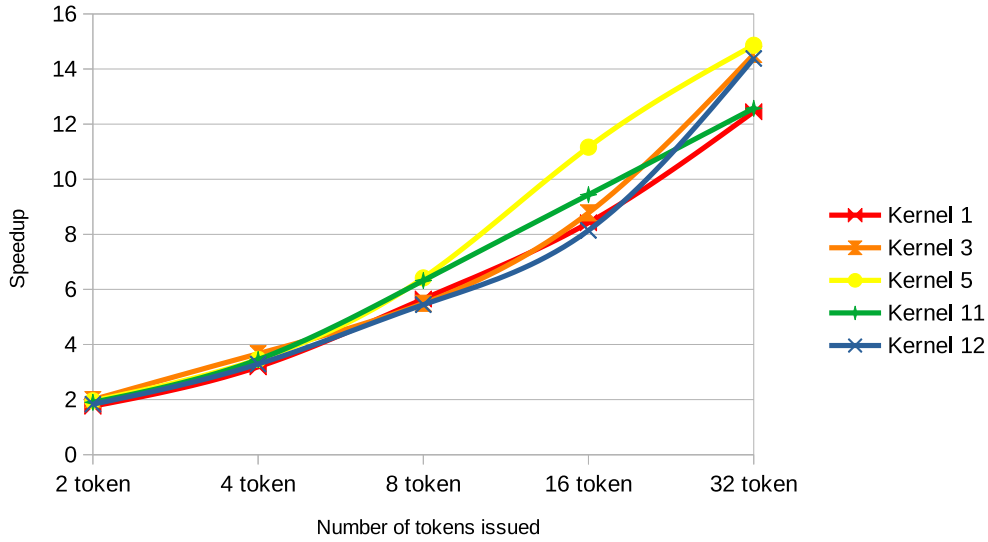


Figure 7.7: 1 token dde processor as base with max 64 active outer loop iteration and procedure with max 128 inner loop iterations

7.5.2.2 Control-flow Single Issue versus DDE

We compare our runs on the demand-driven processor against running the same set of benchmarks on a standard five stage MIPS pipeline with conventional internal data forwarding as shown in Figure 7.8. The MIPS pipeline uses a gshare branch predictor, which is a two level correlating branch predictor with global history sharing, along with a pattern history table. In contrast, the demand-driven processor does not use branch predictions, hence is a non-speculative processor. For our processor a maximum pool of 16 frames were available for simple functions as well as the outer level of nested loops. A maximum pool of 64 frames were made available for innermost loop iterations. Similar to our comparison of DDE with itself, DDE this time easily

outperforms a single issue MIPS processor and can potentially define high speed-ups without a need to develop parallel programs. The performance reaches a factor of four with a 16 token architecture and flattens.

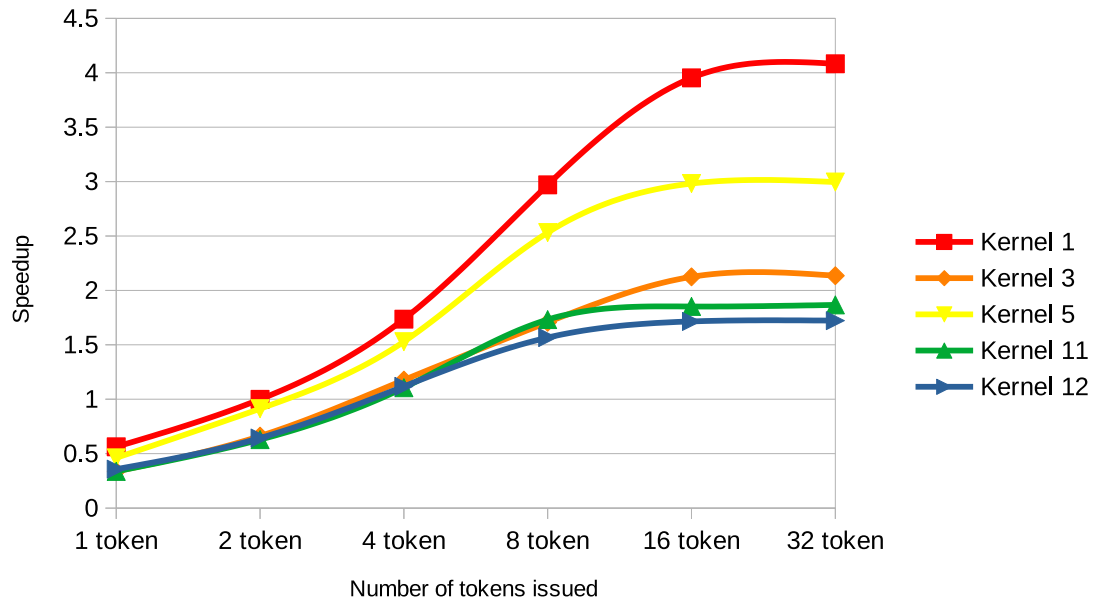


Figure 7.8: MIPS pipelined processor as base with max 16 active outer loop iteration and procedure with max 64 inner loop iterations

Figure 7.9 illustrates a run against the baseline of the standard five-stage MIPS pipeline with conventional internal data forwarding using a gshare branch predictor. For our processor a maximum pool of 64 frames were available for simple functions as well as the outer level of nested loops. The innermost poolsize was set at 128 frames. Figure 7.9 should be taken as an illustration of the scalability of the architecture itself and the scalability of demand-driven execution compared to conventional control-flow computing. The key take-away from these experiments is the feasibility

of extracting large degrees of instruction and loop parallelism without difficult parallel programming.

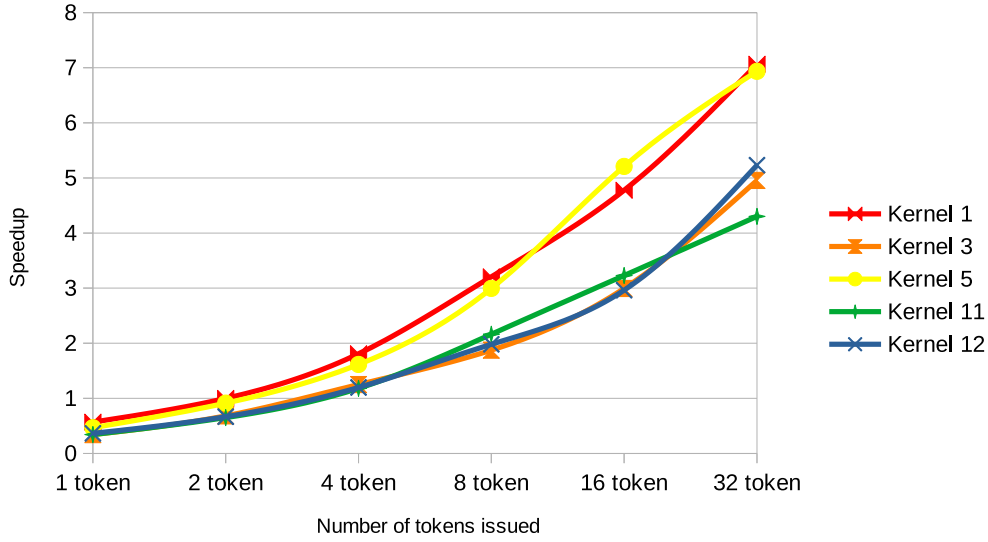


Figure 7.9: MIPS pipelined processor as base with max 64 active outer loop iteration and procedures with max 128 inner loop iterations

7.5.2.3 Superscalar Processor versus DDE

Figure 7.10 compares our processor with an ideal n-issue superscalar processor. The superscalar processor uses a central window and a gshare branch predictor. The store set algorithm is employed for memory disambiguation. The superscalar processor has been allocated 8192 rename registers. The benchmarks were run in an environment with maximum pool of 16 frames for simple functions as well as the outer level of nested loops. A maximum pool of 64 frames were available for innermost loop iterations. A single issue DDE processor brings a single operand at a time.

Most instructions are dyadic and the DDE processor needs to fetch these instructions twice. We therefore compare a $2 \times n$ -issue DDE processor against a n -issue superscalar processor.

