

Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnerarchitektur und Rechnerorganisation

**BMDFM: A Hybrid Dataflow Runtime Parallelization
Environment for Shared Memory Multiprocessors**

Thesis in Computer Engineering
by Oleksandr Pochayevets

A thesis submitted in fulfillment
of the requirements for the degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

Referees:

1. Prof. Dr. Arndt Bode (Technical University Munich)
2. Prof. Dr. Michael Gerndt (Technical University Munich)
3. Prof. Dr. Herbert Eichele (Georg-Simon-Ohm University of Applied Sciences Nuremberg)
4. Prof. Dr. Fridolin Hofmann (Friedrich-Alexander-University Erlangen-Nuremberg)

© 2004 Oleksandr Pochayevets

Abstract

Nowadays parallel shared memory symmetric multiprocessors (SMP) are complex machines, where a large number of architectural aspects have to be simultaneously addressed in order to achieve high performance. The quick evolution of parallel machines has been followed by the evolution of parallel execution environments. An effective parallel environment must be high-level enough so that it is easy for the programmer to use, and map well to the underlying computer architecture for efficient execution.

A recent general methodology of sequential code parallelization for SMP relies on compile-time methods. But a compiler can apply them only when the dependencies are simple and clear. However if dependencies are complex, compilers may not be able to suggest a different parallel execution order. Compile-time optimizations cannot be applied to situations where the time it takes to complete an operation varies at runtime, so the programmers have to synchronize the units of parallelism in their application programs explicitly. Compilers also can perform only limited inter-procedural and cross-conditional optimizations.

Directive-based parallelism for SMP supported by a fork-join paradigm runtime library has disadvantages due to the necessity of expressing parallelism explicitly, insufficient portability, granularity that is too fine and idling for the slowest slave threads.

On the other hand, the dataflow computational model presumes implicit exploitation of parallelism but lacks a natural software development methodology and has dynamic scheduling overhead.

These were the trends we were interested in, when we started the development.

To complement existing compiler-optimization methods we propose a programming model and a runtime system called BMDFM (Binary Modular DataFlow Machine), a novel hybrid parallel environment for SMP, that creates a data-dependence graph and exploits parallelism of user application programs at run time. This thesis describes the design and provides a detailed analysis of BMDFM, which uses a dataflow runtime engine instead of a plain fork-join runtime library, thus providing transparent dataflow semantics on the top virtual machine level.

Our hybrid approach eliminates disadvantages of the compile-time methods, the directive-based paradigm and the dataflow computational model. It is portable and is already implemented on a set of available shared memory symmetric multiprocessors. The transparent dataflow semantics paradigm does not require parallelization and synchronization directives. The BMDFM runtime system shields the end-users from these details. Tunable grain of parallelism provides efficient performance.

BMDFM is ported and evaluated on most available SMP platforms. The evaluation of BMDFM in this thesis was done on a POWER4 IBM p690 SMP machine. The evaluation consisted of several different types of experiments. We evaluated the overhead introduced by the execution environment, and the performance obtained running both standard numerical applications and non-trivial adaptive algorithm based applications.

Table of Contents

<generated_stuff_here>

Acknowledgements

Chapter 1 Introduction

- 1.1. Motivation
- 1.2. Contributions
- 1.3. Thesis Structure

Chapter 2 State of the Art

- 2.1. Overview
- 2.2. Target SMP Hardware
- 2.3. Directive-Based Parallelization and Fork-Join Paradigm for SMP
- 2.4. Dataflow Computation Model
- 2.5. Thread-Level Speculations
- 2.6. Software Dynamic Parallelization
- 2.7. Summary

Chapter 3 BMDFM Architectural Overview

- 3.1. Overview
- 3.2. Basic Concept
- 3.3. Multithreaded Architecture
- 3.4. Static Scheduler
- 3.5. Dynamic Scheduler
- 3.6. Programming Model
- 3.7. Virtual Machine Language and C Interface
- 3.8. Workflow for Applications
- 3.9. Running Applications on a Single Threaded Engine
- 3.10. Running Applications Multithreadedly
- 3.11. Summary

Chapter 4 Dynamic Scheduling Subsystem

- 4.1. Overview
- 4.2. Inter Process Synchronization
- 4.3. Shared Memory Pool
- 4.4. Non-Dead-Locking Policy
- 4.5. Inter Process Communication
- 4.6. Task Connection Zone
- 4.7. I/O Ring Buffer Ports
- 4.8. Data Buffer
- 4.9. Operation Queue
- 4.10. IORBP Scheduling Process
- 4.11. OQ Scheduling Process
- 4.12. CPU Executing/Scheduling Process
- 4.13. Complexity of the Dynamic Scheduling Subsystem
- 4.14. Summary

Chapter 5
Static Scheduling Subsystem

- 5.1. Overview**
- 5.2. Parallel Dataflow Code Style Restrictions**
- 5.3. Static and Dynamic Type Casting**
- 5.4. Code Reorganization**
- 5.5. Generation of Marshaled Clusters**
- 5.6. Uploading of Marshaled Clusters**
- 5.7. Summary**

Chapter 6
Transparent Dataflow Semantics

- 6.1. Overview**
- 6.2. Conventional Programming Paradigm**
- 6.3. Synchronization of Asynchronous Coarse-Grain and Fine-Grain Functions**
- 6.4. Ordering Non-Standard Stream in Out-of-Order Processing**
- 6.5. Speculative Dataflow Processing**
- 6.6. Summary**

Chapter 7
Evaluation

- 7.1. Overview**
- 7.2. Test Environment**
- 7.3. NAS Parallel Benchmarks**
- 7.4. Irregular Test**
- 7.5. Summary**

Chapter 8
Conclusion

- 8.1. Overall Summary**
- 8.2. Future Directions**

List of Defined Terms

References

List of Figures

<generated_stuff_here>

List of Tables

<generated_stuff_here>

Acknowledgements

Many people have made this work possible. I would like to express my deep gratitude to all of them for being there, working with me and helping me.

First of all, I would like to thank my doctoral advisers, Prof. Dr. Herbert Eichele (Georg-Simon-Ohm University of Applied Sciences Nuremberg), Prof. Dr. Arndt Bode (Technical University Munich), Prof. Dr. Michael Gerndt (Technical University Munich) and Prof. Dr. Fridolin Hofmann (Friedrich-Alexander-University Erlangen-Nuremberg) for their encouragement, patience, guidance and valuable support throughout the preparation of this thesis.

I wish to thank the staff of RZ TU Dresden and LRZ Munich for providing the facility and an excellent research environment to conduct my research.

Furthermore, I would like to thank my colleagues. They spent many hours of their spare time to read my thesis and provide numerous comments and ideas on how to improve this thesis.

Finally, but not less important, I would like to thank my family and friends for their support throughout the entire process.

Chapter 1

Introduction

1.1. Motivation

1.2. Contributions

1.3. Thesis Structure

1.1. Motivation

A recent general methodology of sequential code parallelization for SMP relies on compile-time methods. However, a compiler can apply them only when the dependencies are simple and clear, but if dependencies are complex, compilers may not be able to suggest a different parallel execution order.

Compilers cannot apply many interesting optimizations that depend on the knowledge of dynamic information. Compile-time optimizations cannot be applied to situations where the time it takes to complete an operation varies at runtime, which is a common case on cache-based and parallel computers. A user has to explicitly state the interaction and synchronization between the units of parallelism.

In general, the weaknesses with compile-time strategies can be described as follows:

- First, when running in parallel there are many operations that take a non-deterministic amount of time, making it difficult to know exactly when certain pieces of data will become available. In contrast, a runtime strategy allows a degree of adaptivity to tolerate not just large latencies, but varying or indeterminate latencies.
- Second, in a multi-user mode other people's codes can use up resources or slow down a part of the computation in a way that the compiler cannot account for. For example, there can be a severe degradation of performance on multi-programmed environments due to barrier synchronization.
- Finally, compilers can perform only limited inter-procedural and cross-conditional optimizations because they often cannot determine which way a conditional will go or cannot optimize across a function call.

To complement existing compiler-optimization methods we propose a programming model and a runtime system called BMDFM (Binary Modular DataFlow Machine) that creates a data-dependence graph and exploits parallelism of a user application program at run time.

Another important type of overhead is the amount of time required for the programmers to parallelize and synchronize the units of parallelism in their application programs. The BMDFM runtime system shields the end-users from these details, allowing them to make more efficient use of their expertise.

1.2. Contributions

The main contribution of this thesis is the design and implementation of the BMDFM runtime system - a complete parallelization environment, resulting in an efficient product, competitive with widely-used parallel execution environments. The well-combined SMP and dataflow paradigms are the basis of the proposed architecture.

The highlights are summarized below:

- Definition and design of the BMDFM hybrid architecture that combines von Neumann and dataflow computational models running on top of commodity SMP systems. In the proposed architecture the dataflow paradigm is used for the MIMD runtime parallel engine, which is controlled by the virtual machine built in von Neumann manner. Analysis of this design has shown that our approach is efficient and applicable both in the area of numerical high performance computations and for dynamic adaptive algorithms as well, hiding parallelization and synchronization details from the end-users.
- Definition and design of the context dependent data structure in the shared memory pool. This allows dataflow processing of the iterations in parallel storing the iteration's data dynamically under unique

contexts. The proposed data structure uses SVR4 IPC blocking semaphores distributed along the shared memory pool that is efficient and portable across the commodity SMP.

- Definition and design of the speculative tagging dynamic scheduling algorithm that is used to tag ready instructions for execution in the runtime dataflow engine. This is a solution how to significantly reduce dynamic scheduling overhead.
- Definition and design of the multithreaded marshaled clustering of the data loaded from the control virtual machine into the dataflow runtime engine. This approach allows to avoid a bottleneck when the parallel dataflow machine is fed dynamically from the single threaded control virtual machine. In the proposed scheme the marshaled clusters are prepared statically during the compilation stage, which does not cause additional runtime overhead for marshaling.
- Definition and design of transparent dataflow semantics at the top virtual machine level that shields the end-users from the parallelization and synchronization details. No special parallel directives are required. The transparent dataflow semantics level can be used both for conventional manual programming and as the target level for the code generators/translators.

1.3. Thesis Structure

This thesis is organized into 8 chapters. Chapter 2 analyzes existing SMP hardware and various parallelization approaches in the field of parallel SMP computing. We give an overview and compare methods of directive-based parallelization and fork-join paradigm. We also analyze the related works on dataflow computation models and some alternative projects of runtime SMP parallelization such as hardware Thread Level Speculation (TLS) and some software dynamic parallel schemes.

Chapter 3 discusses the architecture of the BMDFM system, functionality of its units, programming model and applicability.

Chapter 4 describes the dataflow runtime engine in details. We analyze the internal data structures, how they are mapped into the shared memory pool and dynamic scheduling algorithms.

Chapter 5 presents the static part of the BMDFM system. Therefore, we present the algorithms running during the compilation phase and the architecture of the multithreaded marshaled clustering.

Chapter 6 gives an overview of the transparent dataflow semantics paradigm introduced in this thesis. We show that a conventional programming approach can be automatically mapped into the proposed dataflow hybrid architecture.

After discussing all the proposals of this thesis, Chapter 7 presents an evaluation of the complete BMDFM parallelization environment running on a POWER4 IBM p690 SMP machine. We evaluate the overhead introduced by the execution environment and the performance obtained running both standard numerical applications and non-trivial adaptive algorithm based applications.

Finally, Chapter 8 contains the overall conclusions of this thesis and work planned for the future.

Chapter 2

State of the Art

2.1. Overview

2.2. Target SMP Hardware

2.3. Directive-Based Parallelization and Fork-Join Paradigm for SMP

2.4. Dataflow Computation Model

2.5. Thread-Level Speculations

2.6. Software Dynamic Parallelization

2.7. Summary

2.1. Overview

This chapter provides an analysis of recent parallelization technologies for SMP. The following important items are considered:

- We analyze the typical SMP architectures existing on the market, paying attention to CPU interconnections and memory latencies. We choose the architecture that is most appropriate for our tests.
- We provide a detailed description of directive-based parallelization and fork-join paradigm for SMP as the main technology in this area. We discuss various approaches and highlight disadvantages.
- We look at related work and alternative technologies for efficient parallelization. We think that the dataflow computation model, thread-level speculation and software dynamic parallelization are closest to our work. We analyze all of them to avoid the weak sides they may have.

2.2. Target SMP Hardware

Parallel processing increases the computational power of computers ranging from low-end workstations and even personal computers to big mainframes containing a large number of processing elements. Small shared-memory multiprocessor (SMP) machines are available from a great number of computer makers: Silicon Graphics, SUN Microsystems, Compaq/DEC, Hewlett Packard, Intel, Sequent, Data General, etc. Some of them also build big mainframes.

The larger the machines, the more complex they are. Complexity comes partially from the fact that the path from the processors to the memory becomes a bottleneck when more than 10-12 processors are put together. Poor scalability due to the memory subsystem bottleneck causes higher memory access latencies and poor performance when the number of processors is increased. Several solutions have been adopted to solve the problem of scalability. They include an improved path from the processors to the memory, through pipelined memory architectures in bus-based and NUMA computers. As a result, memory accesses have different latencies. The resulting architecture is more complex and difficult to manage to achieve high performance. Along with varying data access latencies, other elements that make parallel processing difficult in current hardware systems are the architecture of the current super-scalar processors, improved synchronization mechanisms and support for relaxed memory models.

Examples of SMP nodes are the Compaq/DEC AlphaServer GS140 [50, 51, 52, 53, 54, 55, 56], the SUN Ultra Enterprise 10000 server (Starfire) [33, 138, 139, 140], SGI Origin2000 ccNUMA server [38, 39, 99, 104] and the POWER4 IBM p690 SMP server [82, 105, 119]. These machines are commented on next.

Compaq/DEC AlphaServer GS140

The Compaq/DEC AlphaServer GS140 is based on the Alpha 21264 processor. It supports up to 14 processors. Each Alpha processor has up to 4 MB of external third-level cache. Processors are connected through a system bus supporting a maximum of 28 GB of main memory.

Figure 2.1 shows the architecture of this system. The system bus supports the connection of a maximum of 9 system boards. This limitation seems to confirm that the small bus size required for performance is really

conditioning the design of the computer. There are three types of system boards: processor boards, memory boards and I/O boards. A processor board may contain up to two Alpha processors. A memory board can accommodate up to 4 GB of memory. This means that a 14 processor system is limited to 4 GB of memory, due to the maximum of 9 system boards connected to the system bus.

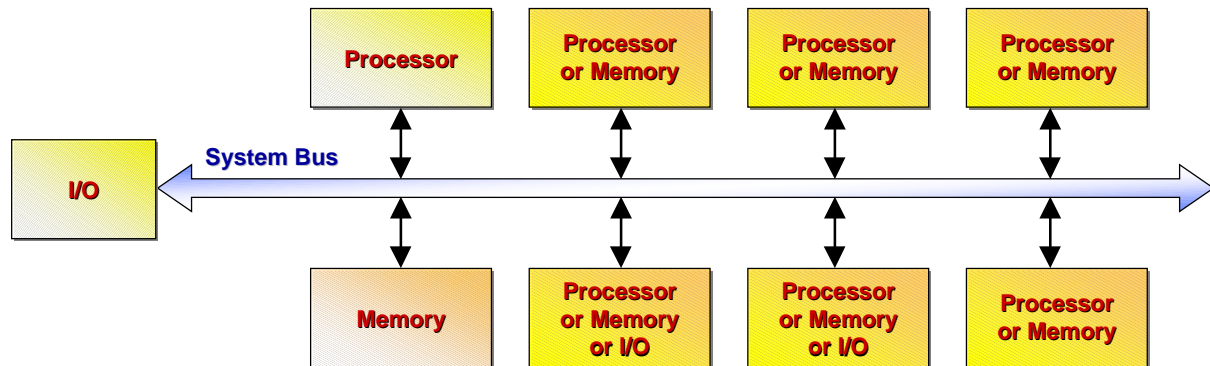


Figure 2.1. Compaq/DEC bus architecture

The SUN Ultra Enterprise 10000 (Starfire)

The SUN Ultra Enterprise 10000 server supports up to 64 processors and 64 GB of main memory. It is based on the Ultra SPARC processor and Gigaplane-XB interconnect technology. Each processor comes with 4 MB of external secondary cache.

