EFFICIENT PARALLEL PROGRAMMING ON DATACENTER PLATFORMS WITH DISCIPLINED MEMORY ACCESS

by

XIANG GAO

A Thesis Submitted to
The Hong Kong University of Science and Technology
in Partial Fulfillment of the Requirements for
the Degree of Master of Philosophy
in Computer Science and Engineering

September 2012, Hong Kong

Copyright © by Xiang Gao 2012
Authorization

I hereby declare that I am the sole author of the thesis.

I authorize the Hong Kong University of Science and Technology to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the Hong Kong University of Science and Technology to reproduce the thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Xiang Gao
XIANG GAO
EFFICIENT PARALLEL PROGRAMMING ON DATACENTER PLATFORMS WITH DISCIPLINED MEMORY ACCESS

by

XIANG GAO

This is to certify that I have examined the above M.Phil. thesis and have found that it is complete and satisfactory in all respects, and that any and all revisions required by the thesis examination committee have been made.

[Signature]
DR. CHARLES ZHANG, THESIS SUPERVISOR

[Signature]
DR. LIN GU, THESIS SUPERVISOR

[Signature]
PROF. HAMDI, MOUNIR, HEAD OF DEPARTMENT

Department of Computer Science and Engineering
21 September 2012
ACKNOWLEDGMENTS

Pursuing a master degree is a long degree journey that one cannot make alone, and I would like to thank the people who have contributed to my journey.

First, I would like to express my deepest gratitude to my supervisors Charles Zhang and Lin Gu. During my study, they gave me great help, constant guidance and support on my research. This thesis would never have got to this stage without their help.

Second, my thanks go to Prof. Shing-Chi Cheung and Qiong Luo who severed as my MPhil examination committee.

In addition, I also would like to express my appreciation to our research group members who all gave me insightful comments. It is a great pleasure to work with them.

Last but not least, my sincere thanks are expressed to my family for the endless love and encouragement.
# TABLE OF CONTENTS

Title Page i
Authorization Page ii
Signature Page iii
Acknowledgments iv
Table of Contents v
List of Figures vii
List of Tables viii
Abstract ix

Chapter 1 Introduction 1

Chapter 2 Background 6
  2.1 Layer Zero: A Substrate for Distributed Computing 6
    2.1.1 Memory and Transaction Management 6
    2.1.2 Task Scheduling and Watchers 8
    2.1.3 Runner Compartment and Binary Translation 9
  2.2 \textit{L0}: The Instruction Set of \textit{L0} 10
    2.2.1 Instruction Formats 10
    2.2.2 Operands and Addressing Modes 10
    2.2.3 General Instructions 11
    2.2.4 Task Management Instructions 11

Chapter 3 Disciplined Runtime 14
  3.1 Overview 14
  3.2 C0 Programming Language 15
    3.2.1 Language-Level Multitasking with “runner” statement 16
    3.2.2 Accessible Memory Declaration with “using” clause 17
    3.2.3 Task Dependencies and Watchers with “watching” clause 18
LIST OF FIGURES

3.1 Overview of Disciplined Runtime 15
4.1 Performance Comparison 30
4.2 Performance Profiling 31
4.3 Scalability of Amazon EC2 32
LIST OF TABLES

2.1 Operand Attributes. Attributes starting with letter 'm' is only for mov instruction, other attributes are used for instructions other than mov 11
2.2 General Instructions in I0 12
2.3 General Instructions in I0 12
3.1 IL Operand Types 23
3.2 Implementation metrics 26
4.1 Benchmark Applications 29
EFFICIENT PARALLEL PROGRAMMING ON DATACENTER PLATFORMS WITH DISCIPLINED MEMORY ACCESS

by

XIANG GAO

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology

ABSTRACT

Many distributed system technologies and programming frameworks are developed to support parallelized computation on the datacenter platform. While simplifying data-centric parallel computation, existing system technologies and application frameworks, however, exhibit limitation in their capability, efficacy and programmability in supporting sophisticated applications. In the thesis, we present the new principle of disciplined memory access, together with the new C0 programming language and a compiler, to support large-scale parallel computing on compute clusters with loosely-coupled commodity servers. Our prototype implementation, called Disciplined Runtime, is built under the principle of a disciplined memory access model, which means the programmer and the compiler furnish memory access information to the system so that the latter can detect memory access patterns and locality preferences at run time. Without limiting the expressing capability, this makes the memory access behavior known to the memory subsystem and the scheduler, and this information can be utilized to significantly improve the efficiency and performance of the system. The Disciplined Runtime provides distributed multitasking in a large uniform memory space with transactional semantics, and the new programming language enables programmers to provide memory access information of parallel tasks in a straightforward way so that the programs naturally follow the disci-
plined memory access model. Evaluation on both our research testbed and Amazon EC2 show that the new multitasking engine provides high performance and scalability on 32 physical compute servers and 256 large EC2 instances.
CHAPTER 1

INTRODUCTION

With the increasing importance of cloud-based computation, more and more programs from different problem domains are written to run in the datacenter. As a large-scale distributed system with thousands of loosely-coupled compute servers, modern datacenters provide both abundant computing resources and numerous challenges in effectively using these resources and exploiting parallelism.

Because traditional parallel programming does not fit well with the new design space where faults are “norm”, concurrent accesses occur at a massive scale, and applications evolve on a day-to-day basis, alternative solutions, pioneered by MapReduce [16], emerge to simplify parallel programming and provide an easy-to-use approach to orchestrating many concurrent tasks to solve certain classes of problems. More recent developments include DryadLINQ [31], Storm [1], MRLite [25] and Piccolo [28].

However, many existing cloud-based computing models achieve the parallelization capability at a significant cost in generality and efficiency. Most of them can achieve high performance on “embarrassingly parallel” workloads, but exhibit difficulty in rendering more sophisticated computation. The reasons are manyfold. First, MapReduce, which has a heavy influence on parallel computation on large clusters with commodity hardware, was constructed for a specific application environment and an earlier design context where physical memory and high-speed networking were very expensive [6]; Second, scalability and fault tolerance were paramount issues in earlier application systems in the datacenter environment; Finally, there are severe technical challenges in efficiently supporting complex computation in a large and dynamic compute cluster.

Existing platforms already make many attempts to support large-scale distributed programming without requiring programmers to deal with the communication details. Some solutions [16, 24] require the programs be written in a specified workflow and avoid using shared memory states. Although forcing the programs to be written in this way could eliminate communication among tasks in order to provide better scalability, it leads to bad single node performance and limits the capability of expressing complex application logics. For example, Hadoop [3] adopts the MapReduce model, which requires applications
to be written as clearly defined two phases. Some iterative algorithms are difficult to be
expressed in Hadoop, and, because Hadoop uses HDFS instead of memory as a way to
store the global states, the global data accessing is much slower than it could be.

Other solutions [11, 21] try to provide a unified programming interface similar to a
single machine. These approaches assume the heuristics such as temporal and spacial
locality to maintain the data access efficiency, and believe these phenomena still have the
same effects in a distributed environment. While these solutions provide general interfaces,
they have one common problem that the phenomena and programming models suitable
for a single machine are not suitable for programming in a distributed environment. As a
result, these solutions usually have difficulties in scaling to a large number of nodes and
even expert programmers do not have many opportunities to optimize it.

As the application systems grow in quality and complexity, the inefficacy and lim-
itations of existing cloud-oriented multitasking technologies and programming models
has made it more difficult to develop sophisticated application programs, provide high
performance and achieve reasonably low latency on the datacenter computing platform.
Although the data substrate for the same platform has evolved, responding to the same
trend in application requirements, from simple file system interfaces [19], key-value store,
NoSQL [10] databases to more advanced transactional abstractions [5, 15], it remains an
open problem how to construct a multitasking engine in the datacenter environment so
that system operations are low-latency, a diversity of workloads can efficiently execute,
and developers can easily program a large number of tasks to process data in parallel with
semi-automatic dependency resolution.

As we design and implement a new distributed platform for compute clusters with
loosely-coupled commodity servers, the key to the problem is to balance scalability, effi-
ciency, and generality. Specifically, the system should be able to handle tens of thousands
of concurrent tasks, incur millisecond-level latency in distributed task control operations,
and provide a general-purpose support for application systems so that such systems are
easy to program and efficient to execute.

To overcome the challenge, we investigate many common applications that run in
datacenters. As we observed, in most applications written for datacenters, the data ac-
cess behaviors follow some specific patterns, which are different from traditional locality
behaviors. For example, in some applications such as image processing and matrix algo-
rintms, a single parallel task processes a part of the entire input data and generates a part
of the output data; in some other applications, such as distributed searching, a parallel
task reads most of the input data but never modify them. The data access pattern highly
depends on the application logic. In order to improve the performance, the system would
benefits from knowing some information about these data access patterns.

Researchers already start to realize that the programmers should have the ability
to specify the data access behavior and locality preference in distributed programming. Much work has been done to exploit this phenomenon. X10 [12] uses the partitioned
global address space (PGAS) model to provide unified program states. X10 and other
PGAS-based solutions [4, 7, 27] require the partition of data to be specified statically
at compile time, but in real applications it is difficult to derive the locality preference
without knowing the input data, and the data access pattern may change during the
execution. Distributed key-value store systems [18, 28] provide a separate interface, and
possibly provide customizable locality preferences and conflict resolution. However, these
systems requires a different accessing interface(key-value pairs) than accessing program
variables(memory references), which adds complexity to application programming and
imposes a burden of porting existing algorithms. Transactional memory systems [9, 20,
29,30] also require accesses to the global data to be achieved via separate interface, which
lead to similar problems as distributed key-value store systems.