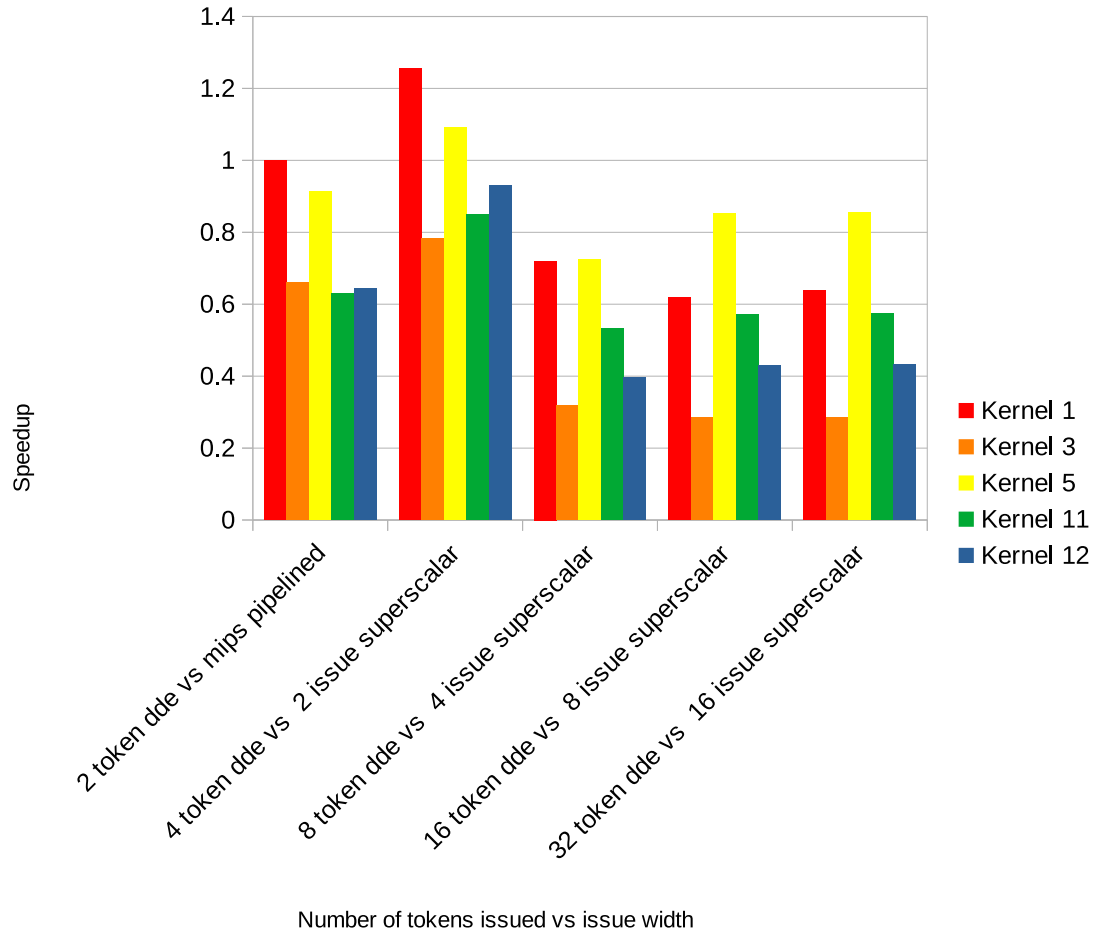


Figure 7.10: Superscalar processor as base with max 16 active outer loop iteration and procedure with max 64 inner loop iterations

These experiments clearly shows what a speculative superscalar processor is capable of. DDE architecture is able to do better than the superscalar processor only on low issue widths and just two kernels, namely, kernel 1 and kernel 5. Superscalar

processor's ability to schedule load instructions early due to its memory disambiguation capability as well as its branch prediction mechanism allows it to exploit a large degree of instruction-level parallelism.

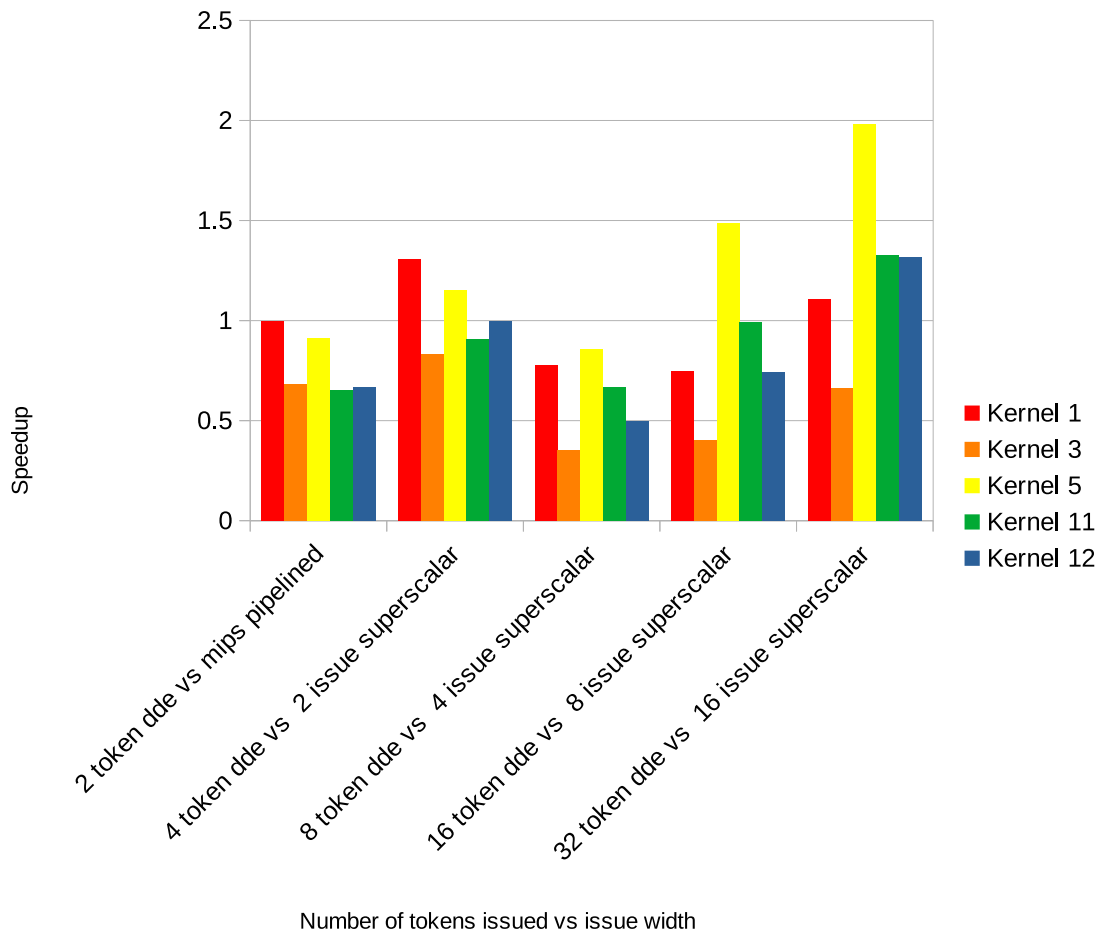


Figure 7.11: Superscalar processor as base with max 64 active outer loop iteration and procedures with max 128 inner loop iterations

Figure 7.11 illustrates a run for the demand-driven processor and an n-issue superscalar processor. The superscalar processor uses the same configuration as mentioned above, but the number of frames for DDE has been increased to 64 for outer loop

iterations and 128 for innermost loop iterations.

Exploring further loop-level parallelism remedy the situation somewhat and at large issue widths, DDE becomes significantly better. The main conclusion here is the significance of memory dependence speculation which DDE paradigm in theory is capable of.