The Enterprise 10000 server accommodates a maximum of 16 system boards. Each system board can be configured with up to 4 processors and 4 GB of memory. System boards are connected through the Gigaplane-XB interconnect, a crossbar designed specifically for this machine. It uses a packed switched scheme with separate address and data paths. The reason is that data is usually communicated point-to-point, while addresses have to be distributed simultaneously throughout the system for the snooping protocol. The main structure of the system is shown in Figure 2.2. Data transfers are done through a 16x16 crossbar allowing communication between any two system boards at the same time. Contention arises when a system board is the origin or the destination for two or more data transfers in the same bus cycle. In this case, only one of the requests can be satisfied and the rest must wait for an available bus cycle. Addresses are communicated to all boards through four independent address busses. Each bus covers 1/4 of the total address space.

With these characteristics, the data crossbar has a latency of 468 ns. The latency of a SUN Ultra 6000, a smaller machine supporting 24 processors is half (216 ns.) The problem is that, in the Enterprise 10000, both local and remote memory references suffer the latency penalty.

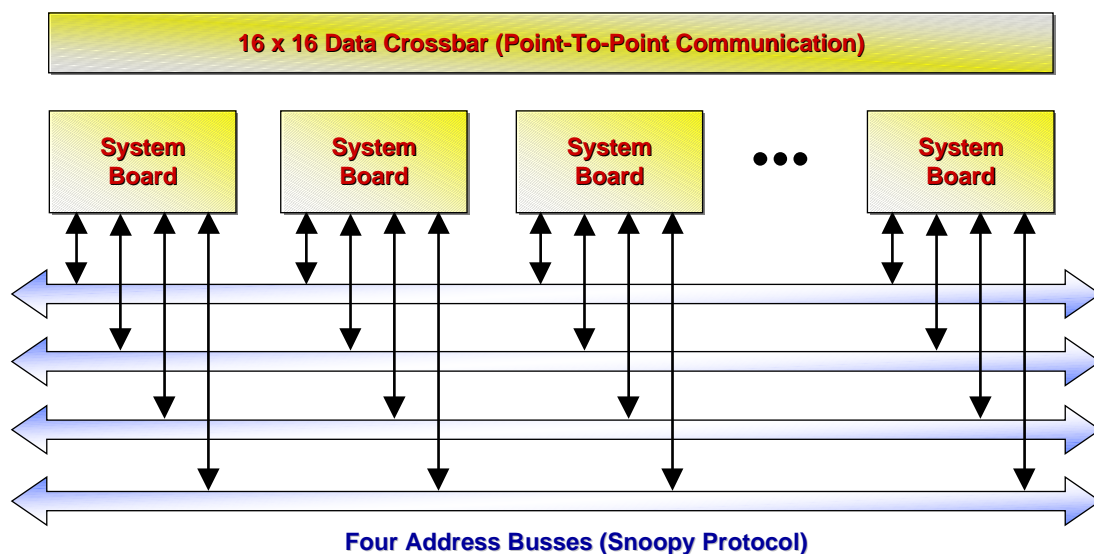


Figure 2.2. SUN Enterprise 10000 architecture

SGI Origin2000 ccNUMA server

The SGI Origin2000 server supports up to 1024 processors and one TB of main memory. It is based on the MIPS R10000 processor (with 4 MB of external secondary cache) and a distributed shared memory (DSM) architecture. The DSM architecture implements directory-based memory coherence, removing the broadcast bottleneck that prevents scalability in the snoopy bus-based SMP implementations.

The basic Origin2000 node is shown in Figure 2.3. Each node contains two R10000 processors, with their respective secondary cache memories. The central element in each node is the HUB, which connects both processors to the memory, I/O and the interconnection network (interconnection fabric, in SGI terminology). Each node can accommodate up to 64 GB of main memory and its corresponding directory memory. The global shared address space is distributed among the nodes in slices. Node 0 contains addresses in the lower range from 0 to N-1, node 1 follows, containing addresses from N to 2*N-1, and so on.

The DSM architecture provides global addressability from any processor to all memory and I/O. The Origin2000 system uses a directory-based memory coherence protocol. Each cache line in memory has an associated directory entry. Directory memory is located near main memory in the same module. Each entry contains information about the associated cache line: its system-wide caching state and bit-vectors pointing to caches, which have copies of the cache line. Memory can determine which caches need to be involved in a given memory operation in order to maintain coherence.

Processor nodes are attached to the interconnection routers, which provide low latency communication. Routers link the HUB inside the basic nodes to the CrayLink Interconnect. Each router has six external full-duplex connections, which are managed internally by a full six-way non-blocking crossbar switch. Machines with 128 processors use meta-routers (routers connecting routers) to connect four 32-processor groups. Meta-routers are replaced by 5-D hypercubes to reach up to 1024 processors. It is remarkable that, in a 64-processor machine, the average latency of a remote memory access is only 2.7 times the latency of a local memory access.

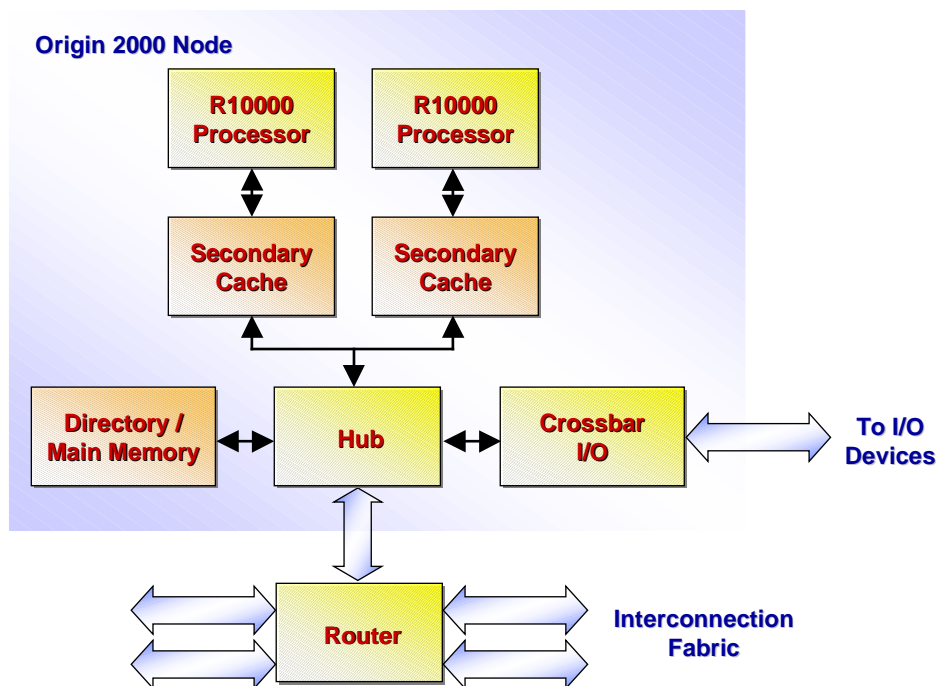


Figure 2.3. Origin2000 node and external connections

POWER4 IBM p690 SMP server

The pSeries 690 system architecture is implemented through Central Electronic Complex (CEC). Logically, the CEC consists of the microprocessors, pervasive functions and the storage subsystem. Physically, the CEC consists of the microprocessor chip, the Level 3 (L3) cache chip and the memory controller chip, which controls main memory.

At the heart of the CEC is the POWER4 chip [49], which contains: either one or two microprocessors; the L2 cache running at the same speed as the microprocessors; the microprocessor interface unit, which is the interface for each microprocessor to the rest of the system; the directory and cache controller for the L3 cache; the fabric bus controller, which is at the heart of the system's interconnection design; and a GX bus controller that enables I/O devices to connect to the CEC.

The second component of the POWER4 CEC is the L3 cache, comprised of two 16MB eDRAM chips mounted on a separate module. Each POWER4 chip controls an L3 cache, connected between the POWER4 chip and the memory controller chip. The third component of the POWER4 CEC is the memory controller chip. It is connected to the L3 cache on one side and to synchronous memory interface (SMI) chips on the other to control main memory. Each memory controller chip can have one or two memory data ports and can support up to 16GB of memory. There is a separate memory controller for each POWER4 chip. Two memory controllers are packaged on each memory card, and a maximum of two memory cards can be attached to each MCM. In all system configurations, all memory and all I/O is transparently accessible to all processors. The basic building block for pSeries 690 systems, the MCM, is shown in Figure 2.4. Each MCM contains four interconnected POWER4 chips, each with its own off-chip L3 cache. The pSeries 690 can contain up to 4 MCMs. Each MCM comprises either a 4-way or an 8-way symmetric multiprocessing (SMP) unit, depending on whether one or two microprocessors are present on each of the POWER4 chips.

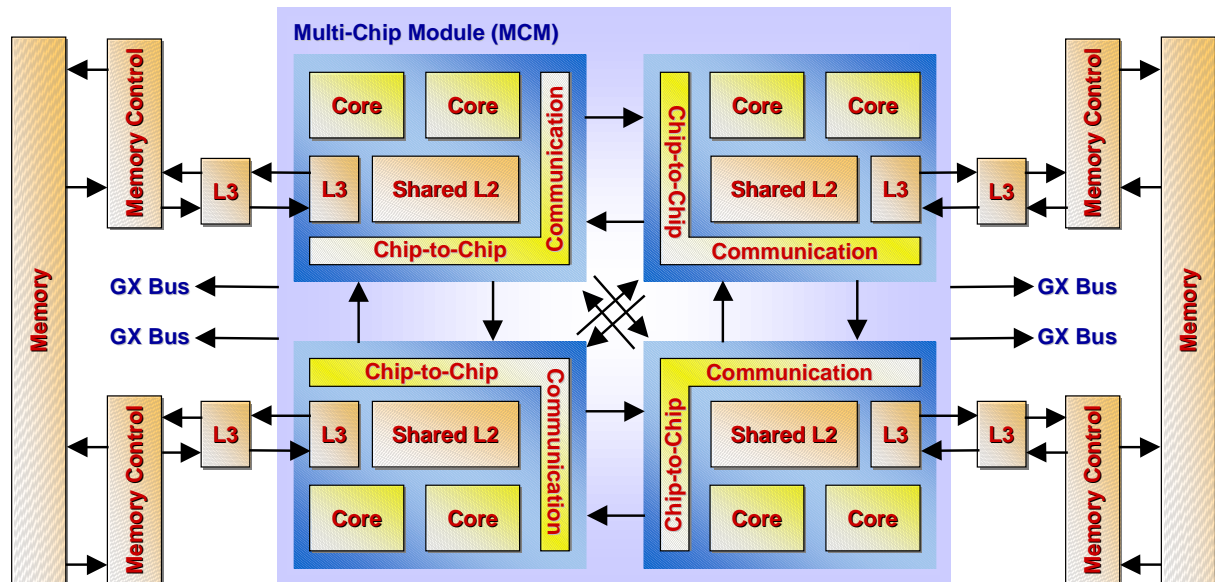


Figure 2.4. IBM POWER4 Multi-Chip Module (MCM)

The POWER4 microprocessor was specially designed to support an SMP memory hierarchy. As a result, IBM SMP architecture based on the POWER4 chip is the most tightly coupled SMP architecture nowadays.

At the beginning of year 2003, the US department of Energy (DOE) created a development program "Creating Science-Driven Computer Architecture: A New Path to Scientific Leadership" [107], which aims to restore American leadership in scientific computing and relies on IBM's POWER4, POWER5 and POWER6 evolving line.

In this thesis, we used the 8-way POWER4 IBM p690 SMP server for the evaluation of our proposals.

2.3. Directive-Based Parallelization and Fork-Join Paradigm for SMP

A recent general methodology of sequential code parallelization for Shared Memory Symmetric Multiprocessors (SMP) is the directive-based parallelization paradigm [5, 17, 20, 126, 127, 177]. A sequential program is first analyzed to discover loops, which are the main source of parallelism, and any dependencies among different loop iterations, which prevent parallelization of those loops for the sake of correctness. Based on this analysis it may be possible to modify the code to remove dependencies. Parallelism is expressed simply by inserting appropriate compiler directive before a loop. As Figure 2.5 indicates, this is essentially an iterative process of modifying the loop nesting in the sequential code until most of the computationally expensive loops

are parallelized. Finally, the parallelized code (i.e. sequential code with compiler directives) is compiled and linked with appropriate runtime libraries to be executed on the target system.

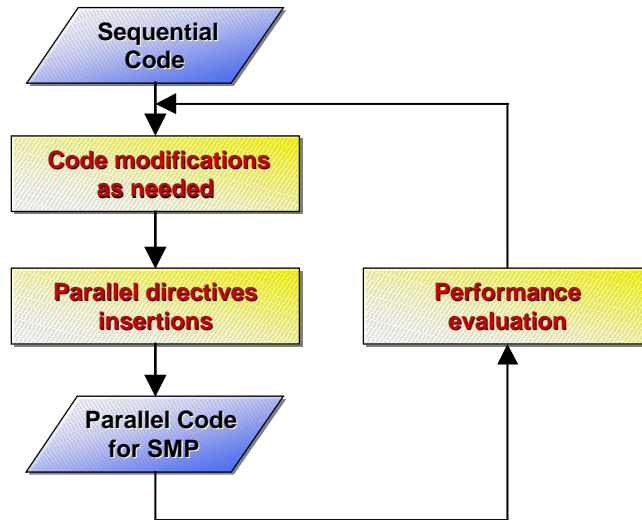


Figure 2.5. General methodology of sequential code parallelization for SMP

Directive-based parallelism is supported by a runtime library, which implements a fork-join paradigm of parallelism. A master thread initiates the program, creates multiple slave threads, schedules the iterations of parallelized loops on all threads including itself, waits for the completion of a parallel loop by all the slave threads and continues to execute the sequential parts of the program. Slave threads must wait for work (i.e. for parts of subsequent parallel loops) while the master thread is executing a sequential portion of the code.

Use of directive-based parallelism is limited due to portability issues. Almost every vendor of a shared memory computer system offers its own extension of Fortran or C language via parallelization directives. Usually these directives are not portable from one shared memory computer system to another. The OpenMP Architecture Review Board spent a lot of efforts to standardize a collection of compiler directives, library functions and environment variables that is known as the OpenMP Application Program Interface (API) [44, 117] and used to specify shared memory parallelism in programs. Although the OpenMP model can be useful for solving a variety of problems, it is somewhat tailored for large array-based applications. In OpenMP any unsynchronized calls to output functions may result in output, in which data written by different threads appears in non-deterministic order. Similarly, unsynchronized calls to input functions may read data in non-deterministic order. According to OpenMP the user explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel. OpenMP implementations do not check for dependencies, conflicts, deadlocks, race conditions or other problems which result in incorrect program execution. The user himself is responsible to ensure that the application using the OpenMP API constructs executes correctly.