Based on these phenomenons and observations, try to introduce a new programming
model to support large-scale disturbed programming. In a distributed system, how appli-
cation accesses data is a critical factor of performance. In an application executing on a
datacenter, the data is located on different nodes and many parallel tasks are distributed
to these nodes. The way the application uses memory can affect the performance in many
ways. Locality is one of the most important issues. The local memory usually has more
than 100Gbps bandwidth and less than 100 nanoseconds latency, which is much faster
than typical high-speed networks with usually no more than 10Gbps bandwidth and more
than 1000 nanoseconds latency. This speed gap makes accessing data located locally over
ten times faster than accessing data located remotely.

In a distributed system, the overhead of memory space manipulation is directly related
to the size of the space. Although the actual content of memory may not need be copied
until accessed, maintaining the metadata is still costly. For example, TL2 [17] requires
maintaining a version number and a lock object for each entry, which is at least 16 bytes.
In a system with page-level granularity, to maintain a 1GB snapshot, there are at least
262,144 metadata entries, and the size of the metadata is 4MB. This amount of data must be stored for each parallel task and must be transmitted through the network when manipulating snapshot states. For a mass-parallel program with thousands of parallel tasks, this overhead will be phenomenal. From the above analysis, it is clear that if the system could know which part of memory a parallel task will access, the size of memory space of each task is controlled to an minimized size which contains only the necessary memory locations.

As a result of importance of memory access, we believe that a distributed system should know more information about the memory affinity of each parallel task and utilize this information to improve the performance, and the application should be able to specify the use of memory and the memory affinity dynamically at runtime. What memory addresses will be accessed by a given parallel task are highly depend on both the application logics and the input data. The application logic is known by the programmers, and the input is know at runtime which the programmers are unable to predict or control. So neither the programmer nor the system alone can know this affinity information. Also, the affinity behavior may change during the execution of the application.

We propose a new principle of distributed programming – disciplined memory access. The programming needs to specify memory range templates which is a set of memory ranges a task wants to access with unknown input parameters according the the application logic. Assisted by the compiler, these range templates are encoded into the application and fed to the runtime system. At runtime when the input is known, the system instantiates these range templates to calculate specific memory ranges, before a parallel task starts executing. For example, a parallel merge-sort algorithm separate parallel task is used for each half of the sequence. The application logic determines that in each step, the two parallel task accesses a segment of the sequence, but only when the input is known at runtime can the exact memory ranges of the segments be determined.

As a result, in a system following the disciplined memory access model, the runtime system has more information to make decisions. The information can be used by many components of the runtime system to achieve better performance. In the distributed memory system, such memory access information is used to trim the metadata of the distributed memory space and the runtime only needs to maintain a very small part of the memory information data structure, which can significantly improve the efficiency of distributed memory management. Also, the distributed task scheduler can distributed a
task to where most of its data located to provide better locality. The system can also delay scheduling two transactions with overlapped memory ranges to avoid conflicts.

Although disciplined memory access requires explicitly requesting accessible memory ranges, it does not hurt the generality or expressiveness. The programmer who design and understand the algorithm is reasonably to be expected to know which part of data a parallel task will access for a given input data set. Most of the existing transactional memory system also require the programmers to use specific API to make all the memory accesses known to the system, without limiting the expressiveness. Actually, the traditional uniformed flat memory access model can be considered as an extreme case of disciplined memory access, where each task can access the entire memory space.

Based on the instruction-set level virtualized system L0 [26], which includes a set of system mechanisms necessary to express, monitor, and enforce the disciplined memory access, we build the Disciplined Runtime. L0 presents a uniform memory space with transactional semantics, and provides system constructs to declare and share data objects among multiple tasks as well as resolve task dependency according to the shared memory state. On top of L0, we design a C-style programming language to support programming with disciplined memory accesses, and developed a prototype compiler for this language and primitive library and application framework. The programmers has the ability to optimize the memory access performance by improving the data structures and access patterns.

To evaluate the effectiveness and performance of the system, we conduct experiments using several benchmark programs from different problem domains. This also demonstrates the generality of our design and the programming model. We use both a research testbed and the Amazon EC2 cloud platform to evaluate the scalability of our system. The experiments on up to 32 physical machines and 256 large EC2 instances show that the system can support efficient parallel computation in the datacenter environment with high performance and scalability.

The rest of the thesis is organized as follows: Chapter 2 introduces some background of L0 necessary for understanding our main technique, Chapter 3 describes the design and implementation of the system, Chapter 4 shows the experiment results and analysis of the evaluation, Chapter 5 introduces some related work and discuss how Disciplined Runtime differ from them, and Chapter 6 concludes the thesis.
CHAPTER 2

BACKGROUND

The Disciplined Runtime is built on top of Layer Zero (L0)\(^1\). The L0 is originally published in [26]\(^2\). This chapter introduce necessary background knowledge to help understanding our main technique.

2.1 Layer Zero: A Substrate for Distributed Computing

Layer Zero is a substrate for large-scale distributed computing. It utilizes many physical computing nodes in a datacenter to provide a illustration of a big virtual machine using a new instruction set I0. It consists a distributed memory manager, a task scheduler, and several execution nodes with the capability of binary translating I0 into platform code.

2.1.1 Memory and Transaction Management

The L0 memory system provides a flat memory space with 64-bit addressing for the applications. However, the entire memory space is divided into several regions which have different characteristics. The most important regions are described as follows.

Private region (PR): The memory content in private region is isolated between tasks. Each task will have a separate storage space for each address within PR. Accessing a memory access first time into private region is faster than accessing into shared region, because PR does not maintain metadata necessary for distributed memory management. As a result, PR is usually used for storing temporary data which does not need to be shared among different tasks.

Shared region (SR): The memory content in shared region is shared among tasks. SR is maintained by the distributed transactional manager which provides transactional

\(^1\)http://www.lazero.net/index.htm

\(^2\)As the paper published, L0 was named as DVM and I0 was named as DISA. This thesis use the new names different from the published paper.
memory semantics. A change in shared region will be visible to all tasks after the transaction successfully commits. SR is usually used for storing input/output data, and intermediate data that needs to be shared among tasks.

The memory management in L0 provides transactional semantics. Although the shared region memory space is flat and continuous, the shared region memory each task can access, which is made up of a set of memory ranges, is usually sparse and significantly smaller than the entire memory space. These ranges are specified by a data structure filled when creating new tasks. Each task can access only the pre-declared memory ranges when executing. When a task finishes, the memory space of the task can be committed into the global space. If no conflict occurs, the changes to the memory made by the task is visible to all tasks created after this commission.

Among all the memory ranges of a task memory space, one is used as the stack, and other ranges are used as the heap. The difference between the stack range and the heap range is that, the stack range are excluded from the transaction. As a result, all changes to the stack range are discarded when task exits. This behavior is similar to memory within the private region, but stack range in the shared region is able to accept input data from parent task, while private region memory cannot be used for passing data from one task to another.

L0 support transaction semantics with snapshot isolation. Instead of using specified transaction memory API like Version Boxes in JVSTM, L0 use transparent memory addressing to provide a more transparent interface. And tasks are only allowed to commit a transaction when it terminates. In other words, there is no way to commit a transaction in the middle of a task. This design choice is made to avoid unnecessary synchronization operations, because synchronization in a distributed platform are quite expensive.

In the current implementation, the distributed virtual memory management is done with the assistance of x86_64 memory management unit. We leverage the paging mechanisms to implement the on-demand loading of memory pages. When an runner starts, the range list of accessible memory addresses is translated into page list, and the page list is transmitted to the distributed shared memory home (DSM Home) server for snapshot management. The DSM Home server will return a page list with the page information such as where a memory page actually locates. When a runner executes and memory address within the task memory space is accesses the first time, it triggers a page fault. In the page fault handler, the runner compartment communicate with other nodes where
the actual content locates to copy the data.

2.1.2 Task Scheduling and Watchers

Scheduling tasks in a distributed system is much more complex than scheduling a thread in a single machine. The locality issue has more significant impact in a distributed system than in a single machine, because accessing data through network is much slower than accessing memory installed on the local machine. As a result, scheduling decisions must be more carefully made. Also, migrating tasks or data in a distributed system is quite expensive, so the scheduling would better to make a good choice at the very first place.

The scheduler in L0 system use the memory ranges of task memory space as a hint of locality. Because a task can only access memory addresses within the task memory space, the scheduler has the coarse memory locality preference once the task is created. Having this information, it is eligible to schedule a task to where most of its data is likely to resides. Additionally, the scheduler assumes the ranges of the memory space are order by priority (in other words, frequency of access), so the programmer can optimize the locality by telling the system which part of the data is most frequently accessed.

A specific design in L0 is that a task will only be scheduled if its parent task (who created it) has successfully committed its changes. This is because the child task is usually depend on the the data written by the parent task. Because changes to the shared memory are not visible before the parent task successfully commits, the child task cannot be started. A task will never be started if the parent task aborted, either intentionally or unintentionally.

In order to support flexible synchronization to represent complex task dependencies, L0 provide a mechanism of watcher type. A watcher type will monitor a set of memory addresses, each successful commits which changes the watched memory will guarantee to start a specified task. There might be more than one tasks launched for each commits if multiple addresses are modified, but the number of tasks started is not guaranteed to be the same as addresses changed.

The design of watcher type guarantees avoiding missing events. Unless a watcher is explicitly deleted, it persists in the system. As a result, once registered, a modification to the watched memory will create at least one task to process the event. This is different from “conditional task” which will be started only once.
The current implementation of L0 task scheduler provides parallel batch processing scheduling. The tasks are chosen by the FIFO order and scheduled to an idle computing node. Once a task is finished, another task (if exists) is scheduled immediately. No preemption is supported because saving the intermediate states can be very expensive. If the application requires low-latency responsiveness, the programmer need to avoid too many long-running tasks being created.