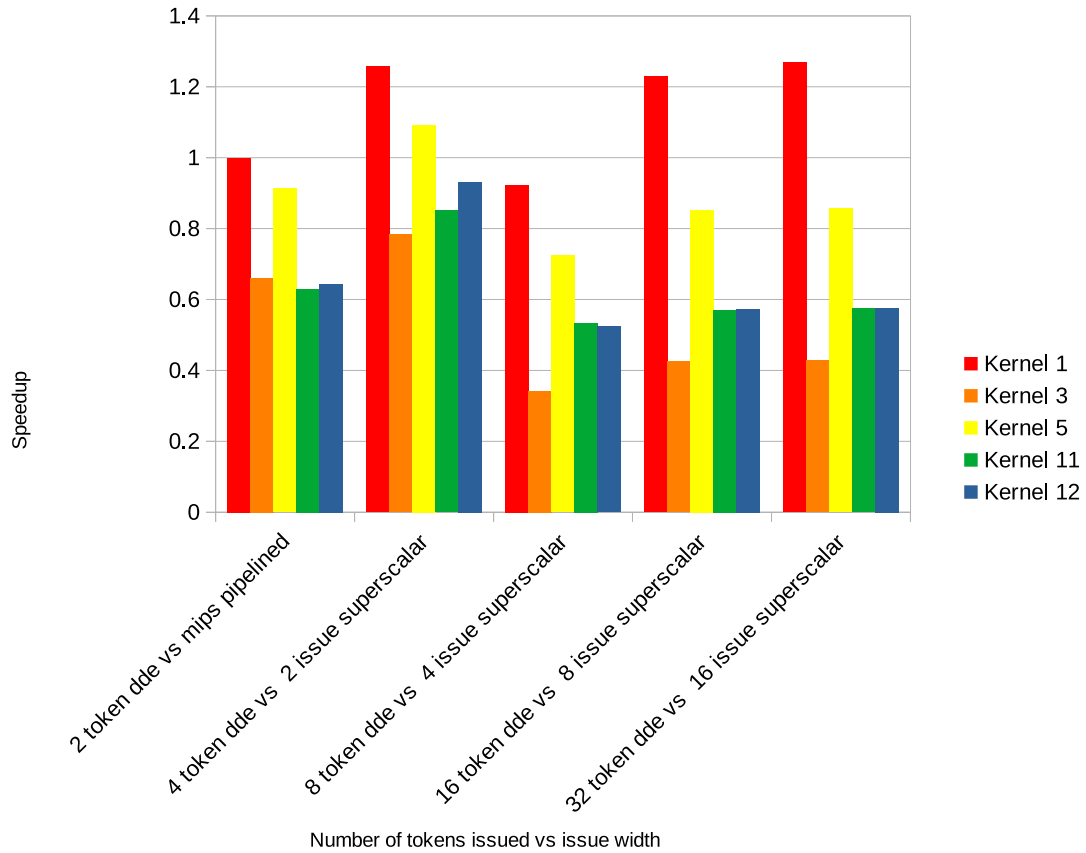


Figure 7.12: Superscalar processor as base without memory disambiguation, with max 16 active outer loop iteration and procedure with max 64 inner loop iterations

In order to further establish this analysis, we remove the disambiguation capability from the superscalar. Figure 7.12 illustrates a run for the demand-driven processor

and an n-issue superscalar processor. The superscalar processor uses the same configuration as mentioned previously but does not perform memory disambiguation. The benchmarks were run in an environment with maximum pool of 16 frames for simple functions as well as the outer level of nested loops. The innermost poolsize was set at 64 frames.

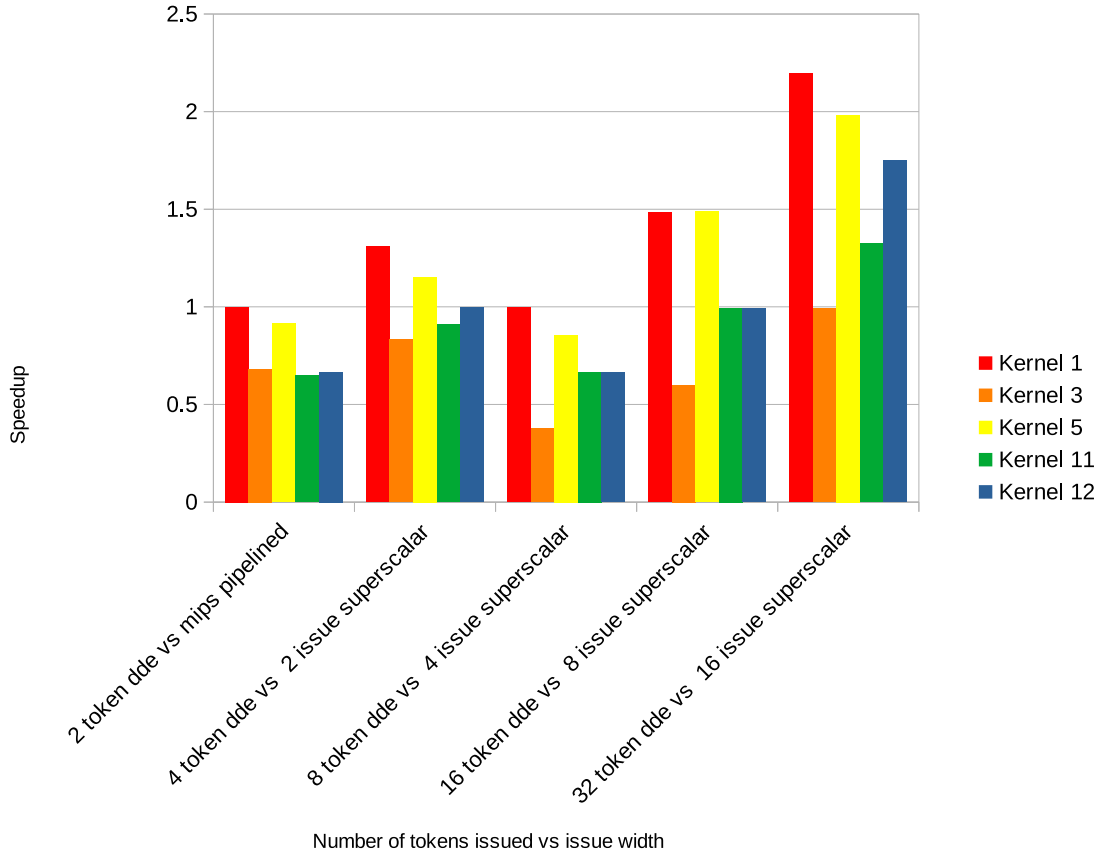


Figure 7.13: Superscalar processor as base without memory disambiguation, with max 64 active outer loop iteration and procedures with max 128 inner loop iterations

Figure 7.13 illustrates a run for the demand-driven processor and an n-issue superscalar processor. The superscalar processor uses the same configuration as mentioned

previously but does not perform memory disambiguation. The benchmarks were run in an environment with maximum pool of 64 frames for simple functions as well as the outer level of nested loops. A maximum pool of 128 frames were again available for the innermost loop iterations.

These experiment again confirm our conclusion that incorporation of dynamic memory disambiguation is a must for exploiting high degrees of instruction level parallelism together with loop-level parallelism.

7.5.3 Compiler Generated Benchmarks

The compiler under development at Florida State University is able to generate code for a few Livermore kernels. We have compiler generated assembly code for kernel 1, kernel 7 and kernel 12 which are vectorizable and kernel 3, kernel 5, kernel 9 and kernel 10 which are non-vectorizable. Kernels 1, 3, 5, 11, 12 were described in Section 7.5.1. The description for the rest of the kernels are as follows:

- 1) Kernel 7 is an *equation of state fragment* used in physics and thermodynamics to study the properties of fluids, mixture of fluids, and solids under different physical parameters such as pressure, volume, and temperature. The kernel performs vector-scalar multiplication and vector-vector addition operations. The code for the kernel is depicted in Figure 7.14.

```

for ( l=1 ; l<=100 ; l++ )
{
    for ( k=0 ; k<200 ; k++ )
    {
        x[k] = u[k] + r*( z[k] + r*y[k] ) + t*( u[k+3] + r*
            *( u[k+2] + r*u[k+1] ) + t*( u[k+6] + r*( u[k+5]
            + r*u[k+4] ) ) ) );
    }
}

```

Figure 7.14: Livermore kernel 7

2) Kernel 9 is an *integrate predictors*. The kernel performs vector-scalar multiplication and vector-vector addition operations as shown in Figure 7.15.

```

for ( l=1 ; l<=100 ; l++ )
{
    for ( i=0 ; i<200 ; i++ )
    {
        px[i][0] = dm28*px[i][12] + dm27*px[i][11] + dm26*
            px[i][10] + dm25*px[i][ 9] + dm24*px[i][ 8] +
            dm23*px[i][ 7] + dm22*px[i][ 6] + c0*( px[i][ 4]
            + px[i][ 5]) + px[i][ 2];
    }
}

```

Figure 7.15: Livermore kernel 9

3) Kernel 10 implements *difference predictors*. The kernel performs vector-vector subtraction operations as shown in Figure 7.16.

```

for ( l=1 ; l<=100 ; l++ )
{
    for ( i=0 ; i<200 ; i++ )
    {
        ar          =      cx[i][ 4];
        br          = ar - px[i][ 4];
        px[i][ 4] = ar;
        cr          = br - px[i][ 5];
        px[i][ 5] = br;
        ar          = cr - px[i][ 6];
        px[i][ 6] = cr;
        br          = ar - px[i][ 7];
        px[i][ 7] = ar;
        cr          = br - px[i][ 8];
        px[i][ 8] = br;
        ar          = cr - px[i][ 9];
        px[i][ 9] = cr;
        br          = ar - px[i][10];
        px[i][10] = ar;
        cr          = br - px[i][11];
        px[i][11] = br;
        px[i][13] = cr - px[i][12];
        px[i][12] = cr;
    }
}

```

Figure 7.16: Livermore kernel 10

7.5.3.1 Scalability of DDE Paradigm

We now evaluate the scalability of our architecture with set of Livermore kernel generated by the compiler. We first evaluate the architecture using a model which evaluates a single token per cycle. As before, we then vary the width of the architecture by using the single token architecture as the baseline. For every kernel, time taken to run

on single token architecture is taken as a baseline 1 for that kernel. We then measure the speedup in terms of execution time for individual kernels as we vary the width of the architecture. The baseline for the same kernel generated using handcoded kernel and compiler generated kernel is different.