A runtime library, which implements a fork-join paradigm, has the drawback of the idle time, in which program execution has to wait for the completion of the slowest thread. Even if a master thread initializes an equal chunk of iterations for each slave thread the execution time will differ due to the Non-Uniform Memory Access (NUMA) of modern SMP architectures. To avoid idle time wasting, a mechanism of guided scheduling of threads was introduced [130]. With guided scheduling the iterations are assigned to threads in chunks with approximately exponentially decreasing sizes. When a thread finishes its assigned chunk of iterations, it is dynamically assigned another chunk until no more remain. The drawback of this approach is the considerably bigger overhead spent on dynamic scheduling of the threads.

One more issue with directive-based parallelization is that multiprocessors present more difficult challenges to parallelizing compilers than the vector machines which initially were their targets. Effective use of the vector architecture involves parallelizing of repeated arithmetic operations on large data streams (e.g. innermost loops in array-oriented programs). On a multiprocessor however, parallelizing of innermost loops typically does not provide sufficient granularity of parallelism - not enough computational work is performed in parallel to overcome the overhead of synchronization and communication between processors. To utilize a multiprocessor effectively the compiler must exploit coarse grain parallelism locating large computations, which can execute independently in parallel [87]. Multiprocessor systems also have more complex memory hierarchies than typical vector machines. Modern multiprocessors also contain multiple levels of caches in addition to the shared

memory. These additional challenges often prevented early parallelizing compilers from being effective for multiprocessors.

Commercial parallel execution environments provide different user-level thread libraries. There are two main types of such libraries: The standard libraries, such as Pthreads (Posix threads) [83], are intended for parallel programmers to build parallel applications expressing the parallelism by hand. The programmer introduces explicit calls to the thread library to create, manage, terminate and synchronize the parallel application tasks. The Pthreads library is oriented to work with shared memory. There are also standard libraries oriented toward message passing, such as PVM [63] and MPI [108, 109]. There are also plenty of custom thread libraries (such as the SGI MP library). Custom thread libraries are highly tuned for execution on top of a parallel architecture. For instance, spawning parallelism in custom libraries is done from a master processor to all the slave processors at once, instead of supplying work on a one-to-one basis, which it is the case in the Pthreads library. Such a fine tuning encourages the use of very simple structures to support the parallelism. Simplicity leads to efficiency, but also there is a lack of functionality with respect to standard libraries. For instance, the SGI MP library forbids spawning parallelism inside a parallel region. That is allowed in the Pthreads library, which does not impose such a restriction by allowing that any pthread be able to spawn a new pthread. In general, it is common that custom libraries do not support the exploitation of multiple levels of parallelism.

A multiprocessor operating system assigns physical processors to application processes (or threads). Different kernel-level scheduling policies are usually provided by the operating system. For instance, the SGI IRIX operating system provides time-sharing and gang scheduling policies [156] to manage parallel applications. Time-sharing is a priority-based scheduling policy, where processes are scheduled independently of each other. In gang scheduling, instead, processes belonging to the same application are scheduled as a group. In the SGI MP execution environment, the time-sharing policy is used because it is more dynamic than gang scheduling and applications are able to adapt to the number of processors allocated. In fact, each application has a specific thread in charge of controlling the load of the system and whether the application is taking advantage of the allocated processors. This thread serves two purposes. First, in case that thread detects that the load is high and the application is not receiving enough resources, it decides to stop some of the processes of the application, to free some physical processors and reduce the system load. Second, when that thread detects that the load is low and the application can use more processors, it starts some of the processes of the application to take advantage of more physical processors. This is a specific feature of the SGI MP execution environment, not found in other environments.

Other operating systems, such as Digital UNIX, provide the FIFO and round-robin scheduling policies, in addition to time-sharing, to schedule processes/threads on top of processors. SunOS provided a gang scheduling class for lightweight processes (LWPs) [131]. Nevertheless, the parallel execution environments running on top of them are simpler than the SGI one, lacking the same kind of communication between the user and kernel levels.

Nowadays parallelization of sequential code for shared memory systems is an extensively researched area. A lot of research efforts have focused on parallelizing sequential programs. Currently, many types of loops can be parallelized with various data dependency analysis techniques [20, 127]. Among them GCD, Benerjee's inexact and exact tests [17, 177], OMEGA test [132], symbolic analysis [74], semantic analysis and dynamic dependence test and program restructuring techniques such as array privatization [170], loop distribution, loop fusion, strip mining and loop interchange [121, 178]. Due to the simplicity of shared memory multiprocessor programming, compiler developers have provided various facilities to allow the users to exploit parallelism. Native compilers support multiprocessing directives to allow users to exploit loop-level parallelism in their programs. Every commercial parallel execution environment provides a set of parallelizing compilers, usually supporting the C and Fortran languages. For instance, both the MIPS Pro C [150] and Fortran 77 [153] compilers provided by Silicon Graphics are able to automatically extract loop parallelism from sequential applications. This is done with the help of the PCA (Parallel C Analyzer) and PFA (Parallel Fortran Analyzer) tools [152], respectively. Both in automatic and annotated parallelizations, the resulting parallel code calls to the SGI MP library [151, 154, 155], the custom thread library provided by SGI to support parallel execution. KAP [96] and Polaris [24, 129] are other parallelizing compilers, which are able to generate parallelized code for SMPs. The Polaris compiler exploits loop parallelism by using inline expansion of subroutine, symbolic propagation, array privatization [57, 170] and runtime data dependence analysis [133]. CAPTools [101] is a semi-automatic parallelization tool, which transforms a sequential program to a message passing program by user directed distribution of arrays. The PROMIS compiler [27, 120] combines the Paraphrase2 compiler [125] using HTG [65] and symbolic analysis techniques [74], and the EVE compiler for fine grain parallel processing. However, these compilers cannot parallelize loops that include complex loop carrying dependences and conditional branches to the outside of a loop.

There are several well-known non-commercial projects in which parallelizing compilers are being developed to generate code to run on top of custom thread libraries. For instance, the SUIF compiler [4, 75] gets sequential Fortran code and automatically generates parallel code to run on a custom library, with no communication with the operating system. The SUIF compiler system incorporates various modules, which can be used to analyze the sequential program, to parallelize loops, distribute program arrays and perform inter-procedural analysis [4, 75, 76], unimodular transformation and data locality optimization [9, 98]. Effective optimization of data localization is more and more important because of the increasing disparity between memory and processor speeds. (Currently, many researchers for data locality optimization using program restructuring techniques such as blocking, tiling, padding and data localization are trying to achieve high performance computing with single chip multiprocessor systems [80, 136, 180].) The SUIF runtime system supports a single-level of parallelism in the same way as the SGI MP library does. The NANOS compiler [13, 104] based on Parafrase2 has been trying to exploit multi-level parallelism including the coarse-grain parallelism by using the extended OpenMP API.

In the field of runtime libraries, searching for support of multiple levels of parallelism and fine granularity is a main goal. The Illinois-Intel Multithreading Library (IML [64]) is a user-level threads package supporting nested parallelism and code generation from the Intel Fortran compiler. It runs on PC-compatible Intel multiprocessor machines, on top of the Windows NT kernel. Communication with the operating system includes a specific interface to get the number of available processors and to stop and resume kernel threads.

COOL [31] is a runtime system supporting parallel object-oriented programs written in the COOL language [32]. It is intended for NUMA machines. The programmer is allowed to express data locality in three different ways: through object, task and processor affinity. Cilk, Filaments, Concert and Active Threads are thread libraries also providing support for compiler generated code. Communication with the operating system is not considered in any of these projects. They provide support for multiple levels of parallelism. Data locality is taken into account, providing tools in the library interface to map the application tasks onto specific processors. The Cilk language extends C with parallel constructs. The Cilk runtime system [25] is oriented to express parallelism in recursive programs. Filaments [58] can be used directly by the C programmer or from the Sisal functional language. It supports fine-grained iterative and fork-join threads. Active Threads [176] is a runtime system supporting code generation for the pSather compiler [111, 162]. Threads are grouped in thread bundles, sharing a common scheduler. Bundles facilitate data locality because threads accessing the same data can be assigned to the same physical processors through the bundle.

Projects considering the improvement of communication with the kernel include Process Control, Scheduler Activations, First-Class User-level Threads and Execution Vehicles. Process Control [171, 172] introduces the concept of dynamic adaptation of the parallelism inside applications to the available resources, as indicated by the kernel. The application receives enough information to start and stop processes when needed to adapt to the allocated resources. Scheduler Activations [6] provides to the user level all scheduling events related to the application. The events are: receiving a new processor, a processor preemption, thread blocking and thread unblocking. All events are communicated through the upcall mechanism, which sometimes is too costly to allow an efficient communication path between user and kernel levels. First-Class User-Level Threads [103] merges the upcall mechanism with the shared memory. The most aggressive approach is taken in the recent implementation in IRIX6.5 of Execution Vehicles [40]. In this approach, the full context of the kernel-level threads is made available to the user-level execution environment, in such a way that both the kernel and the user levels can resume a preempted thread.

There are other research projects providing threads libraries, such as Quartz [8], FastThreads [7], Presto [22] and SwitchStacks [35] which are interesting for various reasons and have also been studied during the development of this work.

2.4. Dataflow Computation Model

Dataflow expresses computations as operations, which may in principle be of any size. The execution of these operations depends solely on their data dependencies - an operation is computed after all of its inputs have been computed, but this moment is determined only at runtime. Operations, which do not have a mutual data dependency, may be computed concurrently.

The fundamental principles of dataflow were developed by Jack Dennis [45] in the early 1970s. The dataflow model [2, 47, 67, 145] avoids the two features of von Neumann model, the program counter and the global updatable store, which become bottlenecks in exploiting parallelism [15]. Due to its elegance and simplicity, the pure dataflow model has been the subject of many research efforts. Since the early 1970s, a number of dataflow computer prototypes have been built and evaluated, and different designs and compiling techniques have been

simulated [10, 46, 62, 100, 114, 144, 149, 157, 159, 165, 167]. Depending on the way of handling the data, several types of dataflow architectures emerged in the past: single-token-per-arc dataflow [48], tagged-token dataflow [12, 175] and explicit token store [122]. The major advantage of the single-token-per-arc dataflow model is its simplified mechanism for detecting enabled nodes. Unfortunately, this model of dataflow has a number of serious drawbacks. Since consecutive iterations of a loop can only partially overlap in time, only a pipelining effect can be achieved and thus a limited amount of parallelism can be exploited. Another undesirable effect is that token traffic is doubled. There is also a lack of support for programming constructs essential to any modern programming language.

The performance of a dataflow machine significantly increases when loop iterations and subprogram invocations can proceed in parallel. To achieve this, each iteration or subprogram invocation should be able to execute as a separate instance of a reentrant subgraph. However, this replication is only conceptual. In an implementation only one copy of any dataflow graph is actually kept in memory. Each token includes a tag, consisting of the address of the instruction for which the particular data value is destined, and other information defining the computational context in which that value is used. A node is enabled as soon as tokens with identical tags are present at each of its input arcs. Dataflow architectures using this method are referred to as tagged-token (or dynamic) dataflow architectures. The most important tagged-token dataflow projects are Manchester Dataflow Computer [18, 19, 26, 29, 30, 66] and MIT Tagged-Token Dataflow Machine [11, 69, 70]. The major advantage of the tagged-token dataflow model is the higher performance it obtains by allowing multiple tokens on an arc. One of the main problems of tagged-token dataflow model is the efficient implementation of the unit that collects tokens. For reasons of performance, an associative memory would be ideal. Unfortunately, it would not be cost-effective since the amount of memory needed to store tokens waiting for a match tends to be very large. Therefore, all existing machines use some form of hashing techniques, which typically are not fast compared with associative memory. To eliminate the need for associative memory searches, the concept of an explicit address token store has been proposed. The basic idea is to allocate a separate memory frame for every active loop iteration and subprogram invocation. Each frame slot is used to hold an operand for a particular activity. The explicit token address store principle was developed in the Monsoon project [42, 123, 146, 166] but is applied in most dataflow architectures developed more recently, i.e. as so-called direct matching in the EM-4 [23, 94, 141, 142, 143] and Epsilon-2 [71, 72].

Pure dataflow computers usually perform quite poorly with sequential code. A further drawback is the overhead associated with token matching. One solution of these problems is combining the dataflow and control-flow mechanisms. The symbiosis between dataflow and von Neumann architectures is represented by a number of research projects developing von Neumann/dataflow hybrids [28, 84]. After early hybrid dataflow attempts, several techniques for combining control-flow and dataflow emerged [21]: threaded dataflow [124], large-grain dataflow [84], RISC dataflow [113] and dataflow with complex machine operations [59]. The threaded dataflow technique has a modified dataflow principle so that instructions of a sequential instruction stream can be processed in succeeding machine cycles. A thread of instructions is issued consecutively by the matching unit without matching further tokens except for the first instruction of the thread.

The large-grain dataflow technique, also referred to as coarse-grain dataflow, advocates activating macro dataflow actors by the dataflow principle while executing the represented sequences of von Neumann instructions. Off-the-shelf microprocessors can be used for the execution stage. Most of the more recent dataflow architectures fall into this category and they are often called multithreaded machines: Threaded Abstract Machine TAM [41, 43], Associative Dataflow Architecture ADARC [163, 164, 181], Pebbles architecture [137].

Another stimulus for dataflow/von Neumann hybrids was the development of RISC dataflow architectures, notably P-RISC [113], which allow the execution of existing software written for conventional processors. Using such a machine as a bridge between existing systems and new dataflow supercomputers made the transition from imperative von Neumann languages to dataflow languages easier for the programmer. Another technique to reduce the instruction level synchronization overhead is the use of complex machine instructions, for instance vector instructions. These instructions can be implemented by pipeline techniques as in vector computers. Structured data is referenced in blocks rather than element-wise, and can be supplied in a burst mode.

Recently the dataflow principles have also been used on the micro level. The latest generation of super-scalar microprocessors as Intel Pentium [37], MIPS [179] and PA-RISC [97] displays an out-of-order dynamic execution that is referred to as local dataflow or micro dataflow. In the first paper on the PentiumPro, the instruction pipeline is described as follows: "The flow of the Intel Architecture instructions is predicted and these instructions are decoded into micro-operations (micro-ops), or series of micro-ops, and these micro-ops are register-renamed, placed into an out-of-order speculative pool of pending operations, executed in dataflow order (when operands are ready), and retired to permanent machine state in source program order". But the micro dataflow of today's super-scalar processors also has some problems when finding enough fine-grain parallelism

to fully exploit the processor. One solution is to enlarge the instruction window to several hundred instruction slots with hopefully more simultaneously executable instructions present. There are two drawbacks to this approach. First, given the fact that all instructions stem from a single instruction stream and that on average every seventh instruction is a branch instruction, most of the instructions in the window will be speculatively assigned with a very deep speculation level. Thereby most of the instruction execution will be speculative. The principal problem here arises from the single instruction stream that feeds the instruction window. Second, if the instruction window is enlarged, the updating of the instruction states in the slots and matching of executable instructions lead to more complex hardware logic in the dispatch stage of the pipeline, thus limiting the cycle rate increase.

Ideally, a parallel programming environment should be able to exploit parallelism, which is not expressed explicitly. From this point of view the dataflow computational model has been attractive because it allows the exploitation of parallelism at instruction, loop, procedure and task levels implicitly. Communication between the operations is not made explicit in the dataflow programs, rather the occurrence of a name as the result of an operation is associated by the compiler with all of those places where the name is the input of an operation. Because operations execute only when all of their inputs are present, communication is always unsynchronized. Having problems with the single assignment and iterations, dataflow languages have taken different approaches expressing repetitive operations. Language such as Id [135] and Val [1] that most recently has evolved to Sisal [60] have syntactic structures looking like loops, which create a new context for each execution of the loop body. These languages seem like imperative languages except that each variable name may only be assigned once in each context. In Sisal parallelism is not explicit at the source level. However, the language runtime system may exploit parallelism. The loop bodies could be scheduled simultaneously and their results then collected.