2.1.3 Runner Compartment and Binary Translation

Each task in L0 system is executed in a container called runner compartment. A runner compartment provides storage space to hold data content of transaction snapshot and processor time to execute a task. A runner will be assigned to a specific runner compartment when get scheduled.

Since the I0 code is not directly executable by the hardware or the host operating system, L0 use binary translation inside runner compartments to convert I0 into the native platform code. By using binary translation, L0 can provide better performance than using interpretation because the I0 will only be translated once and get executed at nearly native speed.

The I0 instruction set is designed to be flexible with as little constraints as possible. (The details of instruction set will be discussed in the next section.) However, actual hardware usually have constraints in instructions. For example, in x86 instruction set, specifying more than one memory operand is prohibited in most instructions. As a result, the binary translator must use multiple instructions to implement the functionality of one single I0 instruction.

The binary translation perform on-demand translating during execution. The binary translator does not translates a piece of code until it is executed. The translated code is saved in a reserved memory range inside the private region, so the translated code can be reused.

A challenge in the binary translator is how to deal with instructions jumping to an address which might not been translated. We use trampoline table technique to solve this. Each I0 branching instruction is translated into a native branching instruction into a stub routine. In the stub routine, the system map the target address of I0 into the target address of translated code, and perform on-demand binary translation if the target code is not yet translated.
In current implementation, a runner compartment is an individual process running in the host operating system. The runner compartment will hold data contents using the process address space and run the program using the processor time allocated to the process. A task will never migrate from one runner compartment into another. The binary translator supports translating I0 into Intel x86 64-bit instructions when run under x86_64 processors with 64-bit Linux operating systems.

2.2 I0: The Instruction Set of L0

I0 is designed for L0 as a new instruction set targeting a large-scale virtual machine running on a plurality of hosts connected through a datacenter network. In summary, ISA0 is scalable, efficient, general, simple, and Turing-complete. To ensure programmability, simplicity and efficiency, ISA0 includes a selected group of frequently used instructions, plus the newly added the newr and exit instructions to facilitate multitasking. ISA0’s architecture does not explicitly contain registers. This design provides a unified operand representation yet does not prevent register-based optimization because the memory model of L0 allows some addresses to be affiliated with registers. The operands can refer to immediate values or memory content using direct, indirect or base+displacement addressing mode.

2.2.1 Instruction Formats

Each instruction in I0 consists of an opcode, an optional instruction option, and optional operands, described as follows:

\[ \text{opcode[:option]} \ [\text{operand1} \ [, \text{operand2} \ ...]] \]

The opcode defines the function of the instruction. Depending on the opcode, there may be an option to specify the behavior of the instruction. For example, the \text{br} instruction can specify the \text{le} option indicating that the control flow should be branched to the target only if the less than or equal relation holds. And depending on the opcode and option, an instruction may require one or more operands.

2.2.2 Operands and Addressing Modes

In ISA0, there is no explicit register operands. The binary translator may map registers to the beginning of the private region. As a result, all operands in ISA0 can be expressed as
immediate values or memory operands. Memory operands can use 3 addressing modes –
direct, indirect, and base+displacement. The addressing mode are summarized as follows:

**Immediate**: Immediate addressing mode represents an immediate data specified in the
operand. Denoted by $address$.

**Direct**: Direct addressing mode represents data stored in a memory address specified by
the operand. Denoted by $address$.

**Indirect**: Indirect addressing mode represents data store in a memory address, which is
store in another memory address specified by the operand. Denoted by $(address)$.

**Base+Displacement**: Base+displacement addressing mode represents data stored in a
memory address, which is the value store in another memory address specified by
the operand added by the specified offset. Denoted by $displacement(base)$.

In most instructions in I0, operands are associated with an attributes to specify the
data type. Valid operand attributes are listed in Table 2.1

<table>
<thead>
<tr>
<th>ATTRIBUTE</th>
<th>DATA TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>muq</td>
<td>64-bit unsigned integer (used in mov instruction)</td>
</tr>
<tr>
<td>msq</td>
<td>64-bit signed integer (used in mov instruction)</td>
</tr>
<tr>
<td>mul</td>
<td>32-bit unsigned integer (used in mov instruction)</td>
</tr>
<tr>
<td>msl</td>
<td>32-bit signed integer (used in mov instruction)</td>
</tr>
<tr>
<td>mub</td>
<td>8-bit unsigned integer (used in mov instruction)</td>
</tr>
<tr>
<td>msb</td>
<td>8-bit signed integer (used in mov instruction)</td>
</tr>
<tr>
<td>uq</td>
<td>64-bit unsigned integer</td>
</tr>
<tr>
<td>sq</td>
<td>64-bit signed integer</td>
</tr>
</tbody>
</table>

Table 2.1: Operand Attributes. Attributes starting with letter ’m’ is only for mov in-
struction, other attributes are used for instructoins other than mov

### 2.2.3 General Instructions

Similar to other instruction sets, I0 provide a set of instructions to support data movement,
control transfer and computation. Table 2.2 lists general instructions in I0 instruction set.

### 2.2.4 Task Management Instructions

The I0 support instruction set level multitasking by providing *newr* and *exit* instructions.
<table>
<thead>
<tr>
<th>OPCODE</th>
<th>OPTIONS</th>
<th>FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>-</td>
<td>nop</td>
</tr>
<tr>
<td>mov</td>
<td>-</td>
<td>mov src, dst</td>
</tr>
<tr>
<td>add/sub/mul/div</td>
<td>-</td>
<td>OPCODE op1, op2, result</td>
</tr>
<tr>
<td>and/or/xor</td>
<td>-</td>
<td>OPCODE op1, op2, result</td>
</tr>
<tr>
<td>not</td>
<td>-</td>
<td>not src, result</td>
</tr>
<tr>
<td>shl/shr</td>
<td>-</td>
<td>OPCODE src, nbits, result</td>
</tr>
<tr>
<td>br</td>
<td>j/ji/ne/le/l/e</td>
<td>br:ji target</td>
</tr>
<tr>
<td></td>
<td></td>
<td>br:ne/le/l/e op1, op2, target</td>
</tr>
</tbody>
</table>

Table 2.2: General Instructions in I0

The *newr* instruction is used to create a new task which will become scheduled after the current task successfully commits the changes. It has the following format:

*newr* stack, heap, watching, start

The operand *stack* is a pointer (address) to the stack metadata, which defines the stack range. The format of the metadata is a 16-byte memory block, with first 8 bytes storing the stack base address, and the next 8 bytes storing the stack size in bytes. The operand *heap* operand is a pointer to the heap metadata, which is a memory block starting with a 64-bit integer indicating the number of ranges, and all the ranges are followed by, with each of them has a 64-bit base and 64-bit size. The *watching* operand is a pointer to the watching list used for creating watchers, and it defines the watching ranges, with the same format of heap ranges metadata. The last operand *start* is a pointer to the starting address of the new task.

The *exit* instruction is used to terminate current task, it has the format

*exit*:OPTION

The option of *exit* instruction specifies the transaction action upon exits. The valid options are listed in Table 2.3:

<table>
<thead>
<tr>
<th>OPTION</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Commit the transaction.</td>
</tr>
<tr>
<td>a</td>
<td>Abort the transaction.</td>
</tr>
<tr>
<td>cd</td>
<td>Commit the transaction and delete the watcher. Only valid in a task started by a watcher.</td>
</tr>
<tr>
<td>ad</td>
<td>Abort the transaction and delete the watcher. Only valid in a task started by a watcher.</td>
</tr>
</tbody>
</table>

Table 2.3: General Instructions in I0

12
If the transaction memory manager detects conflicts, *exit* instruction will always abort the transaction regardless of the option specified. In this case the watcher is not deleted even if *ad* or *cd* option is given.
CHAPTER 3

DISCIPLINED RUNTIME

This chapter describes how we design and implementation the Disciplined Runtime following the principle of disciplined memory access on top of L0, including the programming language, the compiler, and the library and application framework. In Disciplined Runtime, the memory access information is generated with the help of programmers and the compiler. The memory access information is used in many components of the system, such as the distributed shared memory module and the task scheduler, to help the L0 system to achieve good overall performance.

3.1 Overview

We use L0 [26] as our basic building block for the execution engine part. We choose to design our framework on top of L0 because it provides necessary architectural and system-level support which is essential for implementing a framework following the disciplined memory access model. Most important, as required by the principle of disciplined memory access, each parallel task should be able to specify the accessible memory locations at runtime, so the system must know the concept of multitasking, and have some mechanism to specify the memory ranges for each parallel task. The runner mechanism in L0 have implemented all necessary multitasking facilities required by the disciplined memory access principle. As we mentioned in the previous chapter, a task in L0 system should have a memory transaction which specifies all the memory ranges needed to be maintained by the transaction. This information must be specified at the creation of a task when the input is known but before the task starts. This behavior satisfy the requirement of task creation in a system following disciplined memory access principle, that the memory access pattern is provided to the runtime system according to the specific input. As a result, this character can be just directly used to implement the functionality of disciplined memory access. By using L0 as our system-level support, there is no need to invent the wheel again.

On top of the L0 execution engine which uses I0 as the instruction set, we build Disciplined Runtime, which is the proposed programming framework and utilities en-
abling datacenter-scale programming. It consists of a data-centric programming languages named as C0, a compiler named cc0, some necessary libraries and application frameworks. The L0 and Disciplined Runtime together form a complete facility to materialize the concept of disciplined memory access. Figure 3.1 (a) illustrates the architectural of Disciplined Runtime.