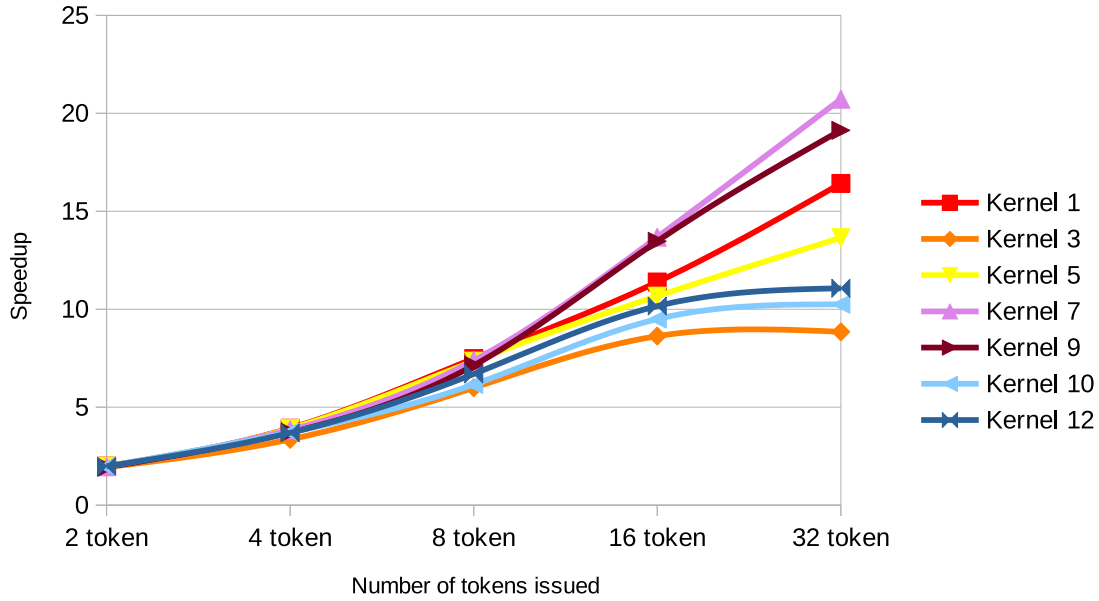


Figure 7.17: 1 token dde processor as base with maximum 16 active outer loop iteration and procedure with maximum 64 inner loop iterations

As mentioned earlier, we have a separate pool of frames to control the achievable parallelism. One pool of frames is used for simple functions as well as the outer levels of nested loops and a second pool of frames is used for the innermost level of nested loops. Figure 7.17 illustrates that a maximum pool of 16 frames were available for simple functions as well as the outer level of nested loops. A maximum pool of 64 frames were made available for innermost loop iterations. Similar to the handcoded

kernels, the performance starts to flatten out around 16 tokens due to lack of further loop-level parallelism to utilize the machine's capacity.

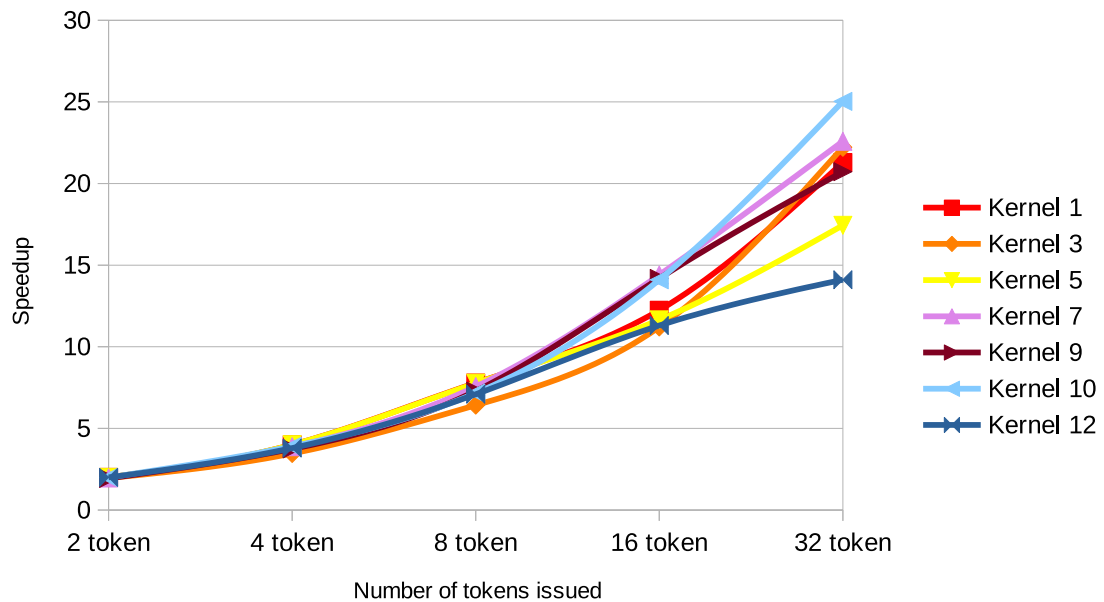


Figure 7.18: 1 token dde processor as base with max 64 active outer loop iteration and procedure with max 128 inner loop iterations

Figure 7.18 illustrates a maximum pool of 64 frames were available for simple functions as well as the outer level of nested loops. The innermost poolsize was set at 128 frames. As it can be seen, there is a significant performance gain compared to running the same set of benchmarks which were throttled using a maximum pool of 16 frames for simple functions as well as the outer level of nested loops. These experiment clearly show that increasing the pool size allows the processor to dynamically spawn more outer level of nested loops as per the availability of frames and the paradigm can exploit the measured level of available parallelism. Individual runs for same kernel

with handcoded and compiler generated code will have its own run on single token architecture as a base line. So we are not comparing the run for the same kernel for the code generated for handcoded and compiler code as a function of scalability.

7.5.3.2 Control-flow Single Issue versus DDE

Demand-driven processor running the compiler-generated code against a standard five stage MIPS pipeline with conventional internal data forwarding is shown in Figure 7.19. We use the same MIPS pipeline as the base when comparing the same kernels generated using handcode code and compiler generated code. The runs use a maximum of 16 outermost and 64 innermost loop frames.

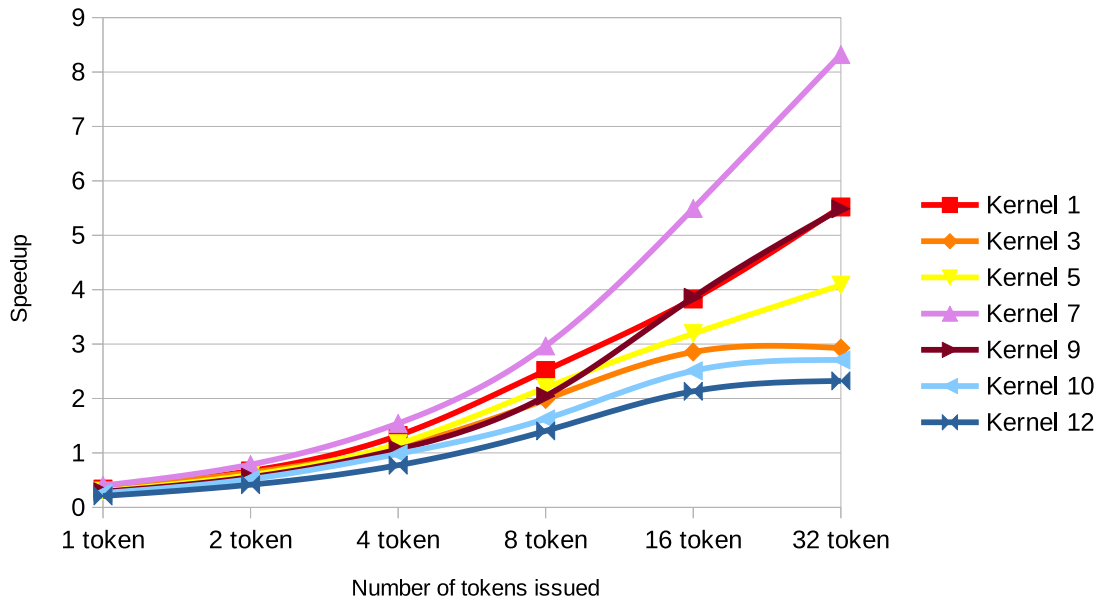


Figure 7.19: MIPS pipelined processor as base with max 16 active outer loop iteration and procedure with max 64 inner loop iterations

As with handcoded kernels, performance for Kernel 3, kernel 10 and kernel 12 starts to flatten out at 16 token architecture, although compiler generated kernels in Figure 7.19 are showing a better performance gain compared to handcoded kernels whose evaluation is shown in Figure 7.8.