2.5. Thread-Level Speculations

Techniques such as simultaneous multithreading [173] (e.g., the Alpha 21464) and single-chip multiprocessing [116] (e.g., the Sun MAJC [168] and the IBM Power4 [85]) suggest that thread-level parallelism may become increasingly important even within a single chip. Despite the significant progress which has been made in automatically parallelizing regular numeric applications, compilers have had little or no success in automatically parallelizing highly irregular numeric or especially non-numeric applications due to their complex control flow and memory access patterns. One promising technique for that is Thread-Level Speculation (TLS), which enables the compiler to create parallel threads optimistically despite uncertainty as to whether those threads are actually independent.

Loop speculation may be used to split virtually any sequential program with loops of any kind into arbitrary threads that can be parallelized across several processors. However, before the system can successfully exploit loops it must know where they occur in the code. It is actually possible to find loops as existing code executes, simply by looking for backward branches. Wherever one is encountered, the hardware may assume that a loop has been found and begin forking off speculative threads with the program counter set to the target of the backward branch in all speculative threads. However, finding loops “on the fly” like this with legacy code requires the hardware to speculatively track references to registers as well as memory. Because it is too expensive it was necessary to add a semi-manual compiler pass to isolate the loops. Figure 2.6 explains how a conventional loop has to be modified to run on a speculative hardware. The loop itself is replaced with the loop starting template that invokes the loop encapsulation function on the main processor after signaling the other processors to start it as well. The loop body is encapsulated within the new “ThisLoop()” encapsulation function. In the process, the necessary “spec_.” marking lines are added to provide the correct addresses to the “spec_begin()” API. The body clauses of the loop are distributed appropriately to use the speculation APIs. In order to ensure correct program execution, TLS hardware must track all inter-thread dependencies. When a true dependence violation is detected, the hardware must ensure that the “later” thread in the sequence executes with the proper data by dynamically discarding threads that have speculatively executed with the wrong data.

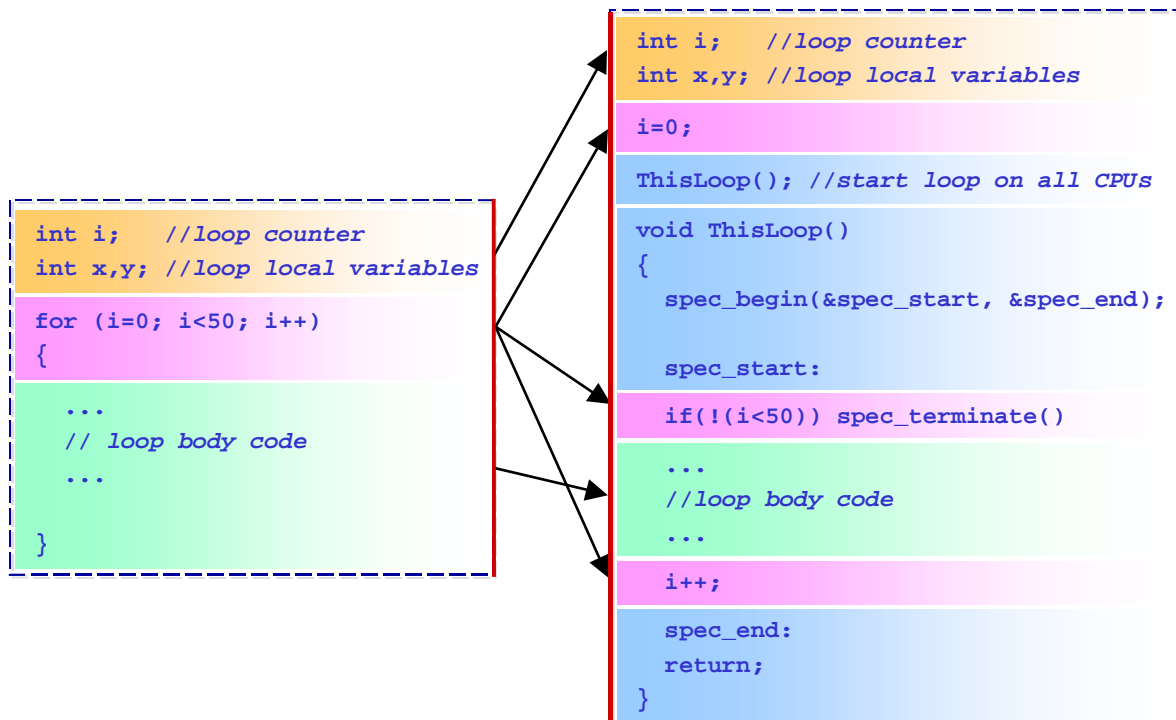


Figure 2.6. Speculation loop transformations

Knights was the first to propose hardware support for a form of thread-level speculation [93]. His work was within the context of functional languages. The Multi-scalar architecture [61, 151] was the first complete design and evaluation of the architecture for TLS. There have since been many other proposals, which extend the basic idea of thread-level speculation [3, 36, 68, 73, 95, 102, 118, 160, 169]. In nearly all of these cases, the target architecture has been a very tightly coupled machine, where all of the threads are executed within the same chip. These proposals have often exploited this tight coupling to help them track and preserve dependencies between threads. For example, the Stanford Hydra architecture [78, 79], which consists of four MIPS cores, uses special write buffers to hold speculative modifications, combined with a write-through coherence scheme that involves snooping of these write buffers upon every store. Hydra simulator tests show overall speedup of 2.5 on a variety of applications from different domains.

While TLS approach may be perfectly reasonable within a single chip, it was not designed to scale to larger systems. However, there are some attempts to scale the thread-level speculation on the traditional SMP architectures using cache coherency techniques [161]. Experimental simulation results demonstrate that TLS scheme offers absolute program speedups ranging from 8% to 46% on a four-processor system. There are also some other attempts [182] for a form of TLS within large-scale NUMA multiprocessors.

2.6. Software Dynamic Parallelization

In general, while compile-time methods have the advantage that they incur little runtime overhead and that the automation frees the user from the details of locality and parallelism, they are limited in that compilers cannot apply many interesting optimizations that depend on knowledge of dynamic information. Compile-time optimizations cannot be applied to situations where the time it takes to complete an operation varies at runtime, a common case on cache-based and parallel computers.

Nowadays only few software systems support dynamic parallelization (that actually are dataflow simulators). The OSCAR compiler has realized a multi-grain parallel processing [88] that effectively combines the coarse-grain task parallel processing with the loop parallelization and near fine-grain parallel processing [81, 90, 91, 92]. In the OSCAR compiler, a dynamic scheduling routine generated by the compiler is used to schedule coarse-grain tasks dynamically onto processors or processor clusters to cope with the runtime uncertainties caused by conditional branches. As the embedded dynamic task scheduler, the centralized dynamic scheduler [89, 115] in the OSCAR Fortran compiler and the distributed dynamic scheduler [110] have been proposed. Concert is a concurrent object-oriented language and runtime system [34] designed to support fine-grain applications, the

behavior of which is unknown at compile time. SMARTS (Shared Memory Asynchronous RunTime System) [174], based on the POOMA (Parallel Object-Oriented Methods and Applications) framework [86, 134], is a C++ class library for high performance scientific computations.

Software dynamic parallelization borrows the ideas from macro-dataflow computation models and builds the graph of data dependencies dynamically. Figure 2.7 demonstrates typical architecture of the runtime dependence-driven execution. The control thread unit provides the iterations to the dependent graph manager/scheduler. Ready iterations feed the work queues and are processed by the CPU units. The completed iterations resolve dependencies for the next iterations to become ready. Evaluations done in [174] show the total amount of time spent on system overhead was only 3.49% on numeric applications, indicating that runtime graph generation does not have significant overhead costs.

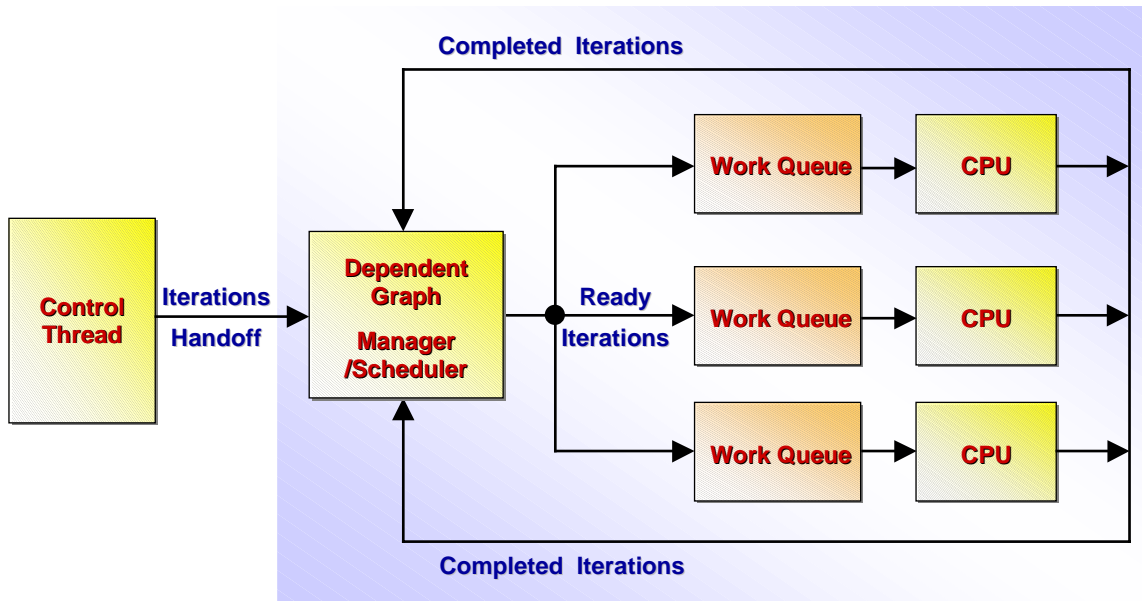


Figure 2.7. Runtime dependence-driven execution

Two important drawbacks can be highlighted for the existing dynamic architectures. First, they are still oriented only toward the numeric loop-based scientific applications and they have no support for irregular adaptive algorithms. Second, the control thread and dependent graph manager/scheduler are centralized in one thread, so they are a bottleneck for the system.

2.7. Summary

Having analyzed the recent parallelization technologies for SMP, we made the following important conclusions:

- We have chosen the POWER4 IBM p690 SMP architecture for the evaluation of our proposals as the most tightly coupled SMP architecture available today. IBM's POWER4, POWER5 and POWER6 evolving line is considered to be main stream in DOE development program for scientific computing.
- Directive-based approach has a lot of weaknesses because of its static nature. Directive-based implementations do not check for dependencies, conflicts, deadlocks, race conditions or other problems, which result in incorrect program execution. A runtime library, which implements a fork-join paradigm, has the drawback of the idle time, in which program execution has to wait for the completion of the slowest thread. Compile-time methods are limited in that compilers cannot apply many interesting optimizations that depend on knowledge of dynamic information.
- Dataflow computation model relies on runtime parallelization, but "pure" dataflow machines have overhead because each dynamic instruction requires dynamic operand matching, and because extra instructions are needed to make copies of data when a particular value is required by several instructions. Dataflow languages are abstract and simple, but they do not have a natural software development methodology and still contain an inconvenient single assignment paradigm.

- Thread-level speculation complements parallelization dynamically when a code is un-analyzable at the compilation stage, but it requires special hardware support; this hardware, however, will be occupied partly with unnecessary speculative processing. The thread-level speculation approach works within a single chip and it is not designed to scale to larger systems.
- Recent attempts at software dynamic parallelization, which are based on dataflow computation model, are still oriented only toward the numeric loop-based scientific applications and do not support the irregular adaptive algorithms. A master control thread is centralized, so it is a bottleneck for the system.

We think that dataflow systems are probably worth another look at this time. The community has gone through a shared-distributed-shared memory cycle since the peak of dataflow activity and applying some of what we have learned in that time to software based systems seems appropriate.

We build our architecture as a hybrid SMP dataflow runtime engine that avoids disadvantages of its predecessors. Our system combines both static and dynamic scheduling, does not require any special hardware, is portable and scalable on commodity SMPs and is applicable for numeric-based and irregular adaptive algorithms as well.

Chapter 3

BMDFM Architectural Overview

- 3.1. Overview
- 3.2. Basic Concept
- 3.3. Multithreaded Architecture
- 3.4. Static Scheduler
- 3.5. Dynamic Scheduler
- 3.6. Programming Model
- 3.7. Virtual Machine Language and C Interface
- 3.8. Workflow for Applications
- 3.9. Running Applications on a Single Threaded Engine
- 3.10. Running Applications Multithreadedly
- 3.11. Summary

3.1. Overview

This chapter introduces the BMDFM architecture. We discuss the following important basic issues:

- The idea of a hybrid architecture comprised of the dataflow and SMP paradigms.
- Architecture and configuration of a dataflow runtime engine.
- Programming model.
- Structure of byte code and the C interface.
- Running an application in single threaded and multithreaded modes.

3.2. Basic Concept

BMDFM (Binary Modular DataFlow Machine) [14, 128] is built as a hybrid of the SMP and dataflow paradigms. The basic concept is shown in Figure 3.1.

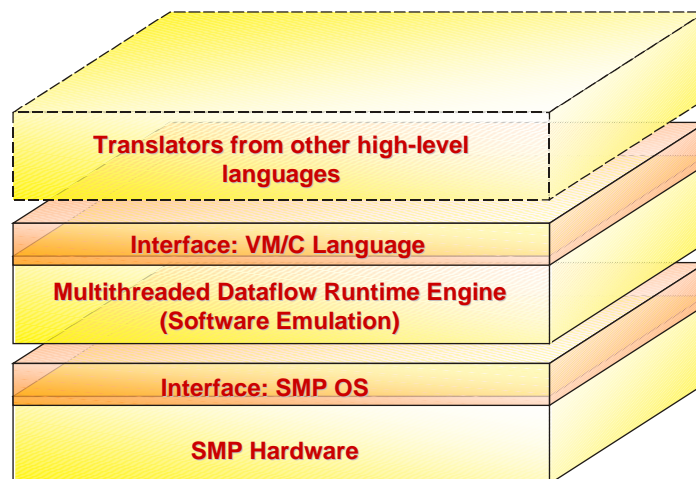


Figure 3.1. Basic concept of BMDFM

Our approach relies on underlying commodity SMP hardware, which is available on the market. Normally, SMP vendors provide their own SMP Operating System (OS) with an SVR4 UNIX interface on top (Linux, HP-UX, SunOS/Solaris, Tru64OSF1, IRIX, AIX, MacOS, etc.). To provide maximal portability we use only very basic UNIX functionality:

- standard I/O and terminal capabilities (termcap);
- parallelism of the processes, which are created once by the fork() system call during the startup of BMDFM;
- SVR4 IPC POSIX functionality: shmget()/shmctl() for initial creation of the shared memory pool and semget()/semctl() for the purpose of runtime synchronization;
- ANSI C compiler to compile the complete system, which is written strictly in ANSI C.

Having no conditional compilation directives, BMDFM is compiled for most SMP machines with their native SMP OS's and is publicly available [14] for download.

On top of an SMP OS we run our multithreaded dataflow runtime engine that performs a software emulation of the dataflow machine. Such a virtual machine has interfaces to the virtual machine language and to C. This interface provides the transparent dataflow semantics for conventional programming. Optionally, the code for the virtual machine can be generated/translated from other high-level languages by some external software tools.

The scope of this work is the BMDFM multithreaded dataflow runtime engine itself and accompanying static preprocessing/loading utilities. Details and extensions of translators from other high-level languages are left for further research.