In this framework, applications are written in the specially designed C0 programming language, then the cc0 compiler generates the I0 binary code ready to in L0. Inside L0, the I0 code will be further translated into platform code because currently there is no hardware support for I0 instruction set. This process is illustrated in Figure 3.1 (b).

### 3.2 C0 Programming Language

The disciplined memory access introduces new principles to the distributed applications, and the programmers needs new developing tools to follow this principle. Specifically, the programmers need a way to declare the accessible memory ranges when creating a new task entity without knowing the actual input. The system must instantiate these memory ranges when the input is known at runtime.

To support this new programming model, we designed a new programming language named C0, which is derived from C programming language. The reason we follow the C style is to provide capabilities of system-level programming and familiar programming
methodology to port existing applications. However, to support the disciplined memory access, we need to add some key features to the standard C programming language, including:

- Language-level multitasking, with explicit memory accessibility declaration and dependency description.
- Partial reference to a large memory block associated to array variables.
- Memory layout hints to avoid false sharing.

To support these features, we introduce special syntax and language structures into C programming language. The following code shows an example of how to program with explicit memory access declaration to follow the disciplined memory access principle, using the proposed C0.

```c
long n;
standalone double a[4096];
void double_it(long start,long end)
    long i;
    for (i=start;i<end && i<n;i=i+1)
        a[i]=a[i]*2;
    commit;
}
void main() {
    long i;
    n = 4096;
    for (i=0;i<n;i=i+512)
        runner double_it(i,i+512) using n, a[i,,i+512];
    commit;
}
```

This program double each elements in an array in parallel, with each parallel task processing 512 elements. The statement in line 17 inside the for loop creates a new task, and line 18 specify the accessible variables of the new task. In this example, the new task will access the variable `n` and a segment of the array `a`. The new syntax is discussed in details in this section. For formal grammar description of the C0 language, please refer to Appendix A.

### 3.2.1 Language-Level Multitasking with “runner” statement

The disciplined memory access model requires memory access information to be provided when creating new tasks. The programmers should be able to express the templates of
memory access ranges in a clear and straightforward way. In other words, there should be some necessary features to express the concept of parallel tasks and memory access in the language level.

This requirement motivates us to construct a programming language with language-level support of multitasking. Specifically, in C0, the keyword `runner` can be used to queue a new task into the scheduler, and the new task will become schedulable after the current task successfully commits. Upon exit of a task, the programmer can specify the transaction action with `commit` or `abort` statement. The following example shows how to create a new task running `func` function and commits the current transaction.

```c
void func()
{
    ...
}
void main()
{
    runner func();
    commit;
}
```

### 3.2.2 Accessible Memory Declaration with “using” clause

The availability of global variables is different between C and C0. In C, if there is no name duplication between a local variable and a global variable, the program can always access this global variable. In C0, however, a runner cannot access a global variable unless this variable is declared to be within the task transaction. This is the requirement of disciplined memory access.

When creating a new task, the `using` clause can be used to specify the memory accessibility of the new task. Each variable that the new task wants to access must appear in the `using` clause. When the program executes, the memory ranges instantiated with the specific runtime value of variables, together with the implicit calling stack, are used to create new parallel tasks in L0. At any time, if a task accesses a memory location outside the pre-declared ranges, the behavior is undefined. The following example illustrate the usage of “using”. In this example, the new task will have access to global variable `a` but not `b`.

```c
long a;
long b[10];
void func() {
    a = 2;
    // b[3] = 3; // Undefined behavior!
```
### 3.2.3 Task Dependencies and Watchers with “watching” clause

Usually there are logic dependencies among parallel tasks in an application. Some tasks must be executed after some specific tasks successful finish. Although we already enforce child tasks be executed after the parent tasks, this dependency mechanism is not sufficient. To support more general task dependency, a wait-notify mechanism is required.

The L0 provides the watcher mechanism which can be used to create tasks when a specified memory location has been touched. In the C0 language, this feature can be used with the `watching` clause together with `runner`, which registers a watcher type in L0, indicates the specified task will be started when a specified variable has been modified. And inside a task created by a watcher type, additional transactional actions `commitd` and `abortd` can be used to delete the watcher type if needed. An example of “watching clause” is as follows. The function of this example is waiting for a variable to be 100. The function `check_it` created by a watcher type checks whether the value of `flag` is 100, if not, it simply aborts itself without deleting the watcher type so that further changes to the variable can be captured correctly.

```plaintext
long flag;
void check_it() {
    if (flag!=100)
        abort;
    do_next_step();
    commitd;
}
void main() {
    runner check_it() using flag watching flag;
    commit;
}
```

### 3.2.4 Partial References via Array Segments

A common pattern of dividing data in a parallel application is letting each task to process a portion of the entire data set. In the language level, the programmer need to specify a range which is a part of a large data structure such as an array. In this paper, we also
consider a pointer to a large block of memory an general array. As a basic element in the programming language, an array can be used to implement more sophisticated data structures such as a hash table or a tree. In the language level, we support representing a segment of an array when specifying memory ranges, in the form of array segments.

The array segment provides the memory access control in an element granularity without forcing the programmers to use pointers explicitly. Without array segment, the programmer must specify the range with a pointer pointing an address in the array and a range length. Dealing with this requires the programmers consider converting array indexes in the parallel tasks, which is usually error-prone. With the help of array segments, the programmer can directly express a part of a large array, which is more straightforward and easier to understand.

An array segment is logically same as an array (or a pointer). However, it restricts the access of elements to a specified range. The array segment is represented as array[start,,end], the start is inclusive and end is exclusive. The following example illustrate the usage of array segments.

```c
int a[10000];
int i;
for (i = 0; i < 10000; i++) {
    a[i] = i;
}
int* seg = a[100,,200];
int b = seg[140]; // b has the value 140
int c = seg[20]; // Undefined behavior
```

### 3.2.5 Avoid False Sharing via Standalone Variables

Although the memory ranges can have arbitrarily boundaries, in most system implementations, the granularity of a snapshot is in the page-level in order to utilize the hardware MMU features. As a result, even though two memory ranges are disjoint, they may overlap after being aligned to the page boundary. This causes the false sharing issue, which means two transactions with disjoint memory ranges will conflict with each other due to page-alignment of memory ranges.

We introduce standalone memory objects which means the objects must be allocated to separate memory pages different from other objects. The standalone memory objects do not share memory pages with other objects, therefore false sharing is avoided because the entire memory page is exclusively occupied and the page-aligned ranges will not overlap.
if the original ranges are disjoint. In the following example, `task_a` and `task_b` will both successfully commits because variable `a` and `b` will be allocated on separate memory pages. Without `standalone`, at least one task will abort because variable `a` and `b` will be allocated into the same memory page.

```c
standalone long a;
standalone long b;
void task_a() {
    a = 1;
    commit;
}
void task_b() {
    b = 1;
    commit;
}
void main() {
    runner task_a() using a;
    runner task_b() using b;
    commit;
}
```

Although standalone objects add some padding between the variables, we consider the space overhead acceptable. There are two scenarios the programmer want to use the standalone memory object. The first one is some small global shared variables. The number of these variables will not be large and the space padding is consequently not a problem. The other one is the large array. Different array objects do not share memory pages. Since the padding space is relative small compared to the array size, the space waste is also acceptable.

### 3.2.6 The “main” Function

In C0, the `main` function, similar to the `main` in C, is the entry point of a source program. However, it takes no argument and does not return a value. All inputs and outputs of a C0 program should be passed via the standard input/output which is the only I/O capability of L0.

In C0, the `main` function is executed in a normal runner, without any memory range on shared region. This is equivalent to creating a runner without a using clause. As a result, `main()` cannot use any global variables. The following code produce unexpected behavior, usually a program crash.
3.3 CC0: C0 Language Compiler Targeting L0

The C0 introduces new language features into standard C, and it targets a new instruction set I0. In order to use C0 to write programs for L0, a new compiler is needed. The prototype compiler we have designed and implemented is named as cc0.

Since both ISA0 and C0 are subject to change in future to accommodate new designs, the compiler must be implemented in a flexible way to deal with potential changes. Also, the compiler must be extensible for future extension such as code optimizations or analysis. In order to provide flexibility, our compiler is designed as a multi-staged architecture which uses an intermediate language (IL) between the frontend and the backend. When compiling, the source code is first translated to the IL, and the middle-part will perform necessary checks, optimizations and analysis on the IL. Finally, the IL are then translated to ISA0. The IL is designed to be language and platform independent to benefits future extensions.

3.3.1 Source Parsing

The purpose of the frontend is parsing the source code and translating it into valid IL. Syntax and semantic checking must also be performed during the translation. A common approach to implement a compiler frontend is to generate an abstract syntax tree from the parser, and traverse the tree to perform various checking and generate the IL. In cc0, however, we consider another issue. Many programming language with different keywords, syntaxes, grammars actually shared the same language structure in terms of semantics. For example, Although C and Pascal are quite different in their syntax, the general language structures are very similar.

We want the compiler frontend to separate the concern of grammar and semantics. In order to achieve this goal, the frontend is also divided into two phases. The first phase translate the source code into an expression tree, which is the grammar-independent representation of the source code structure. The expression tree is ideally capable to
express most of common programming languages, either imperative or functional, and is extensible to deal with uncommon language structures. An expression tree is made up of common language elements such as functions, statements, branches, loops, expressions, etc. In the second phase of the frontend, the expression tree is traversed several times. Semantics checking, such as type checking, together with some optimizations like constant expression propagation, is done at this stage. The expression tree is finally translated into the IL if no semantic error is found.