Figure 7.20 illustrates the performance with maximum 64 outer and 128 innermost loop frames. The scalability of the architecture can be further seen as we are able to achieve even further speedups. When comparing compiler generated code from Figure 7.20 vs handcoded kernels in Figure 7.9 kernel 1 and kernel 3 have better speedups for compiler generated code whereas kernel 5 and kernel 12 show better speedups in handcoded programs.

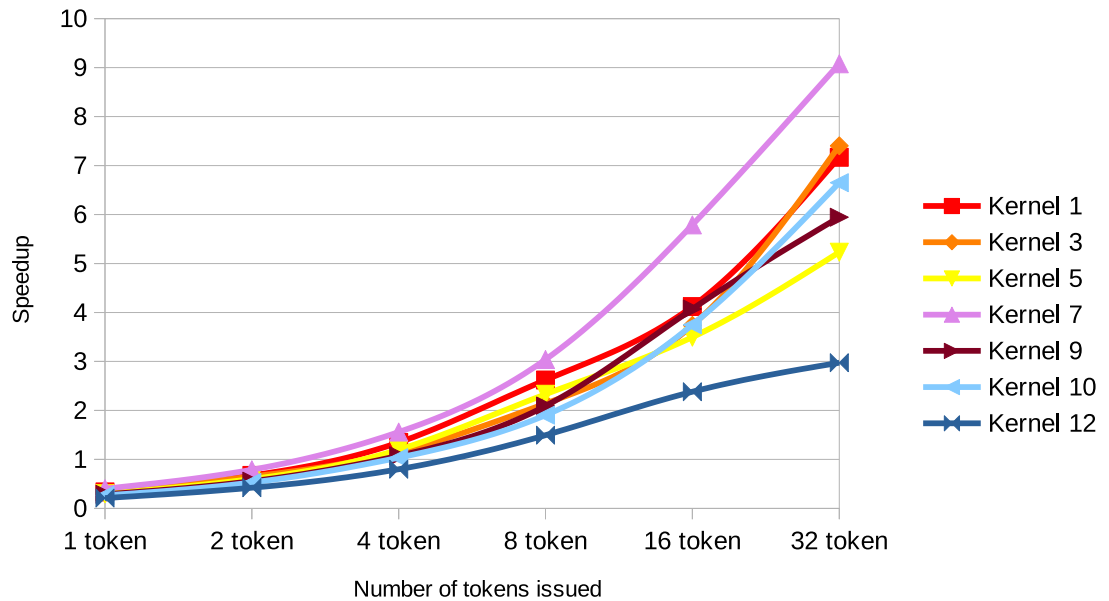


Figure 7.20: MIPS pipelined processor as base with max 64 active outer loop iteration and procedures with max 128 inner loop iterations

7.5.3.3 Superscalar Processor versus DDE

In this section, we compare compiler-generated code with an ideal n-issue superscalar processor. The superscalar processor configuration is the same as before. Figure 7.21 the performance with maximum 16 outer and 64 innermost loop frames.

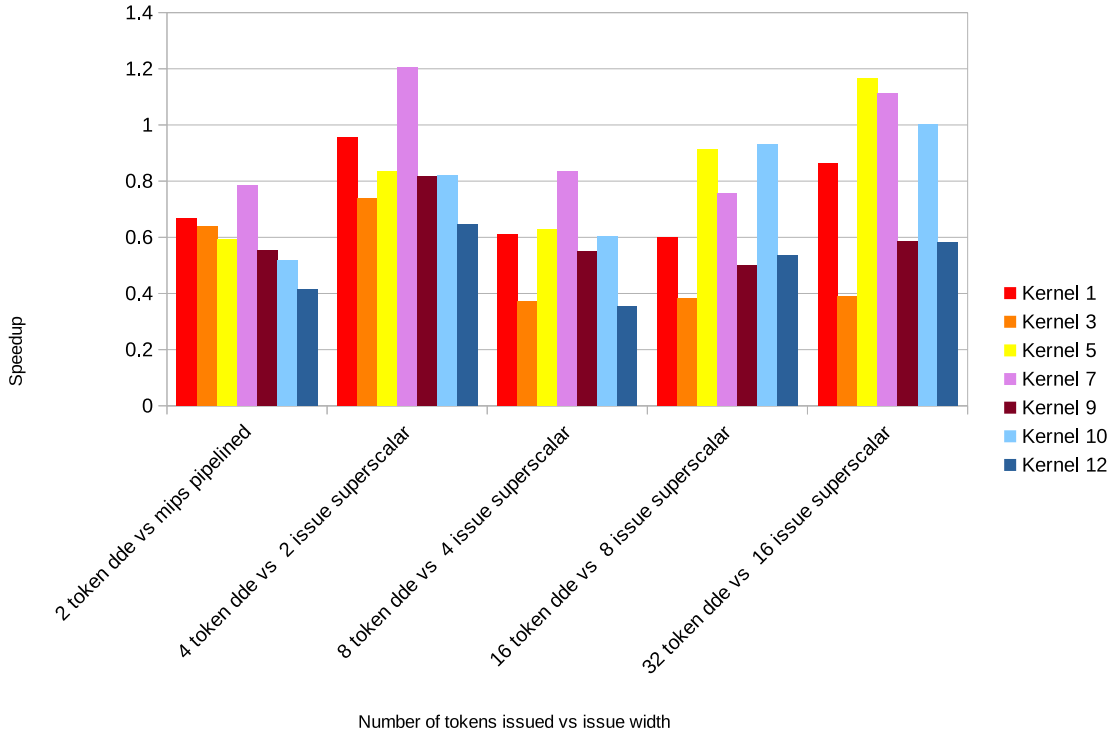


Figure 7.21: Superscalar processor as base with max 16 active outer loop iteration and procedure with max 64 inner loop iterations

Again, the DDE architecture does better than the superscalar on low issue widths with just kernel 7, and on high issue width, only for kernels 5 and kernel 7.

Figure 7.22 illustrates a run with a maximum 64 outer and 128 innermost frame

configuration.

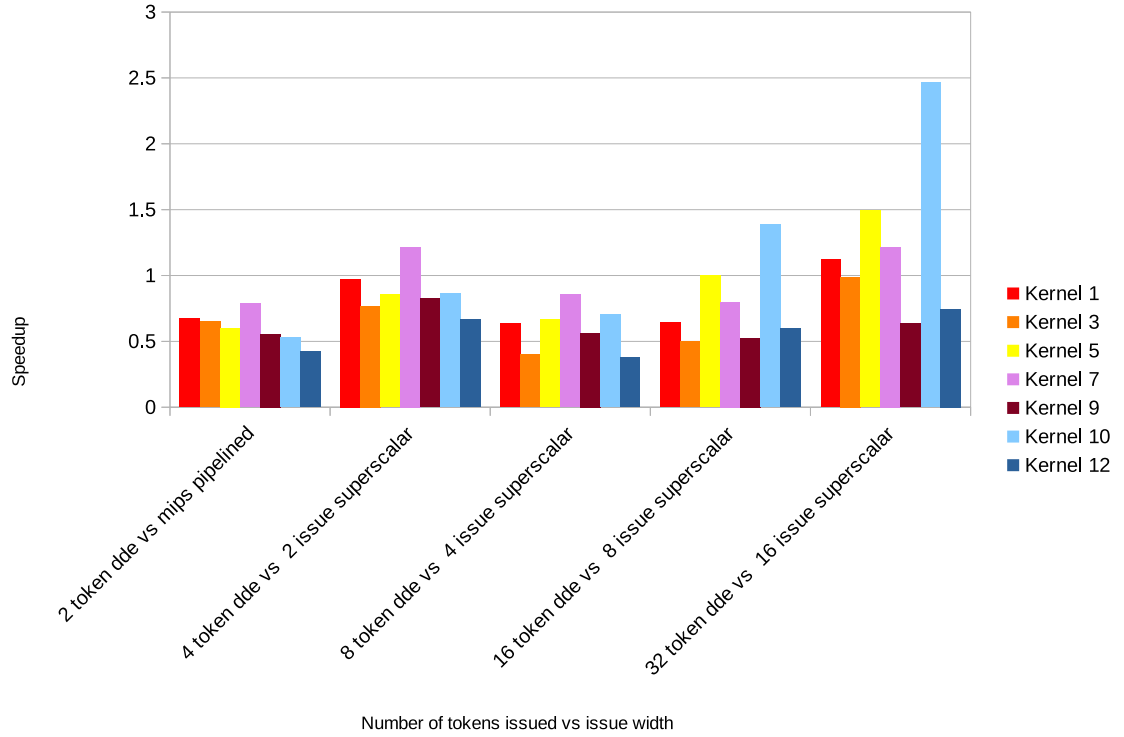


Figure 7.22: Superscalar processor as base with max 64 active outer loop iteration and procedures with max 128 inner loop iterations

With this set-up, superscalar processor does better only on kernel 9 and kernel 12.