3.3. Multithreaded Architecture

BMDFM uses both highly efficient dynamic and static scheduling, combining von Neumann, Shared Memory Symmetric Multi Processing (SMP), Multiple Instruction Multiple Data Stream (MIMD) and Data Flow Machine (DFM) paradigms. According to the dataflow classification, BMDFM is a hybrid of tagged-token dataflow, explicit token store, threaded dataflow and large-grain dataflow. The BMDFM architecture is shown in Figure 3.2.

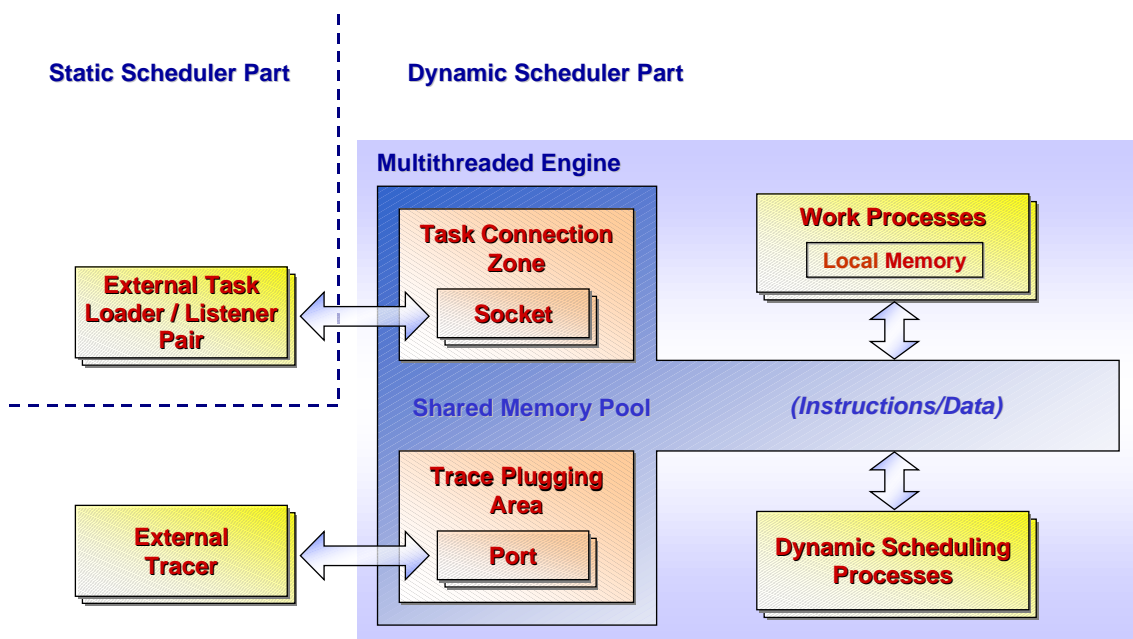


Figure 3.2. BMDFM architecture

A pool of processes is divided into two subsets: work processes, which execute parallel instruction streams, and dynamic scheduling processes, which automatically convert sequential instruction streams into parallel ones.

Running under an SMP OS, the processes will occupy all available real machine processors.

All processes share the shared memory pool containing instructions and data. Each work process also has its own local memory, which may contain user subroutines to implement additional coarse-grain levels of parallelism. The external loader/listener pair performs preprocessing and static scheduling of the input program instructions and stores them clustered in the task connection zone. The listener is responsible for the ordered output after the out-of-order processing in the multithreaded engine. Clustered instructions and data are fetched by the dynamic scheduling processes into the shared memory pool. Additionally, dynamic scheduling processes

release (garbage collect) resources after the data contexts and speculative branches are processed. Lastly, the external tracer assists in debugging the multithreaded out-of-order processing of the input program. The external tracers are connected via the ports of the trace plugging area. The tracer can operate in various modes of full/partial and master/slave debugging.

3.4. Static Scheduler

Figure 3.3 shows the static scheduling part of BMDFM. An application program (input sequential program) is processed in three stages: preliminary code reorganization (code reorganizer), static scheduling of the statements (static scheduler) and compiling/loading (compiler).

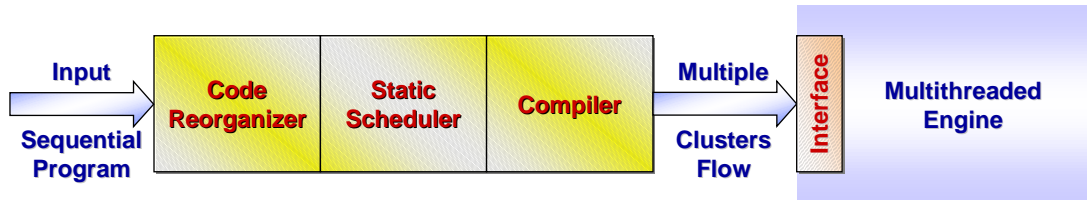


Figure 3.3. Static scheduler

The output after the static scheduling stages is a multiple clusters flow that feeds the multithreaded engine via the interface designed in a way to avoid bottlenecks. The multiple clusters flow can be thought of as a compiled input program split into marshaled clusters, in which all addresses are resolved and extended with context information. Splitting into marshaled clusters allows loading them multithreadedly. Context information lets iterations be processed in parallel.

3.5. Dynamic Scheduler

Figure 3.4 shows the part of BMDFM responsible for the dynamic scheduling in more detail. The BMDFM dynamic scheduling subsystem is an efficient SMP emulator of the tagged-token, explicit token store, threaded DFM.

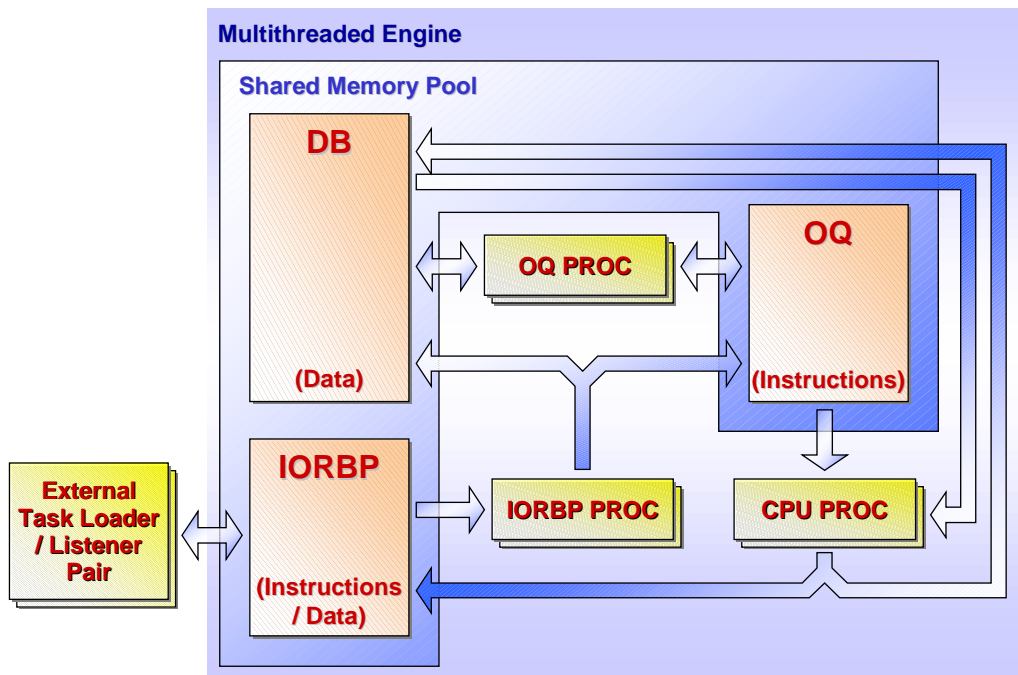


Figure 3.4. Dynamic scheduler

The Shared Memory Pool is divided in three main parts: Input/Output Ring Buffer Port (IORBP), Data Buffer (DB) and Operation Queue (OQ).

The external static scheduler (External Task Loader/Listener Pair) puts clustered instructions and data of an input program into the IORBP. The ring buffer service processes (IORBP PROC) move data into the DB and instructions into the OQ. The operation queue service processes (OQ PROC) tag the instructions as ready for execution if the required operands' data is accessible. Execution processes (CPU PROC) execute instructions, which are tagged as ready and output computed data into the DB or to the IORBP. Additionally, IORBP PROC and OQ PROC are responsible for freeing memory after contexts have been processed. The context is a special unique identifier representing a copy of data within different iteration bodies accordingly to the tagged-token dataflow architecture. This allows the dynamic scheduler to handle several iterations in parallel.

To allow several processes accessing the same data concurrently, the BMDFM dynamic scheduler locks objects in the shared memory pool via SVR4 semaphore operations. Locking policy provides multiple read-only access and exclusive access for modification.

The BMDFM Dynamic Scheduler is configured at startup using a configuration profile. Figure 3.5 illustrates a typical configuration for the 8-way SMP machine.

SHMEM_POOL_SIZE	=	2147483647	# (2GB) Shared memory pool size	[Bytes]
SHMEM_POOL_BANKS	=	10	# Number of banks in pool	
ARRAYBLOCK_SIZE	=	20	# Array block size	
Q_OQ	=	3000	# Operation Queue (OQ) size	[Entities]
Q_DB	=	1000	# Data Buffer (DB) size	[Entities]
Q_IORBP	=	16	# I/O Ring Buffer Port (IORBP) size	[Entities]
N_IORBP	=	4	# Number of the IORBPs	
N_CPUPROC	=	8	# Number of the CPU PROCs	
N_OQPROC	=	8	# Number of the OQ PROCs	
N_IORBPPROC	=	8	# Number of the IORBP PROCs	

Figure 3.5. Typical configuration for the 8-way SMP machine

In our experiments we used the Tuning and Analysis Utilities (TAU) [147, 148] to analyze the performance of BMDFM scheduling routings and to define optimal configuration parameters. TAU uses a timing instrumentation that is triggered at function entry and exit. The instrumentation is responsible for name registration, maintaining the function database, the call stack, and statistics. From the profile data collected, TAU's profile analysis procedures can generate a wealth of performance information for the user. It can show the exclusive and inclusive time spent in each function with nanosecond resolution. Other data includes how many times each function was called, how many profiled functions were invoked by each function, and what the mean inclusive time per call was. On systems where available, TAU can also use hardware performance counters.

SHMEM_POOL_SIZE defines maximal shared memory pool size. Bigger value ensures that BMDFM will operate for larger amounts of data. This value can be limited by the operating system. Normally, a 2GB value is the limit of shared addressable space for 32-bit mode applications. In our experiments we use 64-bit compilation and we set this value to 32GB (34359738368).

SHMEM_POOL_BANKS defines the number of banks in the shared memory pool. To enable access to the shared memory pool from many processes simultaneously, we developed a reentrant driver for shared memory allocation. The driver can run the allocation sequence in every bank independently, which actually causes the shared memory pool to be split into multiple memory banks. Several banks together work faster than one but the memory bank restricts maximal memory block size that can be allocated. CPU PROC processes are the main consumers for the shared memory reallocation routines; IORBP PROC and OQ PROC processes perform only non-intensive resource allocation/release. Therefore it is better to have this value a little bigger than the number of CPU PROCs. Experimentally, we have defined a calculation formula for the number of shared memory pool banks: $SHMEM_POOL_BANKS = 1.25 * N_CPUPROC$.

ARRAYBLOCK_SIZE defines the policy of the memory allocation. All dynamic structures of the shared memory pool are allocated in chunks. Bigger values cause less intensive and faster memory allocation but at the same time cause inefficient memory usage.

Q_OQ defines OQ size. Bigger values allow the running of tasks with more complex data dependencies but a big OQ requires additional memory space, an additional number of semaphores and can slow down associative searches in the dynamic scheduling subsystem. Experimentally, we determined the most appropriate value to be 3000.

Q_DB defines DB size. Bigger values allow running tasks with more variables but a big DB requires additional memory space and an additional number of semaphores. Experimentally, we determined the most appropriate value to be 1000.

Q_IORBP defines the IORBP size. This value can be understood as a maximal number of the buffered marshaled clusters, which are sent to the multithreaded engine from a single instance of the external task loader/listener pair. Bigger values allow more intensive loading of data via the task connection zone but a big IORBP requires additional memory space and an additional number of semaphores. Normally, the IORBP size has to be less than the OQ size to avoid a situation where the dataflow engine is saturated with the OQ full of instructions waiting for some unresolved dependencies from the IORBP. Experimentally, we determined the most appropriate value to be 16.

N_IORBP defines the number of IORBPs, thus the number of tasks (jobs), which can be processed in parallel. Processing several tasks (jobs) simultaneously uses system resources more efficiently. If it is really planned to run many applications on a single BMDFM instance, the DB and OQ sizes have to be increased accordingly.

N_CPUPROC, **N_OQPROC** and **N_IORBPPROC** define the number of CPU PROC, OQ PROC and IORBP PROC processes, respectively. Usually it makes sense to set these values equal to the real number of system processors. Recent hyper-threading technology that allows running multiple threads on one CPU can change this policy only a little bit. Currently, one CPU can run not more than two threads. That means in our configuration every CPU can multiplex between three instances of CPU PROC, OQ PROC and IORBP PROC, roughly. In the future, if one CPU core will be able to run more than two parallel threads, our configuration recommendations could be reviewed. In any case, an additional tuning for these parameters can be done after the analysis of stall warnings in the log files. In our experiments we very often set the **N_OQPROC** value to half the number of system processors.

Even after having determined the most appropriate configuration parameters, we decided not to hard code them but to leave them open in the configuration profile. In this case the end user will have freedom to change the configuration depending on his applications. It is quite possible that some SMP architectures may demonstrate better performance if configured with the number of virtual processes, which considerably exceeds the number of system processors.

3.6. Programming Model

BMDFM can be thought of as a virtual machine, which provides a conventional functional programming model and uses transparent dataflow semantics. No directives for parallel execution are required! From the user's point of view BMDFM is a virtual machine, which runs every statement of an application program in parallel with all parallelization and synchronization mechanisms fully transparent. The statements of an application program are normal operators that any single threaded program might consist of: variable assignments, conditional executions, loops, function calls, etc. BMDFM has a rich set of standard operators/functions, which can be extended by user functions written in C/C++. A BMDFM user application can be built according to the three schemes shown in Figure 3.6.