The frontend is also responsible to build the symbol table of the program, some symbols are assigned a fixed address, addresses of these symbols are directly recorded into the symbol table at this stage. The address allocation is done according to the length of the variable, each variable is allocated to space the same size as its length, routed up to the pointer size. A special case is standalone variables, which will monopolize a whole memory page.

Symbols are marked as unresolved if the address cannot be determined immediately. An example of undeterminable address is the jump target, which can only be determined after the final code of the target has been generated. The unresolved symbols must be resolved during the code generation.

### 3.3.2 Intermediate Language (IL) Processing

The intermediate language (IL) representation generated by the frontend is a type-safe language which can be easily translated into ISA0 or other machine dependent instructions. It can represent data movement, arithmetic operations, function calling, etc.

Each IL instruction consists an opcode. The opcode defines the function of the instruction. Depend on the opcode, there may be some operands of the instruction. There are four kinds of operands in an current IL design:

- **Constant**: a constant operand stores a constant value used by the IL instruction
- **Symbol reference**: a variable reference specify the variable name which would have a determined value at runtime.
- **Runtime type**: a runtime type operand is used in some IL instructions (e.g., type casting) to specify the runtime type.
<table>
<thead>
<tr>
<th>TYPE</th>
<th>LENGTH</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>N/A</td>
<td>The operand has no type</td>
</tr>
<tr>
<td>I</td>
<td>Same as the address size of target machine</td>
<td>Signed pointer address</td>
</tr>
<tr>
<td>U</td>
<td>Same as the address size of target machine</td>
<td>Unsigned pointer address</td>
</tr>
<tr>
<td>I1/I2/I4/I8</td>
<td>1/2/4/8</td>
<td>Signed integer types</td>
</tr>
<tr>
<td>U1/U2/U4/U8</td>
<td>1/2/4/8</td>
<td>Unsigned integer types</td>
</tr>
<tr>
<td>R4</td>
<td>4</td>
<td>IEEE 32-bit floating-point</td>
</tr>
<tr>
<td>R8</td>
<td>8</td>
<td>IEEE 64-bit floating-point</td>
</tr>
<tr>
<td>Str</td>
<td>Variable</td>
<td>String</td>
</tr>
<tr>
<td>Struct</td>
<td>Variable</td>
<td>User-defined compound type</td>
</tr>
</tbody>
</table>

Table 3.1: IL Operand Types

**Empty:** an empty operand means this operand is missing and unused in the IL instruction.

Each IL operand has a type defining the length and data type of the operand. All valid operand types are listed in Table 3.1.

The IL is designed to be extensible, so besides general instructions that will directly map to hardware features, there are an IL instruction **asm** which can embedded platform dependent assembly instructions into the IL. All the IL instructions are listed in following table (span over multiple pages).

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>FORMAT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>nop</td>
<td>No operation.</td>
</tr>
<tr>
<td>label</td>
<td>label op1</td>
<td>Misc IL. Place a label.</td>
</tr>
<tr>
<td>mov</td>
<td>mov op1, op2</td>
<td>Data copy.</td>
</tr>
<tr>
<td>conv</td>
<td>conv op1, op2</td>
<td>Data convert. Convert op2 to the type of op1, and store the value into op1.</td>
</tr>
<tr>
<td>add/sub</td>
<td>OPCODE op1, op2, op3</td>
<td>Arithmatic operations.</td>
</tr>
<tr>
<td>/mul/div/mod</td>
<td></td>
<td>op1 ← op2 OPCODE op3</td>
</tr>
<tr>
<td>clt/cgt/cle</td>
<td>OPCODE op1, op2, op3</td>
<td>Comparison operations.</td>
</tr>
<tr>
<td>/cge/ceq/cne</td>
<td></td>
<td>op1 ← 1 if the comparison relationship holds, otherwise op1 ← 0.)</td>
</tr>
</tbody>
</table>

continued on next page
<table>
<thead>
<tr>
<th>OPCODE</th>
<th>FORMAT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>blt/bgt/ble /bge/beq/bne</td>
<td>OPCODE op1, op2, op3</td>
<td>Conditional branch. Compare op1 and op2, then jump to op3 if the comparison relationship holds.</td>
</tr>
<tr>
<td>bz/bnz</td>
<td>OPCODE op1, op2</td>
<td>Conditional branch. If op1 is zero/non-zero, jump to op2.</td>
</tr>
<tr>
<td>br</td>
<td>br op1</td>
<td>Unconditional branch. Jump to op1</td>
</tr>
<tr>
<td>shl/shr</td>
<td>OPCODE op1, op2, op3</td>
<td>Shifting operations. (op1 ← op2 OPCODE op3)</td>
</tr>
<tr>
<td>and/or/xor</td>
<td>OPCODE op1, op2, op3</td>
<td>Bitwise logical operations. (op1 ← op2 OPCODE op3)</td>
</tr>
<tr>
<td>not</td>
<td>not op1, op2</td>
<td>Bitwise inverse operation. (op1 ← op2)</td>
</tr>
<tr>
<td>call/callind</td>
<td>OPCODE op1, op2, op3, op4, ...</td>
<td>Function direct/indirect calling. op1 is the variable to hold the return value, and it can be empty if the function does not return a value or the return value should not stored. op2 is the callee, for direct call, it is the function symbol or absolute address, for indirect calling, it is a function pointer. op3 and afterwards are the arguments passed to the function.</td>
</tr>
<tr>
<td>ret</td>
<td>ret op1</td>
<td>Return to the caller. op1 is the return value. If the function does not return a value, op1 should be empty.</td>
</tr>
<tr>
<td>ldind</td>
<td>ldind op1, op2</td>
<td>Indirect loading. (op1 ← *op2)</td>
</tr>
<tr>
<td>lda</td>
<td>lda op1, op2</td>
<td>Get the address of op2, and store it to op1. (op1 ← &amp;op2)</td>
</tr>
<tr>
<td>ldelem</td>
<td>ldelem op1, op2, op3</td>
<td>Load array element. (op1 ← op2[op3])</td>
</tr>
<tr>
<td>stelem</td>
<td>stelem op1, op2, op3</td>
<td>Store array element. (op1[op2] ← op3)</td>
</tr>
<tr>
<td>ldelema</td>
<td>ldelema op1, op2, op3</td>
<td>Get the address of array element. (op1 ← &amp;op2[op3])</td>
</tr>
<tr>
<td>ldfld</td>
<td>ldfld op1, op2, op3</td>
<td>Load struct field. (op1 ← op2.op3)</td>
</tr>
<tr>
<td>stfld</td>
<td>stfld op1, op2, op3</td>
<td>Store struct field. (op1.op2 ← op3)</td>
</tr>
<tr>
<td>ldflda</td>
<td>ldflda op1, op2, op3</td>
<td>Get the address of struct field. (op1 ← &amp;op2.op3)</td>
</tr>
<tr>
<td>asm</td>
<td>asm op1</td>
<td>Inline assembly. op1 is the assembly string.</td>
</tr>
</tbody>
</table>

During the IL processing, the IL will be traversed and transformed for one or more times. These transformation will perform optimizations and analysis for the program. Currently, the compiler implementation supports very limited code optimizations, but still emit code outperforming existing systems. We believe the performance can be higher...
3.3.3 Code Generation

The final step of the compiling is to translate the IL into I0. The IL is designed to be flexible without unnecessary constraints, however, there are some constraints in I0. As a result, the translation is not that straightforward. For example, the ISA0 arithmetic instructions can not support 32-bit integers. In this case, the 32-bit integer is first converted into 64-bit integer and stored in a temporary location using the `mov` instruction, after performing the arithmetic operation, the 64-bit integer must be truncated to the original length and stored to the required address.

The symbol resolving is also performed in this stage. When the code generator hits a label or a function entry, it will lookup whether there are unresolved symbols referencing the generating label. If found, the symbol address is filled into the symbol table and all previous reference is modified by filling in the real address, which can be determined at this time. After the code generator finish generating the entire program, it will make sure there is no unresolved symbol left. If there is some, the code generator reports an error of unresolvable symbol.

3.4 Runtime Libraries

Runtime libraries provide some basic functionalities to support the application, including some common features that most applications will use, and some components prepare the runtime environment. In our prototype framework, we only implement very preliminary features in the library.

Logically, the application starts running from the `main` function. However, there are some preparation work to be done before `main` can start. For example, to support dynamic memory management (`malloc` and `free`), the application must initialize the memory pool data structure.

The purpose of the start-up code is depend on the runtime features provided. Currently, we perform following tasks during the start-up code before jumping to `main`.

- Initialize data structures necessary for `malloc` to function properly.
- Initialize the random number generator.
The application usually need to allocate memory blocks at runtime. In traditional C program, one can use `malloc` to do this. However, in C0, the semantics of `malloc` is changed because it is not possible to dynamically change the task memory space, as required by the disciplined memory access principle. Instead, we provide two different library functions to allocate memory space dynamically.

- **prmalloc**: Allocates a memory block in private region (PR), and the allocated memory block is only usable by the current task.

- **srmalloc**: Allocates a memory block in shared region (SR) outside of current task’s memory space. However, the allocated memory block cannot be used by current task. It can only be assigned to a new task created later on.

- **plmalloc**: Allocates a memory block from an existing memory range within current task’s memory space, either in PR or SR but usually in SR. The allocated memory block can be used immediately by the current task.