Evaluation with respect to a superscalar processor without load speculation is shown in Figure 7.23. The runs use 16 innermost and 64 outermost frames.

Most of the kernels are doing better than superscalar processor at higher token architecture and issue width. Superscalar processor is doing better only on kernel 3 and kernel 12.

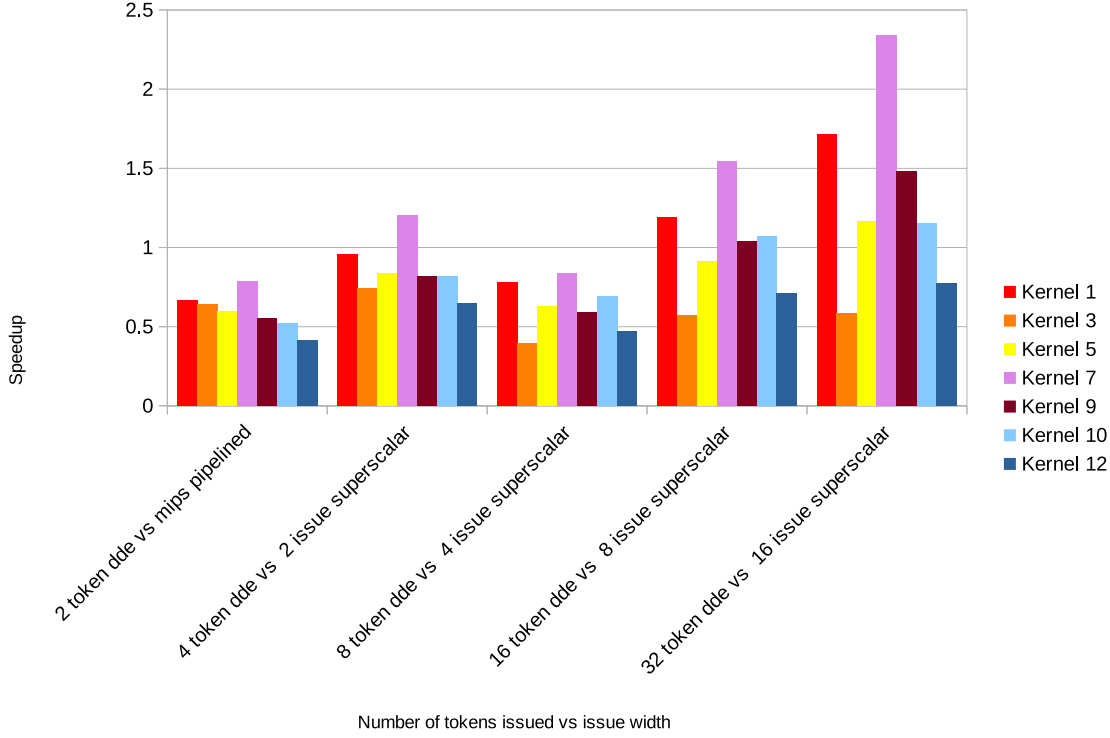


Figure 7.23: Superscalar processor as base without memory disambiguation, with max 16 active outer loop iteration and procedure with max 64 inner loop iterations

Figure 7.24 shows the result for maximum 64 outer and 128 innermost frame configuration.

In this configuration, all the kernels are doing better than superscalar processor at higher token architecture and issue width.

We compare the kernels for compiler generated code in Figure 7.24 with the hand-coded kernels as shown in Figure 7.13. Table 7.1 shows the number of instructions generated for individual kernels with handcoded and compiler-generated code. We

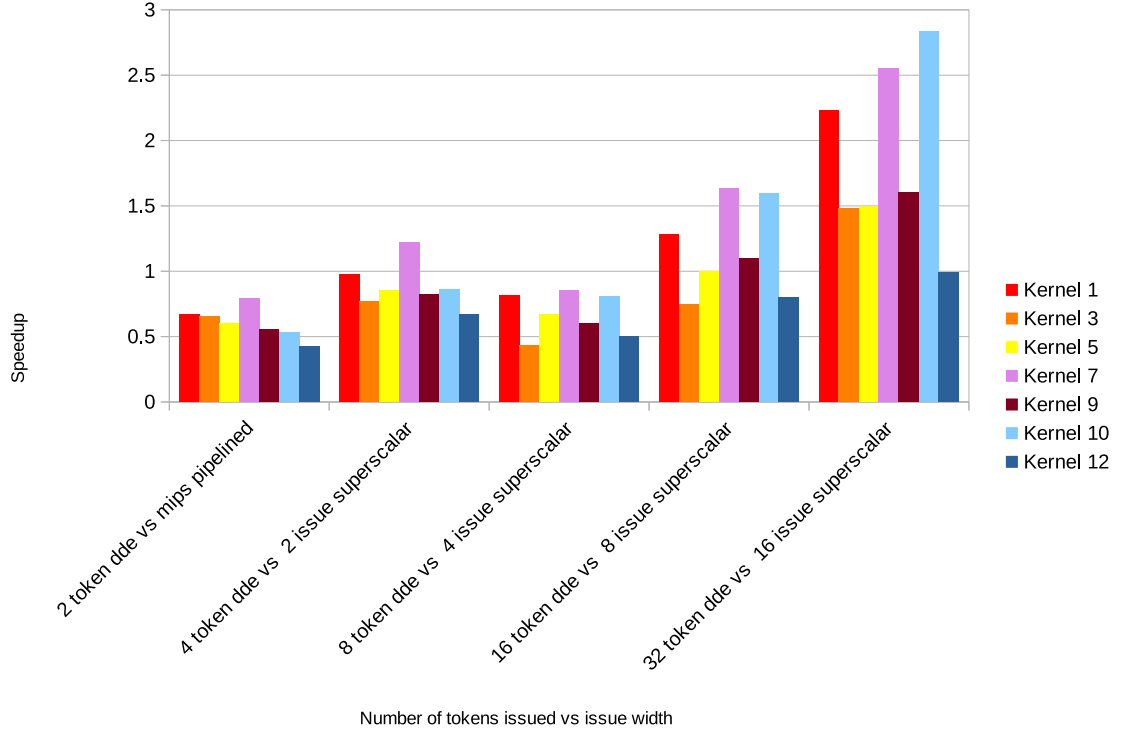


Figure 7.24: Superscalar processor as base without memory disambiguation, with max 64 active outer loop iteration and procedures with max 128 inner loop iterations

use HC and CG in the table to represent handcoded and compiler-generated code. The table displays the number of instruction generated for procedure call, the outermost loop and the innermost loop for individual kernels. All these kernels spend most of the time executing the innermost loops.

This are preliminary comparison between handcoded and compiler-generated code. The performance for individual kernel is dependent on the critical path of the chain of instructions to spawn a new loop iteration. The number of instructions in the innermost loop of an kernel will significantly impact the overall execution time of

Kernels	Procedure call	Outermost loop	Innermost loop
Kernel 1 HC	42	31	46
Kernel 1 CG	72	75	56
Kernel 3 HC	42	31	36
Kernel 3 CG	62	54	31
Kernel 5 HC	42	33	39
Kernel 5 CG	69	65	44
Kernel 12 HC	42	30	30
Kernel 12 CG	62	55	37

Table 7.1
Number of instruction in a Kernel

the kernel. We see from Table 7.1 the number of instructions generated for kernel 1, kernel 5 and kernel 12 is less for handcoded programs versus compiler-generated code for the outermost loop as well as innermost loop. For kernel 3 there are more number of instruction generated in handcoded program for the innermost loop. Kernel 1 is giving overall better performance for handcoded programs as we scale the architecture increasing the number of tokens. Only at architecture with 32 tokens the compiler generated code is able to match the performance of handcoded code. Kernel 3 have almost the same performance for both handcoded kernel and compiler-generated kernel, but the compiler generated code generates better performance at architecture of 16 token and 32 token. Kernel 5 and kernel 12 gives better performance for handcoded programs compared to compiler generated programs as we scale the DDE architecture from 2 token to 32 token.