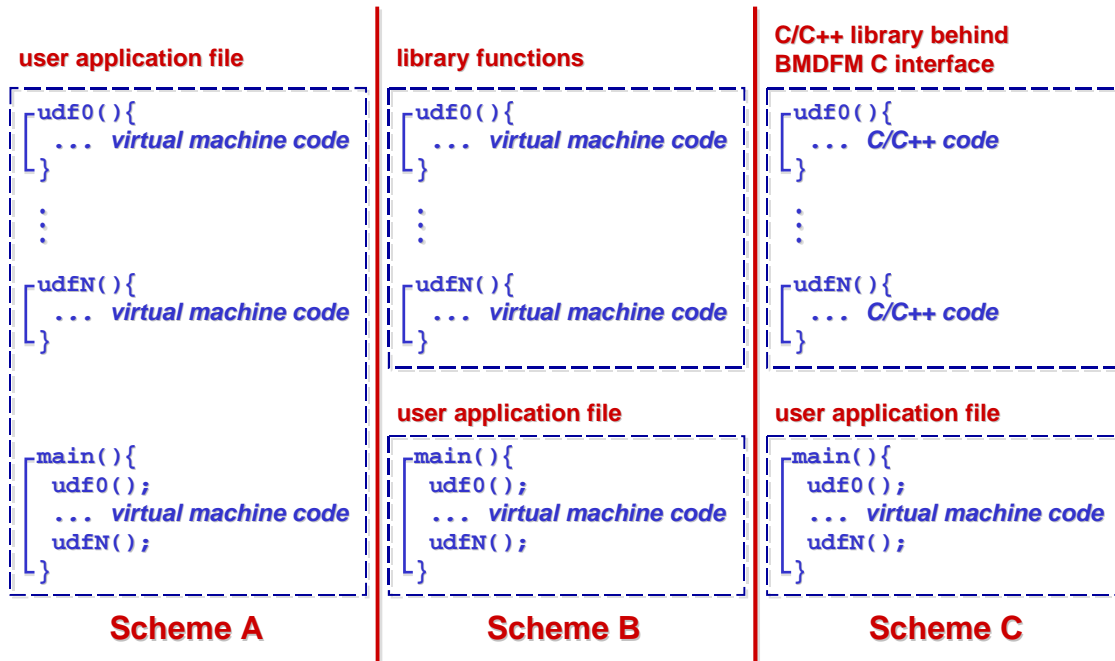


Figure 3.6. BMDFM user programming models

Scheme A. A complete application is written in pure virtual machine language. In this case BMDFM will exploit fine-grain parallelism, thus BMDFM will try to unroll the loops and to execute all statements in parallel. If it runs on a non-UMA (non-Uniform Memory Access) machine the dynamic scheduling can be expensive.

Scheme B. According to this scheme some UDFs (User Defined Functions) will be uploaded into CPU PROCs local memory and their bodies will be prevented from scheduling for parallel processing (such a UDF will be treated as one seamless statement). In this case less time is obviously spent on dynamic scheduling.

Scheme C. This scheme allows using the C code directly instead of the virtual machine code. Of course the C code compiled and optimized by a local C compiler is faster than virtual machine code.

3.7. Virtual Machine Language and C Interface

When we designed a language for BMDFM we chose a subset of LISP with an open C interface, taking into account the following ideas:

- LISP has the simplest syntax ever created in the programming language area, thus it has the simplest syntax checker, parser and byte code generator. That helped us very much in quick prototyping of the complete BMDFM architecture.
- LISP is abstract regarding the data types and convenient enough for manual programming. At the same time the LISP-like prefix format is the de facto standard for the intermediate program presentation after a preprocessing from other high-level programming languages. Similarly to the RTL level of the GCC compiler set, most compilers optimize code presented in a LISP-like prefix format. We used this advantage of LISP because BMDFM is intended for manual programming and for generated code as well.
- LISP efficiently combines features of both algorithmic and artificial intelligence languages that allow using LISP in a wide area of computations.
- The open C interface allows extending the base virtual machine with additional functionality required in an application area. Once a library of specialized functions is written and compiled it can be used from the virtual machine via the C interface. It is a common practice of well-known virtual machines (TCL/TK, Perl, Java) to achieve sufficient performance. Statistically, user applications for the virtual machines use 80% of running time executing the functions from the provided libraries, compiled for the target architecture.

The following Table 3.1 describes the BMDFM virtual machine language briefly. We spend several pages for that because this information is necessary for understanding further examples given in this thesis.

Syntax	
<p>A function may appear as (<func_name> <par_1> ... <par_N>). As a result it returns its calculated <value>. It is possible to give a constant value, variable name, actual parameter/argument or other function in place of the functions' parameters, for instance: (* 2 (+ 2 2)) returns 8.</p> <p>A complex function is a (progn <func_name_1> ... <func_name_N>) encapsulation. As result it returns its last calculated value. A <program> is a function. Formal parameters/arguments appear like \$1 ... \$N in the program's body. All actual parameter/argument types will automatically be converted to the required ones in all cases possible during the actual function calls.</p> <p>'#' char is assumed as a comment symbol. All the rest of the line after '#' char will be ignored.</p>	
Variable assignments	
([al]setq <quota_var_name> <func_name>)	Sets value to a variable
(arsetq <quota_var_name> <func_name_index> <func_name>)	Sets value to an array member
(index <quota_var_name> <func_name_index>)	Array member
(alindex <quota_var_name> <func_name_indices>)	Number of array members
User Defined Function (UDF)	
(defun <quota_func_name> <program>)	
Conditional	
(if <func_name_?> <func_name_true> <func_name_false>)	
Loop	
(while <func_name_?> <func_name_body>)	
(for <quota_ctrl_var_name> <func_from_ctrl_var> <func_step_ctrl_var> <func_to_ctrl_var> <func_name_body>)	
(break)	Cancels an iteration or a UDF execution of the current nested level
(exit)	Cancels a program execution
I/O	
(accept <func_prompt_message_for_console_or_empty_for_stdin>)	
(scan_console <wait_keypress_forever_if_1_or_useconds_if_positive>)	
(outf <func_C_like_printf_format> <func_name>)	
(file_create <file_name>)	File descriptor or -1 if error
(file_open <file_name>)	File descriptor or -1 if error
(file_write <file_descriptor> <string_to_be_written>)	Number of bytes written or -1 if error
(file_read <file_descriptor> <number_of_bytes_to_be_read>)	String read or empty string if error
(file_close <file_descriptor>)	Zero or -1 if error
(file_remove <file_name>)	Zero or -1 if error
Compare	
(== ...) (!= ...) (< ...) (> ...) (<= ...) (>= ...)	Names are in C notation
Boolean (short-circuit evaluation)	
(&& ...) (...) (! ...)	Names are in C notation
Integer	
(ival <Val>)	Explicitly converts to integer
(indices <Val>)	Number of array indices
(irnd <range>)	Random number within the range of 0 to <range>
(+ ...) (- ...) (* ...) (/ ...) (% ...) (++ ...) (-- ...)	Names are in C notation
(*+ ...)	Multiplication and addition
(0- ...)	Negation
(iabs ...)	Absolute value
(& ...) (! ...) (^ ...) (~ ...) (>> ...) (<< ...)	Names are in C notation
Float	
(fval <Val>)	Explicitly converts to float
(frnd <range>)	Random number within the range of 0 to <range>
(+. ...) (-. ...) (*. ...) (/)	Names are in C notation
(*+. ...)	Multiplication and addition

(fabs ...)	Absolute value
(int ...) (round ...) (cos ...) (sin ...) (atan ...)	Names are in C notation
(cas ...)	Sine plus cosine
(exp ...) (pow ...) (sqrt ...)	Names are in C notation
(ln ...)	Logarithm of e base
String	
(str <Val>)	Explicitly converts to string
(chr <I_val>)	String of one character
(asc <S_val>)	ASCII code of the first character
(type <Val>)	Value type among I, F, S, Z (Z for nil value)
(dump_i2s <I_val>)	Memory dump of integer
(dump_f2s <F_val>)	Memory dump of float
(dump_s2i <string>)	Integer mapped from string
(dump_s2f <S_val>)	Float mapped from string
(notempty <S_val>)	True if string is not empty
(len <S_val>)	Length of string
(at <pattern> <within>)	Occurrence position searching from left
(rat <pattern> <within>)	Occurrence position searching from right
(cat <S_val> <S_val>)	Concatenation
(space <space_number>)	String of spaces
(replicate <pattern> <repetition_number>)	Replication
(left <string> <position_from_left>)	Left part of string
(leftr <string> <position_from_right>)	Left part of string
(right <string> <position_from_right>)	Right part of string
(rightl <string> <position_from_left>)	Right part of string
(substr <string> <position> <number>)	Substring
(strtran <string> <pattern> <substitution>)	Find and replace
(str_raw <S_val>)	Raw string
(str_unraw <S_val>)	Special characters are slashed
(str_dump <S_val>)	Semi-hexadecimal string dump
(str_fmt <C_like_printf_format> <value>)	Formatted string
(ltrim <S_val>)	String without leading blanks
(rtrim <S_val>)	String without ending blanks
(alltrim <S_val>)	String without leading and ending blanks
(pack <S_val>)	String without redundant blanks
(head <S_val>)	First token of string
(tail <S_val>)	Remaining tokens of string
(upper <S_val>)	Upper case string
(lower <S_val>)	Lower case string
(rev <S_val>)	Back ordered string
(padl <string> <width>)	Left justified string
(padr <string> <width>)	Right justified string
(padc <string> <width>)	Centered string
(time)	Current time
(getenv <environment_variable>)	Environment value
Constants	
(ee)	E
(gamma)	Aler-McCheroni constant
(phi)	Golden
(pi)	Pi
(prn_integer_fmt)	Default integer format “%ld”
(prn_float_fmt)	Default float format “%.16E”
(prn_string_fmt)	Default string format “%s”

(reinit_terminal <terminal_type_or_empty>)	Terminal status
(term_type)	TERM environment
(lines_term)	termcap(li)
(columns_term)	termcap(co)
(clrscr_term)	termcap(cl)
(reverse_term)	termcap(mr)
(blink_term)	termcap(mb)
(bold_term)	termcap(md)
(normal_term)	termcap(me)
(hidecursor_term)	termcap(vi)
(showcursor_term)	termcap(ve)
(gotocursor_term <y_coord> <x_coord>)	termcap(cm)
(n_cpuproc)	Number of CPU PROCs currently configured
Mapcar	
(mapcar <program>)	(<preprinted_info> <result_of_execution> <syntax_error_code> <syntax_error_message> <runtime_error_code> <runtime_error_message> <processed_function> <processed_function_compiled> <processed_function_linked> <time_spent_in_seconds>)
Asynchronous Heap	
(asyncheap_create <bytes>)	Descriptor
(asyncheap_getaddress <descriptor>)	Physical address
(asyncheap_putint <descriptor> <offset> <integer>)	1
(asyncheap_getint <descriptor> <offset>)	Integer value
(asyncheap_putfloat <descriptor> <offset> <float>)	1
(asyncheap_getfloat <descriptor> <offset>)	Float value
(asyncheap_putstring <descriptor> <offset> <string>)	1
(asyncheap_getstring <descriptor> <offset> <length>)	String value
(asyncheap_reallocate <descriptor> <bytes>)	New descriptor
(asyncheap_replicate <descriptor>)	New descriptor
(asyncheap_delete <descriptor>)	1

Table 3.1. BMDFM virtual machine language

The virtual machine language is first compiled into byte code and is then linked to resolve all external references. Figure 3.7 explains the structure of the byte code we designed for our BMDFM project. The structure is trivial but at the same time is flexible enough to extend the virtual machine with a C function library and to relocate the byte code harmlessly into the shared memory pool.

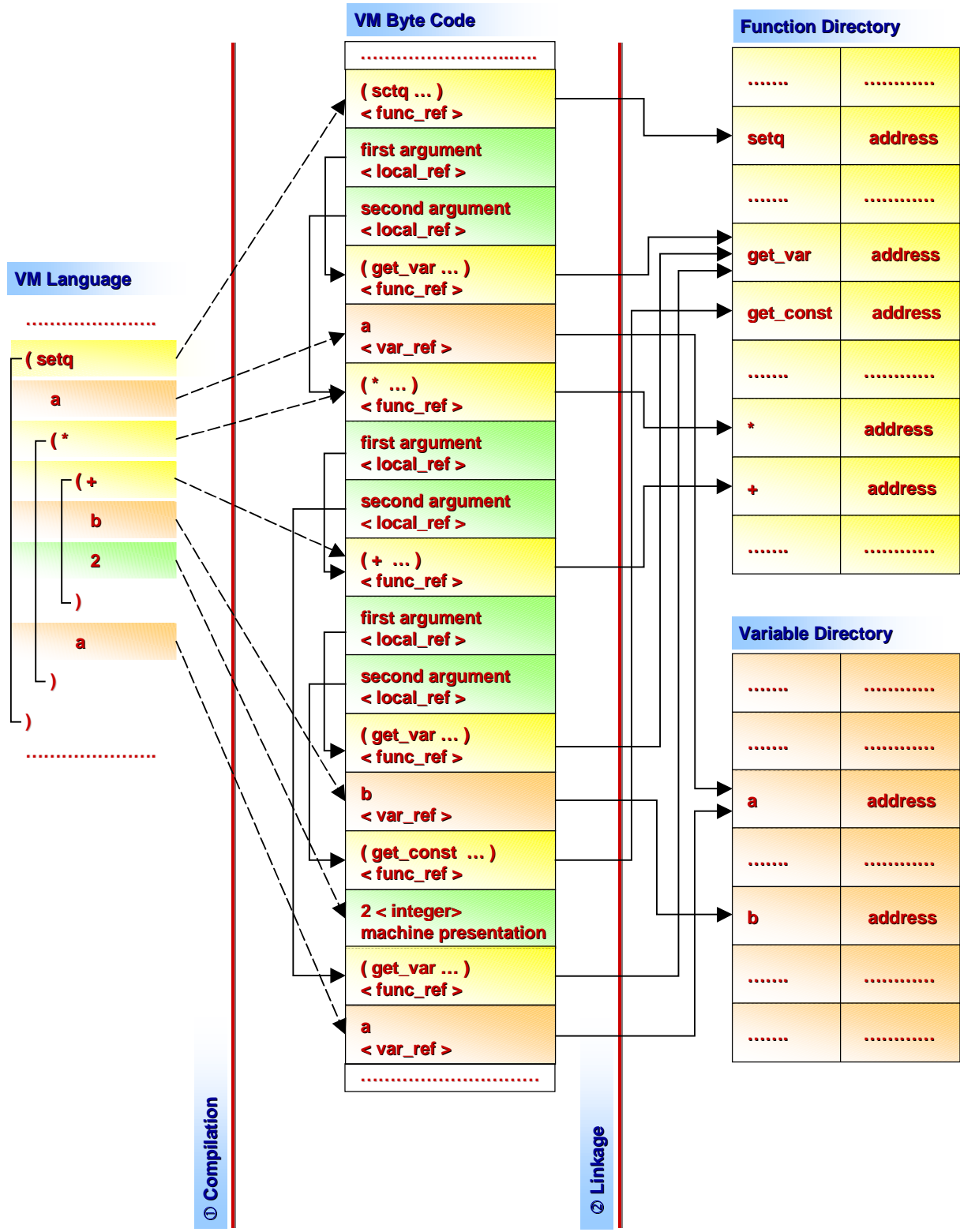


Figure 3.7. Byte code generation

The byte code fragment shown reflects one to one the “(setq a (* (+ b 2) a))” expression of the virtual machine language. Every function from the byte code portion refers to the common function directory and is followed by a list of the arguments’ local references. This list is required to represent different offsets to the real locations of the arguments. Predefined constants are stored in the byte code directly but they are accessed through the “get_const” function call at runtime. Variable references point to the common variable directory.

The function directory contains all standard functions, virtual machine functions defined via “defun” and functions added through the C interface. Normally, the directory is fixed after a successful compilation/linkage stage. The exception case is the “mapcar” invocation, which can force dynamic expansion of the function directory. The variable directory is dynamically created in the stack every time the runtime engine enters a virtual machine UDF recursively.