### 3.4.1 Standard C Library

We also choose to implement some of the standard C library functions that is widely used. The functions used by the benchmarks in our evaluation get higher priority to get implemented. These functions includes:

**Memory block operations**: Including `memcpy`, `memset`, etc.

**Mathematical operations**: Including `rand`, `abs`, etc.

### 3.5 Implementation

The compiler and the library require significant work to complete, Table 3.2 summarizes the implementation code base. In implementing `cc0`, we take portability as an important consideration. This allows our technique benefit developers working under various platforms. In order to achieve the goal of portability, we select cross-platform programming

<table>
<thead>
<tr>
<th>COMPONENT</th>
<th>LOC</th>
<th>LANGUAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>12547</td>
<td>C++</td>
</tr>
<tr>
<td>Library</td>
<td>366</td>
<td>C0</td>
</tr>
</tbody>
</table>

Table 3.2: Implementation metrics
languages and tools to implement the system. We choose CMake \(^1\) 2.8 as the build system. CMake is a cross-platform, open-source build system to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice. For example, it generates GNU Makefile under Linux platforms, and Visual Studio solution files under Windows platforms.

The source parser of the frontend is implemented with the help of ANTLR \(^2\) 3.4, which is a language tool that provides a framework for constructing grammar recognizers from grammatical descriptions. It generates an LL parser is called an LL(*) parser which does not restricted the number of lookahead tokens to a finite number, but can make parsing decisions by recognizing whether the following tokens belong to a regular language. Other parts of the compiler are implemented using standard ANSI C++ without any platform dependent features or libraries. The CMake build system will automatically use proper C++ compiler. For example, in Linux platform, it uses GCC \(^3\) and Binutils \(^4\) to compile and link.

The C0 runtime library and application framework are implemented in C0 and compiled in cc0. Since cc0 can be built on various platforms, these library can be generated for different platforms. In the current implementation, we simply ships these library and application framework as the form of C0 source header files. So in order to use them, one can simply include the header files in the source code.

\(^1\)http://www.cmake.org/
\(^2\)http://www.antlr.org/
\(^3\)http://gcc.gnu.org/
\(^4\)http://www.gnu.org/software/binutils/
CHAPTER 4

EVALUATION

The goal of the disciplined memory access principle and the Disciplined Runtime is to enable the utilization of memory access information to provide better multitasking performance on distributed platforms. In this section, we evaluate the following aspects of the disciplined memory access principle and our prototype Disciplined Runtime implementation:

1. **Generality.** Can disciplined memory access and Disciplined Runtime support different types of applications in various problem domains? Our system should be able to accommodate not only “embarrassingly parallel” workloads, but also be able to perform sophisticated computations. Also, the additional programming effort requirement should not introduce too much burden to programmers.

2. **Performance.** What is the cost of following the disciplined memory access principle? What’s the execution performance of the Disciplined Runtime compared to other solutions? The disinclined memory access concepts should not necessarily implies additional overhead.

3. **Scalability.** Can the Disciplined Runtime scale to environments with a large number of compute nodes? Our system should be able to run efficiently on both small clusters and large datacenters.

We use both a research testbed and the Amazon EC2 cloud platform to evaluate the our system. The experiments are carried out on up to 32 physical machines and 256 EC2 instances. In our research testbed, each node is equipped with two Intel Xeon quad-core processors supporting hyperthreading, and has 32GB memory with swapping disabled. The nodes are connected with a 10Gbps network. The EC2 instances we used are in the same zone, and each instance is of the extra large instance type with 4 cores (8 ECUs) and 8GB RAM.

Considering the system randomness, in the data collecting process, we measure the elapsed time and execution time for of the benchmark programs, excluding the data
Table 4.1: Benchmark Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>matmul</td>
<td>matrix multiplication</td>
<td>8K × 8K matrixes (two inputs and one output)</td>
</tr>
<tr>
<td>pathfinder</td>
<td>shortest path</td>
<td>256K x 256K grid</td>
</tr>
<tr>
<td>kmeans</td>
<td>k-means clustering</td>
<td>64M 2D points, with k = 1024</td>
</tr>
<tr>
<td>pagerank</td>
<td>Page ranking</td>
<td>256K web pages</td>
</tr>
</tbody>
</table>

initialization time, as the measurement of performance and scalability. For each data item, we execute the benchmark three times, make sure the extreme difference of the three data items is less than 10% of the mean value, and use the mean value as the final result.

4.1 Generality

In order to show that the disciplined memory access principle and the Disciplined Runtime can support general applications, we use different benchmark applications throughout the experiments. To ensure generality, we choose applications in different various problem domains from scientific mathematical calculation, machine learning algorithms to graph algorithms. Each benchmark is widely used for evaluating the system performance by many other researchers. Table 4.1 lists the characteristics of the benchmarks.

The matmul program calculates the product of two matrixes. Matrix multiplication is one of the most important basic building blocks in scientific programs. Many algorithms can be decomposed as a series of matrix operations.

The pathfinder program finds the shortest path on a 2-dimension grid from the bottom row to the top row where each step of the path moves straight or diagonally forward. It is modified from the OpenMP version of the Rodinia [13,14] benchmark suite which is an computation-intensive applications.

The kmeans program is a clustering algorithm showing a high degree of data parallelism. It is used extensively in data-mining.

The pagerank program is a link analysis algorithm used by the many search engine. It assigns a numerical weighting to each element with the purpose of measuring its relative importance.

All these benchmark applications can be correctly supported in the Disciplined Runtime designed under the disciplined memory access principle. The next two sections show
that the performance of all these applications are better than existing platforms. This demonstrates that the Disciplined Runtime is able to support general computation for various problem domains.

4.2 Performance

We use all the benchmark programs mentioned previously to evaluate the performance of the system. We carry out experiments on our research testbed using up to 32 physical nodes. We use the inverse of the elapsed time as the measure of performance. For comparison, we include the performance on the Hadoop and X10 platforms. And all the performance is relative to those of Hadoop program using one node. The input data sizes used for the experiment are also listed in Table 4.1. The relative performances of different applications are shown in Figure 4.1.

Figure 4.1(a) shows the performance of matrix multiplication program executed in our research testbed. The matrix multiplication is a highly parallel algorithm. The
performance on the Disciplined Runtime is about ten times faster than that in Hadoop and X10. As indicated by the slopes of the lines in logarithmic axis, the scalability of the the Disciplined Runtime is as good as Hadoop and X10. Figure 4.1(b) to 4.1(d) shows similar results for other benchmark applications.

In order to reveal the reason why the Disciplined Runtime is much faster than Hadoop and X10. We carry out performance profiling to measure the different overheads of all the three systems. We run the benchmark programs at 8-node configurations, and use performance profilers to examine the time cost structure of a task running in one single node.

The profiling result indicates that the most important factor is the overhead of data fetching. The cost of data transfer varies hugely in different platforms. Figure 4.2 shows the time spent on different operations in all the benchmark applications. The computation part is the time to execute the application logics. The data transfer part is the cost of fetching data, which means loading data from and saving data to HDFS in Hadoop, serializing data in X10 and fetch memory content from remote node in the Disciplined Runtime. From the results, we can see that, due to lack of optimization, the programs run in Disciplined Runtime execute computation tasks slower than in Hadoop and X10, which are based on backends with mature single-node optimizations. However, because of high efficiency of dealing with distributed data, the overhead in the Disciplined Runtime
Figure 4.3: Scalability of Amazon EC2

is much smaller, while Hadoop and X10 introduce much higher overhead. Hadoop avoids using in-memory shared states, and all the intermediate results must be accessed via disks which is much slower than memory. In X10 using PGAS model, the data belonging to one place cannot belong to another, and it must be serialized through the network if it needs to be migrated from one place to another. This serialization introduces large overhead, which reduces the performance. This experiment shows that the significant better performance in the Disciplined Runtime is because of better data management and scheduling, which provide better locality and more efficient multitasking.

4.3 Scalability on Amazon EC2

To evaluate how the Disciplined Runtime scales on platforms with a large number of nodes, we deploy the Disciplined Runtime on Amazon EC2 [2] and carry out experiments using matmul and kmeans applications. Amazon EC2 is a widely used cloud computing platform which can support thousands of virtual nodes. In this experiment, the input data sets are larger than those in our research testbed. The input of matmul is two matrixes of 32K x 32K size, and the input data of kmeans is 64M 2-dimension points with $k=8192$. The results are shown in Figure 4.3.

Figure 4.3(a) shows the performance of matrix multiplication on the Amazon EC2 platform using the Disciplined Runtime. Matrix multiplication is a high parallel algorithm with theoretical linear speedup, however when executed on very large clusters, the relative small serial part of the application can have a significant impact, which means the actual performance speedup is sub-linear. The implementation of current matrix multiplication benchmark introduce about 0.5% computations which can not be parallelized,
and according to this overhead we can calculate the ideal speedup as a comparison. The results shows that using disciplined memory access, the speedup of the matrix multiplication run in the Disciplined Runtime has little difference from the calculated ideal speedup. For example, the ideal speedup for 256 nodes is $1/(0.995/256 + 0.005) = 112$, and the actual speedup is 105.

Figure 4.3(b) shows the performance of k-means clustering on Amazon EC2 platform. The k-means clustering algorithm is more computation-intensive than matrix multiplication. It shows a higher degree of parallelism. The experiment result shows that the Disciplined Runtime can provide over 200x speedup at 256 nodes indicating satisfactory scalability.
CHAPTER 5

RELATED WORK

In this chapter, we discuss some state-of-the-art work that closely related to the proposed one in this thesis. The solutions are well-known by the academic and/or industrial world all focus on the challenge of efficient programming in large datacenter. We introduce these projects and discuss how our technique differ from them.

5.1 Hadoop and Dryad

MapReduce [16] is a framework of processing parallelizable workloads with huge datasets. MapReduce organize computation tasks in map and reduce operations. Provided that each mapping/reducing operation is independent of others, all map/reducing operations can be parallelized very efficiently. As a result, this parallelism can offer significantly larger capability than what common servers can handle. Hadoop [3] is a widely used free and open source implementation of MapReduce in Java, and it is used by many commercial companies like Yahoo! [23] and Facebook [22].