Chapter 8

Conclusion

This dissertation explores a new execution paradigm for imperative programming languages. One of the significant contributions towards this goal is the development of the necessary programming language pragmatics, which allows imperative programs to be executed on a demand driven processor. Important contributions towards the developed programming language pragmatics are (1) A method showing representation of imperative programs for the demand driven paradigm; (2) The addressing modes (a) Literal, (b) Frame direct, (c) Displacement with respect to frames; (3) Procedures for dynamic creation and mapping of frames using static frame creation instructions; (4) Procedures for dynamic deallocation of frames when the use of a frame is complete; (5) A policy to pass arguments to a called function; (6) A formal method to call procedures; (7) Implementation of memory ordering between memory

instructions; (8) A method to represent loops for demand driven paradigm and a policy to dynamically unroll loops.

Another major contribution towards the development of demand driven paradigm is the development of a unified instruction set architecture, with instructions capable of running on a controlflow as well as demand-driven processor. This ISA is capable of handling arithmetic, logical, memory, gated, synchronization, and data transfer instructions.

The third major contribution is the design of microarchitectures for multiple-issue pipelined demand-driven processors. These designs elaborated in the earlier sections are realistic and can serve as a starting point for an actual implementation of the processor.

During the course of this work, it has become clear that speculative processing and execution of demand-driven programs, and returning of multiple values from functions, and parallel expansion are critical for competitive performance. We leave these aspects as future work.

References

- [1] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Comput. Surv.*, 15(1):3–43, March 1983.
- [2] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, October 1989.
- [3] David E. Culler and Gregory M. Papadopoulos. The explicit token store. Computation Structures Group Memo 312, Laboratory for Computer Science, MIT, 1991.
- [4] Jack B. Dennis, Clement K. Leung, and David P. Misunas. A highly parallel processor using a data flow machine language. Computation Structures Group Memo 134-1, Laboratory for Computer Science, MIT, 1979.
- [5] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic

- data-flow processor. In *Proceedings of the 2Nd Annual Symposium on Computer Architecture*, ISCA '75, pages 126–132, New York, NY, USA, 1975. ACM.
- [6] Shuhan Ding. *Future Value Based Single Assignment Program Representations and Optimizations*. PhD thesis, Michigan Technological University, Houghton, MI, USA, 2012. AAI3509734.
- [7] Shuhan Ding, John Earnest, and Soner Önder. Single assignment compiler, single assignment architecture: Future gated single assignment form*; static single assignment with congruence classes. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 196:196–196:207, New York, NY, USA, 2014. ACM.
- [8] Shuhan Ding and Soner Önder. Unrestricted code motion: A program representation and transformation algorithms based on future values. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 26–45, Berlin, Heidelberg, 2010. Springer-Verlag.
- [9] Robert A. Iannucci. Toward a dataflow/von neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, pages 131–140, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

- [10] MIPS Technologies Inc. *MIPS Architecture for Programmers*, volume II-A: The MIPS32 Instruction Set Manual v6.4. Imagination Technologies LTD., November 2015.
- [11] Omkar U. Javeri, Zhaoxiang Jin, and Soner Önder. A demand-driven instruction set architecture. Technical Report CS-TR-18-01, Department of Computer Science, Michigan Technological University, 2018.
- [12] AT&T Bell Laboratories and Oak Ridge National Laboratory. <https://www.netlib.org/benchmark/livermorec>.
- [13] Frank H. McMahon. Lfk. livermore fortran kernel computer test. <https://www.osti.gov/biblio/145762>.
- [14] Frank H. McMahon. The livermore fortran kernels: A computer test of the numerical performance range. <https://www.osti.gov/biblio/6574702>.
- [15] Rishiyur S. Nikhil. Can dataflow subsume von neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ISCA '89, pages 262–272, New York, NY, USA, 1989. ACM.
- [16] Soner Önder. Methods and systems for ordering instructions using future values. *US. Patent 7,747,993*, Filed Dec. 2004, Issued Jun. 2010.

- [17] Soner Önder and Rajiv Gupta. Automatic generation of microarchitecture simulators. In *Proceedings of the 1998 International Conference on Computer Languages*, ICCL '98, pages 80–, Washington, DC, USA, 1998. IEEE Computer Society.
- [18] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 82–91, New York, NY, USA, 1990. ACM.
- [19] Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins. Data-driven and demand-driven computer architecture. *ACM Comput. Surv.*, 14(1):93–143, March 1982.
- [20] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, December 1986.
- [21] Graph Visualization. <https://www.graphviz.org/doc/info/lang.html>.
- [22] Graph Visualization. <https://www.graphviz.org/download>.
- [23] Wikipedia. [https://en.wikipedia.org/wiki/DOT_\(graph_description_language\)](https://en.wikipedia.org/wiki/DOT_(graph_description_language)).

Appendix A

Instruction Set Architecture

Our instruction set design is guided by our need to explore the new paradigm. The compiler work necessary to compile complete programs for the paradigm is nothing but simple. Yet, the promise of the paradigm can be explored by compiling parts of the given program for the demand-driven execution while leaving the rest of the code in control-flow style is a feasible option. These design requirements call for an instruction set that can support both paradigms at the same time. We therefore design our instruction set as an extension of widely used MIPS instruction set, such that special instructions permits changing of the paradigm.

As previously described, we have three major addressing modes: 1) Literal addressing mode allowing the use of immediate value; 2) Frame direct addressing mode to directly

refer to any location in the frame; 3) Displacement addressing mode to indirectly refer a location from another frame using a pointer to another frame available at a location in the current frame along with a suitable offset.

The information about the addressing modes and the memory model which uses frames are used to design the instruction set architecture for the paradigm. The encoding of the instructions embeds the information for the opcode, upto two operands, an additional predicate operand, an immediate value, a signal/data bit, an execution mode bit and a floating point instruction expansion.

We use MIPS 32 instruction set as our reference instruction set. We use this to develop our own customized 32-bit instruction set with added functionality and features to run control-flow programs. For the demand driven execution, we encode considerable amount of information compared to control-flow instructions in order to support a modest frame direct and displacement addressing.

In order to use a unified instruction set where control-flow instructions are encoded with 32-bits, we spread the encoding information for a demand driven instruction between two 32-bit instructions. In short, we fuse two 32-bit instructions to make a 64-bit wide demand driven instruction. In our ISA, every instruction has one execution mode bit which indicates if the instruction is a control-flow or a demand driven instruction. Demand driven instructions always need to occur in a pair where a first instruction opcode provides the information about the actual functionality of the

instruction and a second instruction opcode serves as an extension to the first, thus making a 64-bit demand driven instruction. Fusing the instructions allowed us to save considerable amount of time compared to redesigning and developing an optimized instruction set where each instruction would have been 48-bit wide for control-flow and demand-driven instructions.

Each 64-bit instruction encodes sufficient information for different addressing modes. For instance, using the displacement addressing mode, an instruction can encode two operands and an additional predicate operand, where each of them are 12-bits long. The instruction set also makes it possible to hold an absolute 32-bit immediate value which can serve as a 32-bit memory address. We have two flavors of the instruction set for demand driven execution. 1) ISA based on 6-bit base + 6-bit displacement for operands. 2) ISA based on extended frame support with 1 bit indirect addressing + upto 11-bit displacement for operands.

ISA based on 6-bit base + 6-bit displacement allows demanding a location from another frame using a pointer which is read using a base field and adding a suitable displacement provided by the displacement. The ISA supports frame size of 64 location.

ISA based on extended frame support uses indirect addressing which uses the 11-bit displacement field to point to a location that provides the absolute frame address for a location in another frame. The location being pointed has an address computation

instruction. The ISA allows large frames, upto 2048 locations but requires an additional address computation instruction to compute a pointer used to demand value from another frame.

The rest of this appendix gives a description of the instruction categories for the demand-driven machine, the encoding used and the definition of each field used for encoding each of this instruction category.