In addition to the virtual machine language, BMDFM provides an open C interface, which is described below. To simplify declaration constructions we use the following abbreviations for the standard types.

```
#define UCH unsigned char
#define SCH signed char
#define USH unsigned short int
#define SSH signed short int
#define ULO unsigned long int
#define SLO signed long int
#define DFL double
```

The virtual machine stores each variable in the universal structure that enables it to change data types dynamically and to have a single value or an array with different types of members, thus supporting lists and trees. Declaration of a variable of the universal structure type allocates single value on the stack and array in the heap that is very convenient assuming most variables store only single values. No memory overhead is needed for storing arrays with members of the same type.

```
struct fastlisp_data{
  UCH disable_ptr; // 1 = stores value, 0 = ptr to a variable possible
  UCH single; // 1 = single value, 0 = array
  UCH type; // 0=undef, 'I'=int, 'F'=float, 'S'=string, 'Z'=nil
  UCH arraytype; // 0=undef, 'I'=int, 'F'=float, 'S'=string, 'Z'=nil
  union{
    SLO ival; // integer value
    DFL fval; // float value
  } value;
  UCH *svalue; // string value
  ULO indices_num; // number of indices in the array
  UCH *aready_tags; // flags 'OIFSZ' for arraytype!=0 for every member
  union{
    struct fastlisp_data *mix; // array members of mixed types
    SLO *ival; // array members of integer type
    DFL *fval; // array members of float type
    UCH **svalue; // array members of string type
  } array;
};
```

A user C function can be defined through the following type declaration.

```
typedef void (*fcall)(ULO*, struct fastlisp_data*);
```

The first argument is a pointer to the function parameters/arguments and the second argument is a pointer to the result structure. Passed parameters/arguments can be obtained from inside the function via the following set.

```
void ret_ival(ULO *dat_ptr, SLO *targ); // gets integer or pointer value
void ret_fval(ULO *dat_ptr, DFL *targ); // gets float value
void ret_sval(ULO *dat_ptr, UCH **targ); // gets string value
void (*fcall)(ULO*, struct fastlisp_data*); // gets universal structured data
```

There are two additional functions for copying and deleting the universal data structure.

```
void copy_flp_data(struct fastlisp_data *dest, struct fastlisp_data *source,
  ULO indices_num);
void free_flp_data(struct fastlisp_data *ret_dat);
```

The last step, which should be performed after a user C function is defined, is to fill the virtual machine instruction database according to the following structure.

```
typedef struct{
  UCH *fnc_name; // function name
```

```

SSH operands; // number of arguments
UCH ret_type; // result type: 'I'=int, 'F'=float, 'S'=string, 'Z'=nil
UCH *op_type; // flags 'IFSZ' for every argument
fcall func_ptr; // pointer to the function
} INSTRUCTION_STRU;

```

Figure 3.8 gives an example of a user defined C function. According to the byte code structure the function's actual parameters/arguments are obtained sequentially through the incremented "dat_ptr". Internal calls to "ret_ival", "ret_fval" and "ret_sval" provide dynamic type casting if required. Direct "fcall" function invocation omits the dynamic casting and returns a universal data structure.

```

void my_function(ULO *dat_ptr, struct fastlisp_data *ret_dat){
    ULO *tmp_ptr;
    SLO n,result=0;
    UCH *str=NULL;
    DFL *f_array,koef;
    struct fastlisp_data idat={0,1,0,0,{0},NULL,1,NULL,{NULL}};

    ret_ival(dat_ptr,&n); // arg0: integer (n)
    ret_ival(dat_ptr+1,(SLO*)&f_array); // arg1: ptr to floats (f_array)
    ret_fval(dat_ptr+2,&koef); // arg2: float (koef)
    ret_sval(dat_ptr+3,&str); // arg3: string (str)
    tmp_ptr=((ULO**)(dat_ptr+4)); // arg4:
    (*(fcall)*tmp_ptr)(tmp_ptr+1,&idat); // integer as universal data (idat)

    if(noterror){
        // ...
        // data processing to compute 'result' or noterror=0 if error;
        // ...
        ret_dat->single=1;
        ret_dat->type='I';
        ret_dat->value.ival=result;
    }

    if(idat.disable_ptr)
        free_flp_data(&idat);
    free_string(&str);

    return;
}

INSTRUCTION_STRU INSTRUCTION_SET[]={
    {"MY_FUNCTION",5,'I',"IIFSI",&my_function}
};
const ULO INSTRUCTIONS=sizeof(INSTRUCTION_SET)/sizeof(INSTRUCTION_STRU);

```

Figure 3.8. A user defined C function

Finally, "my_function" is registered in the virtual machine instruction database. The corresponding record states that the function has five parameters/arguments and returns an integer value. Parameter/argument types are integer, integer, float, string and integer respectively. Specification of the parameter/argument types is not strictly necessary because the virtual machine casts the data types dynamically, but such a specification can help an internal optimizer to generate more efficient byte code.

3.8. Workflow for Applications

Normally, the life cycle of a BMDFM user application has two major steps as shown in Figure 3.9. First the application is developed and tested under the BMDFM single threaded engine, then if it works properly it can be moved without any modifications into the BMDFM multithreaded engine.

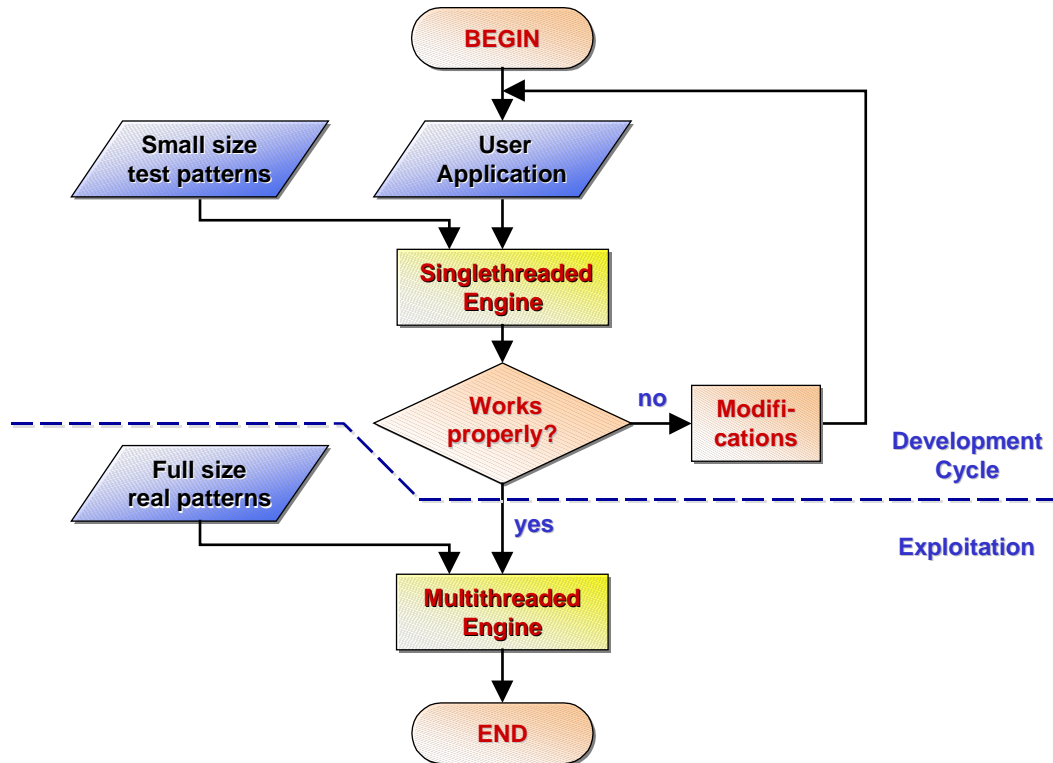


Figure 3.9. Application life cycle

The proposed BMDFM workflow for the applications fully follows conventional approaches. The application design and debug processes can be accomplished on an inexpensive PC for example. Later, after the application has reached a certain state of maturity, it is run on an SMP mainframe multithreadedly in a batch mode using full size input data patterns. Two important features have to be mentioned in the context of BMDFM:

- If the application uses specific functions defined in C they have to be linked with both single threaded and multithreaded BMDFM engines. The BMDFM byte code structure and the C interface are designed in such a way that no modifications in C code are required.
- The BMDFM single threaded engine warns about all variables, which are used without a prior initialization. Such an uninitialized variable endangers dataflow processing, bringing it to an endless idle state (this comes from the dataflow principle itself). The severity of “warning” is changed to “error” in the multithreaded dataflow engine to prevent potential blocking.

3.9. Running Applications on a Single Threaded Engine

The BMDFM single threaded engine compiles, links and runs a user application in standalone mode as shown in Figure 3.10. If the application uses specific functions defined in C they have to be linked with the runtime engine.

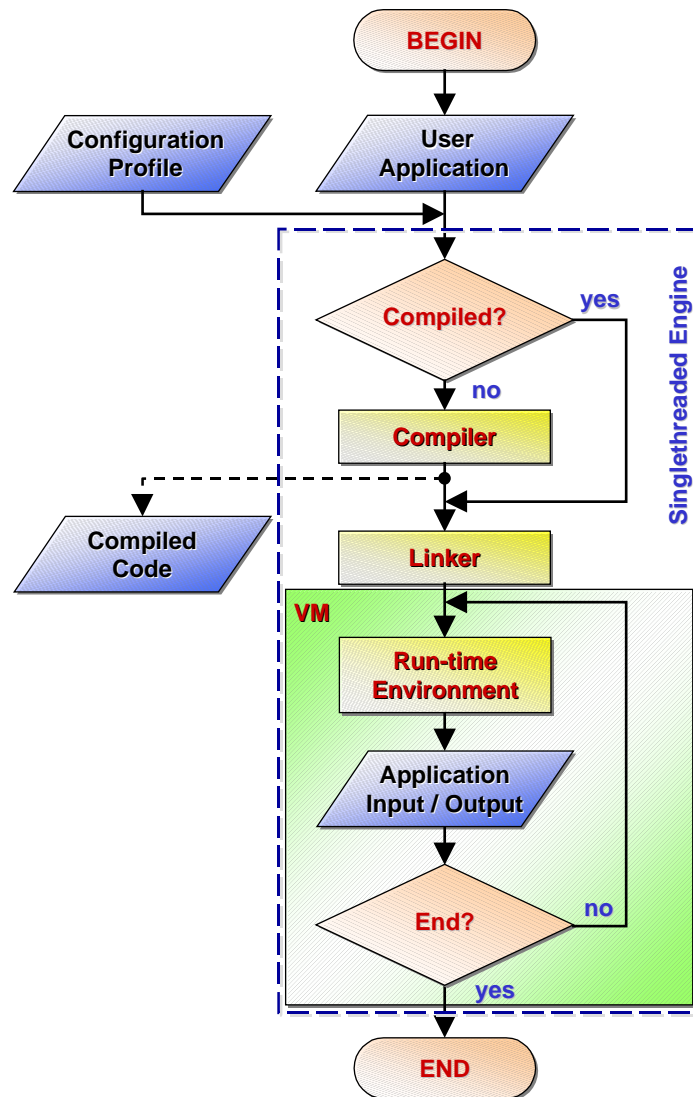


Figure 3.10. Single threaded engine workflow

This is a classic scheme used in all other (TCL/TK, Perl, Java) virtual machines. In our case we use this workflow only for preliminary preparations before any application is moved to the BMDFM multithreaded engine.

3.10. Running Applications Multithreadedly

Multithreaded workflow for the applications is implemented in a client/server manner. The BMDFM multithreaded runtime engine acts as a server accepting and processing the connection sessions. The external task load/listener pair acts as a client virtual machine uploading the application to the server.

The BMDFM server unit reads the configuration profile, initializes the shared memory pool, starts multiple copies of the daemons (CPU PROCs, OQ PROCs, IORBP PROCs and an additional PROC stat daemon that collects statistic information) in the background and enters the console mode. The BMDFM server unit is also responsible for shutting down the whole multithreaded engine correctly. This procedure is illustrated in Figure 3.11.

The external task unit reorganizes the user code, makes static scheduling, compiles, connects the multithreaded engine, links, starts the listener thread and uploads the user application into the multithreaded engine as shown in Figure 3.12.

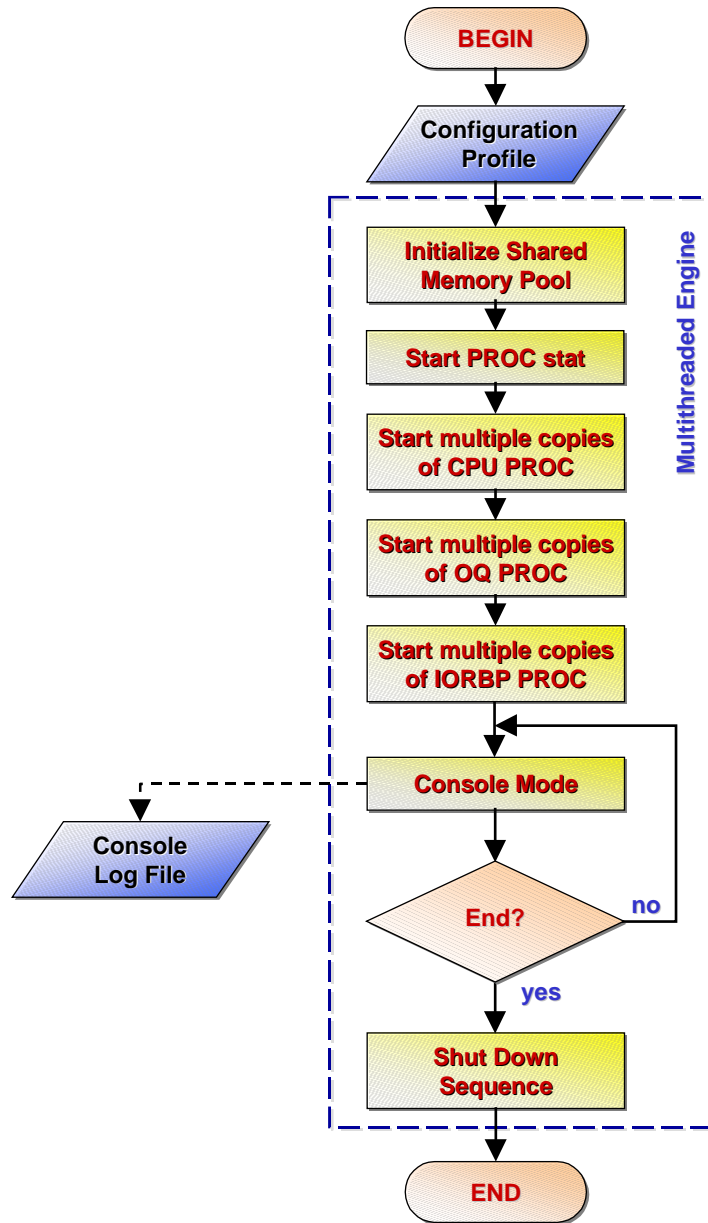


Figure 3.11. BMDFM server unit workflow

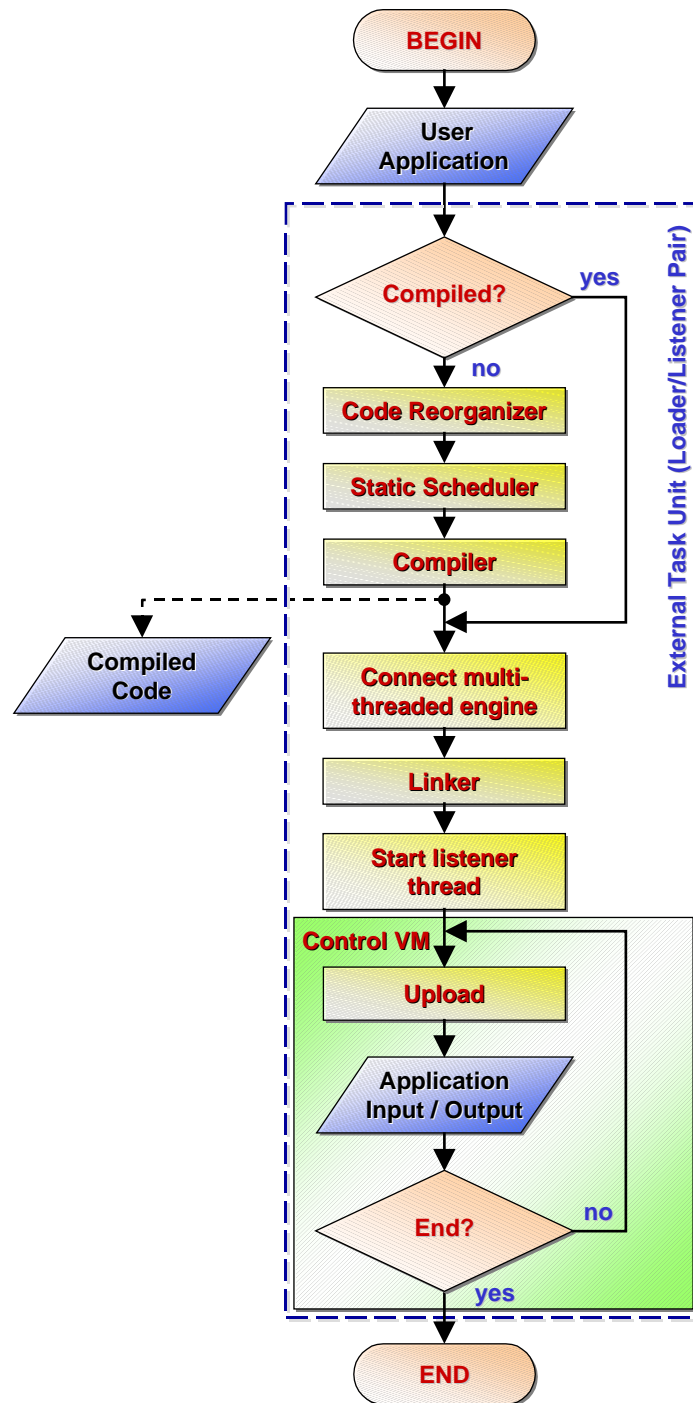


Figure 3.12. External task unit workflow

The front-end virtual machine (Figure 3.13) plays a different role than in the case of a single threaded workflow. It does not execute the byte code of an application but it uploads the code dynamically to the dataflow runtime engine.