In the map step, the input is divided into smaller parts and distributed to different worker nodes. The worker node solves a small sub-problem by processing a small part of the input data. Different worker nodes do not communicate with each other during processing, avoiding any overhead related to distributed coordination. In the reduce step, the results produced by the map step are collected and combined in some way to generate the final output, which solve the original large problem trying to solve. Provided that different map operations are independent with each other, theoretically all map tasks can be performed in parallel. Similarly, provided that outputs of map operation sharing the same key are processed by the same reducer, the reduce operations can also run in parallel.

In the meantime, however, the design of simplified application workflow limits the expressiveness of MapReduce framework. The process of solving a problem in MapReduce is usually less efficiency than the corresponding sequential algorithms, because in very rare cases an algorithms can be expressed efficiently using only map and reduce operations.
There are even many parallel algorithms that cannot be directly implemented in MapReduce, these algorithms will use some features that MapReduce cannot support, such as mutable global states or conditional loops. In order to work around these limitations, programmers need to use other programming language and/or frameworks to write glue code to incorporate many MapReduce tasks together. This imposes a burden for solving complex problems in MapReduce framework.

As one of the most typical implementation of MapReduce framework, Hadoop has some implementation choices which made MapReduce not satisfactory in modern datacenter computing. For example, it is based on an earlier design context where physical memory and high-speed networking were very expensive [6], so it uses HDFS to store all the intermediate results onto the disk. This makes the single-node performance of Hadoop not as good as it could be.

Dryad [24], developed by Microsoft Research, is a general purpose runtime for execution of data parallel applications. It is also require the program to be written as a dataflow, but it is more expressive than MapReduce. Rather than forcing the program to be written in map and reduce operations, the structure of Dryad jobs can be in any directed acyclic graph (DAG) form. A vertex in the DAG, namely “computational vertex” is written in sequential code without concurrency or synchronization. The runtime parallelize the computation by distribute different computational vertices to different cores or nodes. defines the dataflow of the application

It is difficult and inefficient to write the application using the DAG. To support better programming experience, DryadLINQ [31] is a LINQ [8] implementation using Dryad as the back-end. It is difficult to directly express loops and mutable states in the DryadLINQ framework, so DryadLINQ is integrated into other high-level programming languages such as C#.

Different from MapReduce and Dryad, Disciplined Runtime built on top of L0 use much more flexible programming model and is capable to utilize modern hardware resources such as physical memory and CPU registers. In C0, programmers can write any logic that can efficiently express that application logic. The L0 system uses uniform memory addressing model, but the virtual address can be mapped into different resources such as distributed shared memory, local memory, registers. It is even feasible to implement persistent memory facility by mapping the virtual memory address to disk files.
5.2 X10

X10 [12] is a programming language developed by IBM for parallel programming using the partitioned global address space (PGAS) model. The PGAS model assumes that portions of the memory space have some kind of affinity and locality. X10 provides much more flexible programming model than dataflow solutions, it has a strong type system and supports general object-oriented programming. It has some features specially designed for parallel programming, such as globally distributed arrays.

In X10, a computation is divided among a set of places. Each place holds part of the data and executes some activities that process those data. X10 uses the concept of parent and child relationships for activities to prevent deadlocks. An activity cannot wait for a parent activity to finish, but a parent can wait for a child using the finish command.

However, there are also some limitations to use PGAS in sophisticated distributed programming. In X10, the partition of the memory space must be determined at compile-time, which is sometimes difficult because in some applications the data access pattern is undecidable until the input is known. X10, also requires the partitioned ranges to be disjoint and static, which means a memory address belongs to only once place throughout the entire lifecycle of the program, this make difficult to write some applications that locality preference may change according to different input or during different phases of the program.

Disciplined Runtime impose much less constraints in how to use memory, and allowing managing distributed data more efficiently. When programming L0 using C0, each task can define it’s own disciplined memory space dynamically according to the application logic and the input. A memory address may be used by more than one tasks provided that only one task modifies it. The task scheduler can dynamically schedule each task to where most of its data locates.

5.3 vNUMA

vNUMA [11] is a distributed shared memory (DSM) approach built on the virtual machine level. vNUMA simulates a single virtual NUMA machine on multiple physical machines to provide shared memory in the virtual machine hypervisor.

vNUMA is designed to make legacy applications utilize the power of distributed sys-
tems. It can virtualize a single system using a legacy operating system written in the Itanium instruction set without significant modifications. vNUMA uses distributed shared memory (DSM) techniques to provide NUMA shared memory.

In the experiments in a small clusters, vNUMA provides satisfactory performance. However, the Itanium instruction set is designed for processor running on a single node. It is based on the characteristics of the hardware in a single machine. However, these characteristics may not still be valid in a distributed cluster which is quite different. As a result, vNUMA is not designed for scaling to hundreds of nodes.

The Disciplined Runtime made another design choice. Rather than provide compatibility to existing software, we choose to design the system to best incorporate large-scale datacenters. The instruction set I0 used by L0 is not compatible to any existing softwares. However, the instruction set level multitasking and transaction semantics can be much more efficient to support a system following the disciplined memory access principle. Although new compiler is needed and programs need to be ported, Disciplined Runtime has much better scalability, which can utilize hundreds of nodes efficiently.

5.4 JVSTM

Software transactional memory (STM) is a concurrency mechanism to access to shared memory in concurrent way. As an alternative to lock-based synchronization, a transaction executes a series of reads and writes to shared memory logically occur at a single instant in time. Any intermediate (uncommitted) states are not visible to other transactions. JVSTM [9] is a Java library that implements software transactional memory, which allows transaction programming at the programming language level independently of an external transaction manager.

JVSTM proposes the use of Versioned Boxes, which keep a history of values, as the basis for language-level memory transactions. It also reduces transaction conflicts by delaying computations and re-executing only parts of a transaction in case of a conflict.

The transactional memory interface implemented in JVSTM is not transparent to the program. In order to read or write the transactional memory states, one must use Versioned Boxes rather than normal variables defined in the source code. The requirement of using Versioned Boxes makes implementing complex data structures difficult, and makes porting existing algorithms much harder.
In contract to providing a library level interface, Disciplined Runtime choose to utilize the transactional capability of L0, which provides transparent, instruction-level transactional memory semantics. This provides a unified, flexible programming interface which is transparent to the program. The “native” support for transaction semantics makes existing algorithms written in C easier to be ported to C0, and allows programmers define complex data structures in a familiar way using variables and pointers. It also eliminate library-level overhead which enables the system to have high performance and good scalability.
CHAPTER 6
DISCUSSION AND CONCLUSION

This chapter discusses some limitations of current work and possible future improvement, and summarizes the contributions of disciplined memory access principle and our prototype system.

6.1 Discussion and Future Work

Although our prototype system shows very significant improvement in performance, scalability and generality comparing to previous solutions, there are still many aspects we can improve. These limitations, however, does not hurt the contribution of the concept of disciplined memory access and our prototype system.

In the current implementation, all the accessible memory ranges of a task memory space must be specified by the programmer. However, using program analysis techniques, we can automatically calculate some memory ranges. Although neither static nor dynamic program analysis can determine the memory ranges more precisely than expert programmers, it can provide a more comfortable way of programming and a useful default settings for beginner programmers.

The version of L0 we are using focuses on parallel multitasking, and the I/O capabilities are limited. The standard input and output via files are supported. A special memory address is mapped to the standard input/output, and the actual content of standard input/output is from/to the disk files specified by the command line parameters of the execution engine. However, this can be deeply improved. Actually, the latest version of L0 supports persistent memory which maps the distributed virtual address into disk files. The cc0 compiler can be easily extended to use this feature by adding a new rule in symbol allocating, similar to dealing with standalone variables.

We can also extend the existing L0 system to better utilize the memory access information. For example, the scheduling algorithms can be tuned according to sophisticated scheduling techniques. We can also implement the L0 runner compartment under other hardware and/or operating systems, and schedule different types of tasks into where the
platform are good at running it. There are many other aspects we can improve, hence the performance could be better in our future versions.

6.2 Conclusion

The disciplined memory access can make the system perform better by utilizing the memory access information, improve the performance and scalability of distributed applications without limiting the expressiveness of programming. Multitasking with disciplined memory access principle is efficient and scalable. The disciplined memory access is a suitable principle for designing large-scale applications in datacenters. Our Disciplined Runtime built on the principle of disciplined memory access can support efficient multitasking of applications from different problem domains, which means the disciplined memory access has the generality in terms of application logic.
REFERENCES


APPENDIX A

GRAMMAR OF C0 LANGUAGE

translation_unit
   : (function_definition | declaration)*
   ;

function_definition
   : declaration_specifiers declarator compound_statement
   ;

declaration
   : 'typedef' declaration_specifiers? init_declarator_list ';
   | declaration_specifiers init_declarator_list ';
   ;

declaration_specifiers
   : (storage_classSpecifier | typeSpecifier | type_qualifier)+
   ;

init_declarator_list
   : init_declarator+
   ;

init_declarator
   : declarator '=" initializer"
   | declarator
   ;

storage_classSpecifier returns [Type::Specifier retval]
   : 'extern'
   | 'static'
   | 'auto'
   | 'register'
   | 'standalone'
   ;

typeSpecifier returns [Type retv]
   : 'void'
   | 'bool'
   | 'char'
   | 'float'
   | 'double'
   | 'signed' 'short'
   | 'unsigned' 'short'
   | 'signed' 'int'
   ;
| 'unsigned' 'int' |
| 'signed' 'long' |
| 'unsigned' 'long' |
| 'short' |
| 'int' |
| 'long' |
| struct_or_unionSpecifier |
| enumSpecifier |
| typeId |

```plaintext
typeId :
IDENTIFIER |

struct_or_unionSpecifier :
( 'struct' | 'union' ) IDENTIFIER? '{' structDeclaration+ '}' |

structDeclaration :
specifierQualifierList structDeclaratorList ';' |

specifierQualifierList :
(typeSpecifier | typeQualifier)+ |

structDeclaratorList :
structDeclarator (',' structDeclarator)* |

structDeclarator :
declarator |
declarator ':' constantExpression |

denumSpecifier :
'enum' '{' enumeratorList '}' |
'enum' IDENTIFIER '{' enumeratorList '}' |
'enum' IDENTIFIER |

denumeratorList :
enumerator (',' enumerator)* |

denumerator :
IDENTIFIER |
IDENTIFIER '=' constantExpression |
```