We classify DDE instructions into three major instruction formats:

1. Data Memory location format (D-format)

The D-format instructions are capable of operating on two operands along with a predicate operand.

2. Constant format (C-format)

The C-format instructions are capable of operating on one operand and one immediate value along with a predicate operand.

3. Memory format (M-format)

The M-format instructions are capable of operating on upto two operands along with a predicate operand to read or store a value from or to the memory.

We have additional two minor format of instructions :

4. New Frame format (N-format)

The N-format instruction handles two operands along with an address label to assist in mapping of dynamically allocated frame and its arguments.

5. Frame address and NOOP format (FN-format)

The FN-format instructions do not have any operands. They assist in referencing of frames and the availability of data shared by a controlflow register.

We use the following terms to represent DDE ISA as available in DDE-ISA technical report by Javeri et al. [11]:

Opcode	: Operation code for the machine.
$sd_{operand}$	: Signal or Data field for an operand
rop_{base}	: Right operand base field.
$*rop_{base}$	: $*rop_{base}$ provides the base address for M-formate instructions for a functional implementation.
rop_{disp}	: Right operand displacement field.
$func_{float}$	: Opcode extension for floating point instructions.
FMT	: Format field defines single and double precision floating point format.
reserved	: Reserved for future use.
cf	: Control Flow (cf=1) or Demand Driven Execution (cf=0) machine mode selection field.
lop_{base}	: Left operand base field.
lop_{disp}	: Left operand displacement field.
$*lop_{base}$	: Act as a pointer to the source location of the argument block for functional implementation.
$*lop_{disp}$	: Provides the location at which the argument block pointer will be stored in the target frame for functional implementation.
pop_{base}	: Predicate operand base field.

pop_{disp}	: Predicate operand displacement field.
$*pop_{disp}$	: $*pop_{disp}$ is the higher 11-bit of a signed 32-bit constant for C-type instructions in pipelined implementation with large frame support.
constant_l21	: Constant_l21 is the lower 21-bit of a signed 32-bit constant.
$op2_{base}$	: Base address location for a memory instruction.
$op_{arg_{src}}$	: Pointer to the source location of the argument block.
$op_{arg_{trg}}$	: Location at which the argument block pointer will be stored in the target frame.
constant_l25	: Constant_l25 is the lower 25-bit of a signed 32-bit constant.
constant_h5	: Constant_h5 is the higher 5-bit of a signed 32-bit constant.
constant_l20	: Constant_l20 is the lower 20-bit of a signed 32-bit constant.
constant_h6	: Constant_h6 is the higher 6-bit of a signed 32-bit constant.
constant_m6	: Constant_m6 is the middle 6-bit from bit 26 to bit 21 of a signed 32-bit constant.
constant	: Constant is of the form $constant = (constant_l21 \text{ concatenate } *pop_{disp})$ for pipelined implementation with large frame support.
constant	: Constant is of the form $constant = (constant_h6 \text{ concatenate constant_m6 concatenate constant_l20})$ for functional implementation.
unused	: Unused implies currently unused fields.
unused_c	: unused_c implies currently unused fields.
unused_noop	: unused_noop implies currently unused fields.

displacement	: Displacement is a 16-bit signed offset.
DEP	: Data Environment Pointer.
rop	: rop is of the form rop_{disp} for pipelined implementation with large frame support.
lop	: lop is of the form lop_{disp} for pipelined implementation with large frame support.
preop	: preop is of the form pop_{disp} for pipelined implementation with large frame support.
rop	: rop is of the form $rop_{disp}(rop_{base}) = rop_{base} + rop_{disp}$ for functional implementation.
lop	: lop is of the form $lop_{disp}(lop_{base}) = lop_{base} + lop_{disp}$ for functional implementation.
preop	: preop is of the form $pop_{disp}(pop_{base}) = pop_{base} + pop_{disp}$ for functional implementation.
<i>dest</i>	: An location in Data Memory(DM) used to store a value as a destination location.

Table A.1
Instruction fields

Demand Driven Execution (DDE) instruction format as described by Javeri et al. [11]:

Data Memory location format (D-format)(**pipelined large frames**):

Opcode Mnemonic

31 - 26	25	24	23 - 13	12 - 7	6 - 3	2	1	0
opcode	sd_{rop}	rop_{base}	rop_{disp}	$func_{float}$	FMT	sd_{pop}	reserved	cf

DDE extension

31 - 26	25	24	23 - 13	12	11 - 1	0
opcode	sd_{lop}	lop_{base}	lop_{disp}	pop_{base}	pop_{disp}	cf

Data Memory location format (D-format)(**functional**):

Opcode Mnemonic

31 - 26	25	24 - 19	18 - 13	12 - 7	6 - 3	2	1	0
opcode	sd_{rop}	rop_{base}	rop_{disp}	$func_{float}$	FMT	sd_{pop}	reserved	cf

DDE extension

31 - 26	25	24 - 19	18 - 13	12 - 7	6 - 1	0
opcode	sd_{lop}	lop_{base}	lop_{disp}	pop_{base}	$*pop_{disp}$	cf

Note: $*pop_{disp}$ provides 11 most significant bits for the immediate value.

Constant format (C-format)(**pipelined large frames**):

Opcode Mnemonic

31 - 26	25 -5	4-2	1	0
opcode	constant_l21	unused_c	reserved	cf

DDE extension

31 - 26	25	24	23 - 13	12	11 - 1	0
opcode	sd_{lop}	lop_{base}	lop_{disp}	pop_{base}	$*pop_{disp}$	cf

Note: $*pop_{disp}$ provides 11 most significant bits for the immediate value.

Constant format (C-format)(**functional**):

Opcode Mnemonic

31 - 26	25 -6	5-2	1	0
opcode	constant_l20	unused	reserved	cf

DDE extension

31 - 26	25	24 - 19	18 - 13	12 - 7	6 - 1	0
opcode	sd_{lop}	lop_{base}	lop_{disp}	constant_h6	constant_m6	cf

Memory format (M-format)(**pipelined large frames**):

Opcode Mnemonic

31 - 26	25	24 - 19	18 - 3	2	1	0
opcode	sd_{rop}	$op2_{base}$	displacement	sd_{pop}	reserved	cf

DDE extension

31 - 26	25	24	23 - 13	12	11 - 1	0
opcode	sd_{lop}	lop_{base}	lop_{disp}	pop_{base}	pop_{disp}	cf

Memory format (M-format)(**functional**):

Opcode Mnemonic

31 - 26	25	24 - 19	18 - 3	2	1	0
opcode	sd_{rop}	$*rop_{base}$	displacement	sd_{pop}	reserved	cf

DDE extension

31 - 26	25	24 - 19	18 - 13	12 - 7	6 - 1	0
opcode	sd_{lop}	lop_{base}	lop_{disp}	pop_{base}	pop_{disp}	cf

Note: $*rop_{base}$ provides the base address for M-formate instructions.

New Frame format (N-format)(**pipelined large frames**):

NEWF

31 - 26	25 - 1	0
opcode	constant_l25	cf

DDE2 extension

31 - 26	25-15	14 - 6	5 - 1	0
opcode	$op_{arg_{src}}$	$op_{arg_{trg}}$	constant_h5	cf

New Frame format (N-format)(**functional**):

NEWF

31 - 26	25 - 6	5-2	1	0
opcode	constant_l20	unused	reserved	cf

DDE

31 - 26	25	24 - 19	18 - 13	12 - 7	6 - 1	0
opcode	sd_{lop}	$*lop_{base}$	$*lop_{disp}$	constant_h6	constant_m6	cf

Note: $*lop_{base}$ act as a pointer to the source location of the argument block.

$*lop_{disp}$ provides the location at which the argument block pointer will be stored in the target frame.

Frame address and NOOP format (FN-format)(**pipelined large frames**):

Opcode Mnemonic

31 - 26	25 - 2	1	0
opcode	<i>unused_{noop}</i>	reserved	cf

DDE

31 - 26	25	24	23 - 13	12	11 - 1	0
opcode	<i>sd_{lop}</i>	<i>lop_{base}</i>	<i>lop_{disp}</i>	<i>pop_{base}</i>	<i>pop_{disp}</i>	cf

Frame address and NOOP format (FN-format)(**functional**):

Opcode Mnemonic

31 - 26	25 - 2	1	0
opcode	<i>unused_{noop}</i>	reserved	cf

DDE

31 - 26	25	24 - 19	18 - 13	12 - 7	6 - 1	0
opcode	<i>sd_{lop}</i>	<i>lop_{base}</i>	<i>lop_{disp}</i>	constant_h6	constant_m6	cf