46
type_qualifier
   : 'const' | 'volatile'
   :

declarator
   : pointer direct_declarator
   | direct_declarator
   | pointer
   :

direct_declarator
   : (IDENTIFIER | '(' declarator ')') declarator_suffix*
   :

declarator_suffix
   : '[' constant_expression ']
   | '[' ']
   | '([' parameter_type_list ')]
   | '([' ']
   // Function without parameters
   :

pointer
   : type_qualifier '*' pointer
   : type_qualifier '*'
   :

parameter_type_list
   : parameter_list
   | parameter_list ',' '...'
   :

parameter_list
   : parameter_declaration (',', parameter_declaration)*
   :

parameter_declaration
   : declaration_specifiers (declarator | abstract_declarator)
   :

identifier_list
   : IDENTIFIER (',', IDENTIFIER)*
   :

type_name
   : specifier_qualifier_list abstract_declarator
   | specifier_qualifier_list
   :

abstract_declarator
   : pointer direct_abstract_declarator
   :

47
| pointer |
| direct_abstract_declarator |
|

direct_abstract_declarator
: ( '(' abstract_declarator ')' | abstract_declarator_suffix )
    abstract_declarator_suffix*
;

abstract_declarator_suffix returns [DeclarationTemplate ::
    DeclarationSpecifier retval]
: '[' ']' |
    '[' constant_expression ']' |
    '(' ')' |
    '(' parameter_type_list ')' |
;

initializer
: assignment_expression |
    '{' initializer_list '}' |
;

initializer_list
: initializer ( ',', initializer )* |
;

argument_expression_list
: assignment_expression ( ',', assignment_expression )* |
;

additive_expression
: multiplicative_expression ( '(' '+' '|' '-' ) multiplicative_expression ) *
;

multiplicative_expression
: cast_expression ( '(' '*' '|' '/' '|' '%' ) cast_expression ) |
;

cast_expression
: '(' type_name ')' cast_expression |
    unary_expression |
;

unary_expression
: postfix_expression |
    '++' unary_expression |
    '--' unary_expression |
    unary_operator_cast_expression |
    'sizeof' unary_expression |
    'sizeof' '(' type_name ')' |

48
postfix_expression :
  primary_expression
  ( ']' expression ']' |
   '[' expression ',' ',',' expression ']' |
   '(': argument_expression_list ')' |
   '.' IDENTIFIER |
   '-' IDENTIFIER |
   '->' IDENTIFIER |
  ) |
  ;

unary_operator :
  '&' | '*' | '+' | '-' | '-' | '!' |
  ;

primary_expression :
  IDENTIFIER |
  constant |
  '(' expression ')' |
  ;

constant :
  HEX_LITERAL |
  OCTAL_LITERAL |
  DECIMAL_LITERAL |
  CHARACTER_LITERAL |
  STRING_LITERAL |
  FLOATING_POINT_LITERAL |
  'true' |
  'false' |
  ;

expression :
  assignment_expression ( ',' assignment_expression )* |
  ;

constant_expression :
  conditional_expression |
  ;

assignment_expression :
  lvalue assignment_operator assignment_expression |
  conditional_expression |
  ;

lvalue :
  unary_expression |
  ;
assignment_operator
  : \begin{small}
  & '=' \mid '*=' \mid '/=' \mid '%=' \mid '+=' \mid '-=' \mid '<=' \mid '>=' \mid '\&=' \mid '\^=' \mid '|=' \mid '; \\
  \end{small}

conditional_expression
  : \begin{small}
  & \text{logical_or_expression} '?' \text{expression} ':' \text{conditional_expression} \\
  & \text{logical_or_expression} \mid \text{logical_or_expression} \\
  \end{small}

logical_or_expression
  : \begin{small}
  & \text{logical_and_expression} ('||' \text{logical_and_expression})* \\
  \end{small}

logical_and_expression
  : \begin{small}
  & \text{inclusive_or_expression} ('&&' \text{inclusive_or_expression})* \\
  \end{small}

inclusive_or_expression
  : \begin{small}
  & \text{exclusive_or_expression} ('|' \text{exclusive_or_expression})* \\
  \end{small}

exclusive_or_expression
  : \begin{small}
  & \text{and_expression} ('\^' \text{and_expression})* \\
  \end{small}

and_expression
  : \begin{small}
  & \text{equality_expression} ('\&' \text{equality_expression})* \\
  \end{small}

equality_expression
  : \begin{small}
  & \text{relational_expression} ('=='|'!=') \text{relational_expression} \\
  \end{small}

relational_expression
  : \begin{small}
  & \text{shift_expression} ('<'|'<='|'>'|'>=' \text{shift_expression})* \\
  \end{small}

shift_expression
  : \begin{small}
  & \text{additive_expression} ('<<'|'>>') \text{additive_expression} \\
  \end{small}

statement
  : \begin{small}
  & \text{labeled_statement} \\
  & \text{compound_statement} \\
  & \text{expression_statement} \\
  & \text{selection_statement} \\
  & \text{iteration_statement} \\
  & \text{jump_statement} \\
  & \text{runner_statement} \\
  \end{small}
| inline_asm_statement |
| empty_statement |
| ; |
empty_statement : ';' |
| ; |
inline_asm_statement : "asm" '(' STRING_LITERAL ')' ';' |
| ; |
runner_statement : 'runner' primary_expression '(' argument_expression_list? ')' |
| ('using' argument_expression_list)? |
| ('watching' argument_expression_list)? |
| ';' |
| ; |
labeled_statement : IDENTIFIER ':' statement |
| ; |
compound_statement returns [BlockExpression *retval] : '{' declaration* statement_list '}' |
| ; |
statement_list : statement* |
| ; |
expression_statement : empty_statement |
| | expression ';' |
| ; |
selection_statement : 'if' '(' expression ')' statement ('else' statement)? |
| | 'switch' '(' expression ')' switch_body |
| ; |
switch_body : '{' |
| ('case' expression ':' statement_list |
| | 'default' ':' statement_list |
| )* |
| '}' |
| ; |
iteration_statement
  : 'while' '(' expression ')' statement
  | 'do' statement 'while' '(' expression ')' ';' |
  | 'for' '(' expression_statement expression statement expression ')' statement
  ;

jump_statement
  : 'goto' IDENTIFIER ';' |
  | 'continue' ';' |
  | 'break' ';' |
  | 'return' expression? ';' |
  | 'commit' ';' |
  | 'abort' ';' |
  | 'commitd' ';' |
  | 'abortd' ';' |
  ;

IDENTIFIER
  : LETTER (LETTER| '0'..'9')* |
  ;

LETTER
  : '$' |
  | 'A'..'Z' |
  | 'a'..'z' |
  | '_' ;

CHARACTER_LITERAL
  : '\' ( EscapeSequence | ~( '\\'| '\\') ) '\'' |
  ;

STRING_LITERAL
  : '"' STRING_GUTS '"' |
  ;

STRING_GUTS
  : ( EscapeSequence | ~( '\\'| '"') )* |
  ;

HEX_LITERAL
  : '0' ('x'| 'X') HexDigit+ IntegerTypeSuffix? |
  ;

DECIMAL_LITERAL
  : ( '0' | '1'..'9' '0'..'9'+ ) IntegerTypeSuffix? |
  ;

OCTAL_LITERAL
  : '0' ('0'..'7')+ IntegerTypeSuffix?
HexDigit
  : ( '0'..'9' | 'a'..'f' | 'A'..'F' )

IntegerTypeSuffix
  : ( 'l' | 'L' ) |
  | ( 'u' | 'U' ) ( 'l' | 'L' )?

FLOATING_POINT_LITERAL
  : ('0'..'9')+ '.' ('0'..'9')* Exponent? FloatTypeSuffix?
  | '.' ('0'..'9')+ Exponent? FloatTypeSuffix?
  | ('0'..'9')+ ( Exponent FloatTypeSuffix? | FloatTypeSuffix )

Exponent
  : ( 'e' | 'E' ) ( '+' | '-' )? ('0'..'9')+ 

FloatTypeSuffix
  : ( 'f' | 'F' | 'd' | 'D' )

EscapeSequence
  : '\\' ( 'b' | 't' | 'n' | 'f' | 'r' | '\"' | '\\' | '\\' )
  | OctalEscape

OctalEscape
  : '\\' ( '0'..'3' ) ( '0'..'7' ) ( '0'..'7' )
  | '\\' ( '0'..'7' ) ( '0'..'7' )
  | '\\' ( '0'..'7' )

UnicodeEscape
  : '\\u' HexDigit HexDigit HexDigit HexDigit

WS
  : ( ' ' | '\r' | '\t' | '\u000C' | '\n' )

COMMENT
  : '/\*\* .\* /\*

LINECOMMENT
  : '/\* ( '\n' | '\r' )* \r' ? '\n'