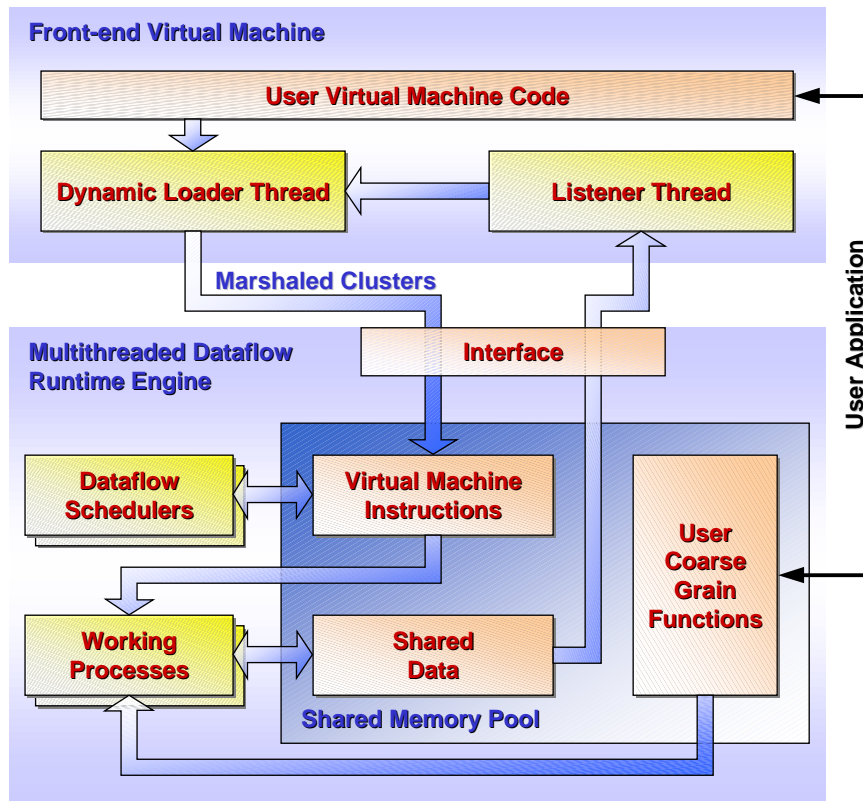


Figure 3.13. Multithreaded engine workflow

Depending on programming model a user application is scheduled differently. In “scheme A” all byte code resides in the front-end virtual machine. Each statement of the code acts as a standalone virtual machine instruction and is scheduled in the dataflow engine. In “scheme B” and “scheme C” the dataflow engine uses coarse-grain parallelism of the user functions. The last schemes are preferable because they reduce dynamic scheduling overhead.

3.11. Summary

We have designed the BMDFM architecture as a hybrid of the dataflow machine built on commodity SMP hardware. The BMDFM runtime engine is an efficient SMP emulator of the tagged-token, explicit token store, threaded dataflow machine.

The aim we pursued was to use the dataflow runtime engine in the role of a fork-join runtime library. This idea brings a solution to the following problems:

- Compile-time strategies are useless against parallel operations that take a non-deterministic amount of time, making it difficult to know exactly when certain pieces of data will become available. This issue is solved in the dataflow runtime engine in a natural way.
- The dataflow runtime engine can resolve complex dependencies, which can not be detected during the compilation stage.
- The dataflow approach eliminates the fork-join drawback of the idle time, in which program execution has to wait for the completion of the slowest thread.

Our architectural approach has the following important features that are not present in known runtime parallelization projects:

- The dataflow runtime engine is not aggressively optimized for the applications in some specific areas such as numeric processing, for example. It can solve inter-procedural and cross-conditional dependencies as well.

- The dynamic scheduling subsystem is decentralized and is executed in parallel on the same multiprocessors that run the application itself. This approach eliminates a situation where the task scheduling becomes a bottleneck of the entire computing process.
- An application is comprised of the conventional virtual machine language and classic C. There is no special language to control dataflow. The application program itself controls dataflow fully automatically and transparently.
- From the point of view of dataflow programming our approach excludes the problem of a single assignment paradigm. We think that our way of dataflow programming with a conventional algorithmic language can remove the known gap of a missing programming methodology for dataflow.

Chapter 4

Dynamic Scheduling Subsystem

- 4.1. Overview**
- 4.2. Inter Process Synchronization**
- 4.3. Shared Memory Pool**
- 4.4. Non-Dead-Locking Policy**
- 4.5. Inter Process Communication**
- 4.6. Task Connection Zone**
- 4.7. I/O Ring Buffer Ports**
- 4.8. Data Buffer**
- 4.9. Operation Queue**
- 4.10. IORBP Scheduling Process**
- 4.11. OQ Scheduling Process**
- 4.12. CPU Executing/Scheduling Process**
- 4.13. Complexity of the Dynamic Scheduling Subsystem**
- 4.14. Summary**

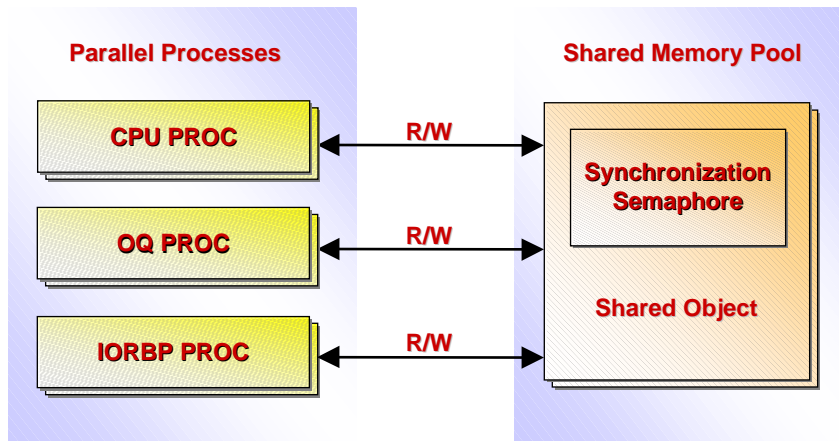
4.1. Overview

In this chapter we discuss the BMDFM dynamic scheduling subsystem, which is an efficient emulator of the tagged-token dataflow built upon shared memory. We have carefully optimized all dynamic shared memory structures and scheduling algorithms to reach high performance. We share our experience on how to avoid bottleneck and dead-locking effects. The following ideas are described:

- The architecture of the shared memory pool split on multiple banks.
- Semaphore synchronization of the parallel process-distributed dynamic scheduler.
- The architecture of an abstract shared memory dataflow machine.
- Effective parallel scheduling algorithms.
- Ordering of a sequential output stream after out-of-order dataflow processing.
- Program complexity of the dynamic scheduling subsystem.

4.2. Inter Process Synchronization

To synchronize multiple parallel processes of the BMDFM dynamic scheduling subsystem we use standard SVR4 IPC semaphores. When several processes access the same data concurrently in the shared memory pool the semaphore locking policy provides multiple read-only access and exclusive access for modification excluding possible data WAR/RAW/WAW hazards. This basic synchronization principle is shown in Figure 4.1. The synchronization semaphores themselves are also stored in the shared memory pool together with the shared objects they control.



Semaphore init: `SEM_INIT(MAX_VAL);`
Object read: `SEM_OP(-1); RD; SEM_OP(+1);`
Object write: `SEM_OP(-MAX_VAL); WR; SEM_OP(+MAX_VAL);`

Figure 4.1. Basic synchronization paradigm

Although the standard SVR4 IPC semaphore provides a broad range of operations (such as “undo” for example), we use a subset consisting of only three main functions:

- **Semaphore initialization** initializes a semaphore with a maximal possible value. This operation is performed only once when the semaphore is created. In our implementation all semaphores are created in the shared memory pool during the startup sequence and remain there until the dataflow engine is closed.
- **Object read** operations decrement the semaphore value by one before reading and increment the value back after reading is finished. A decrement can suspend the process to an idle state if the semaphore value is not positive. Thus all parallel processes can read the object simultaneously.
- **Object write** operations decrement the semaphore by the maximal value before writing and restore the value after. The calling process will be suspended if at least one other process is already accessing the object. Thus only one process can modify the object.

Normally, the semaphore synchronization is considered to be an expensive one but we use it taking into account portability issues. The three mentioned operations are implemented as separate procedures in the source code and can be changed if the target SMP hardware provides some specific inexpensive type of synchronizers (special mutexes, barriers, etc.).

One other important issue regarding the semaphores is that the operating system can allocate only a limited number of them. This number sometimes can be insufficient especially in a case when the dataflow engine is configured for a big data buffer (DB) and/or operation queue (OQ). Therefore we came up with idea of an interleaved semaphore distribution within the shared memory pool as it is illustrated in Figure 4.2.

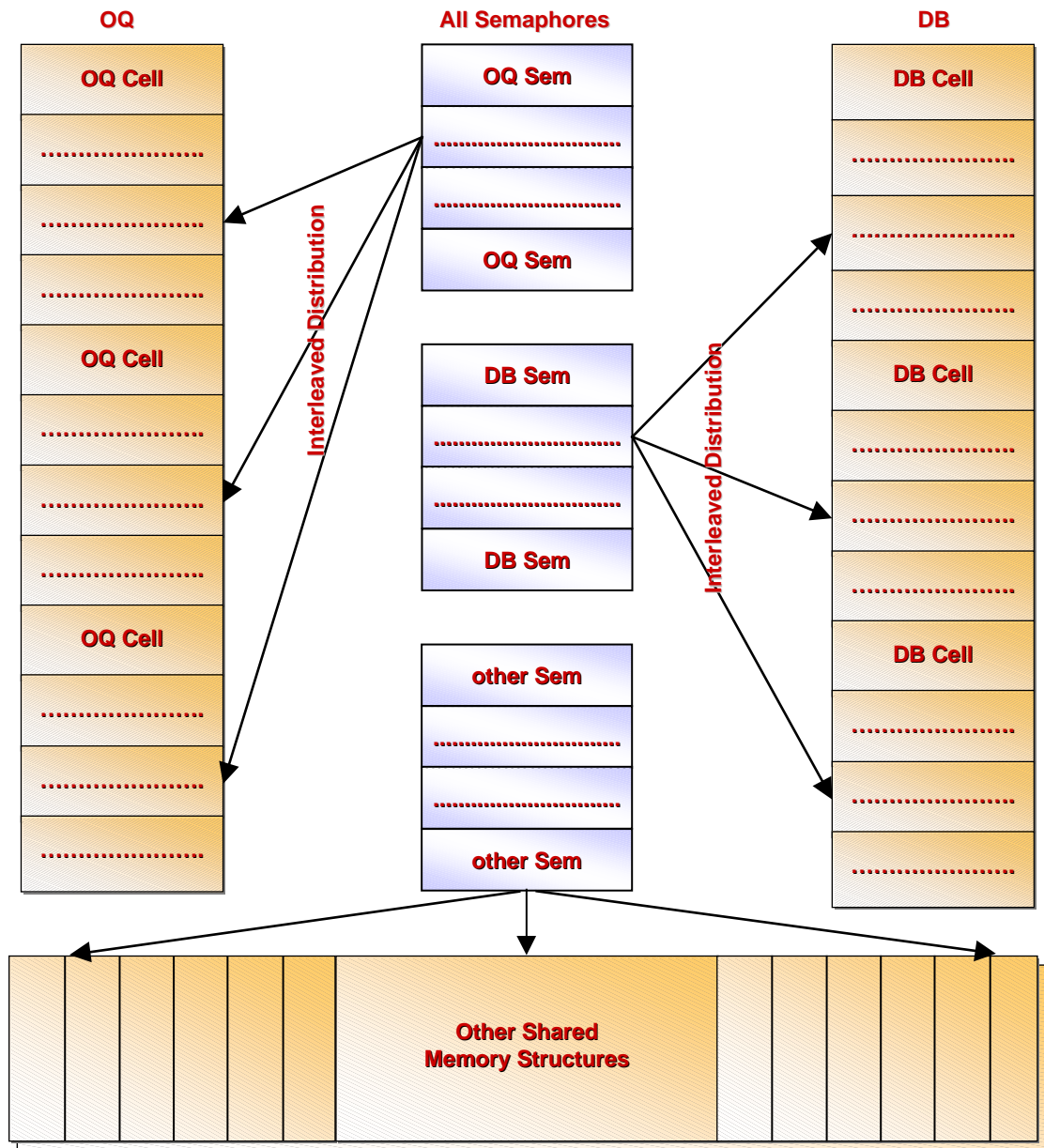


Figure 4.2. Interleaved semaphore distribution in the shared memory pool

All semaphores are divided into three groups: OQ semaphores, DB semaphores and others. The ratio of semaphores for a given semaphore group to the total number of semaphores is equal to the ratio of resources in the corresponding resource group to the total number of resources. Thus, the number of semaphores in each group is calculated according to the proportional sizes of OQ, DB and other shared memory structures. Then each cell number (index) will possess a semaphore from the appropriate group interleaved through the modulo operation ($\text{index} \% \text{number_of_semaphores_in_group}$). Such a distribution has two important features:

- Neighbouring cells will not be assigned to the same semaphore, which increases the probability that independent access attempts will not block each other.
- Different shared memory pool resources will not be intersected through the same semaphores, which prevents a dead-locking situation where a parallel processing is trying to block one resource keeping another resource blocked.

The same distribution policy is applied to all other shared memory zones (shared areas of the memory pool to be protected for a concurrent access). Figure 4.3 shows an access mechanism to the shared zones within the range of 0 to Number_of_zones. To save on the number of the dedicated semaphores we use only N semaphores, supposing that N is less than Number_of_zones. Parallel processes PROCs, which access the shared zones

randomly, select the blocking semaphore $i\%N$. Such simple modulo calculations take a negligible amount of time and can be easily performed at runtime.

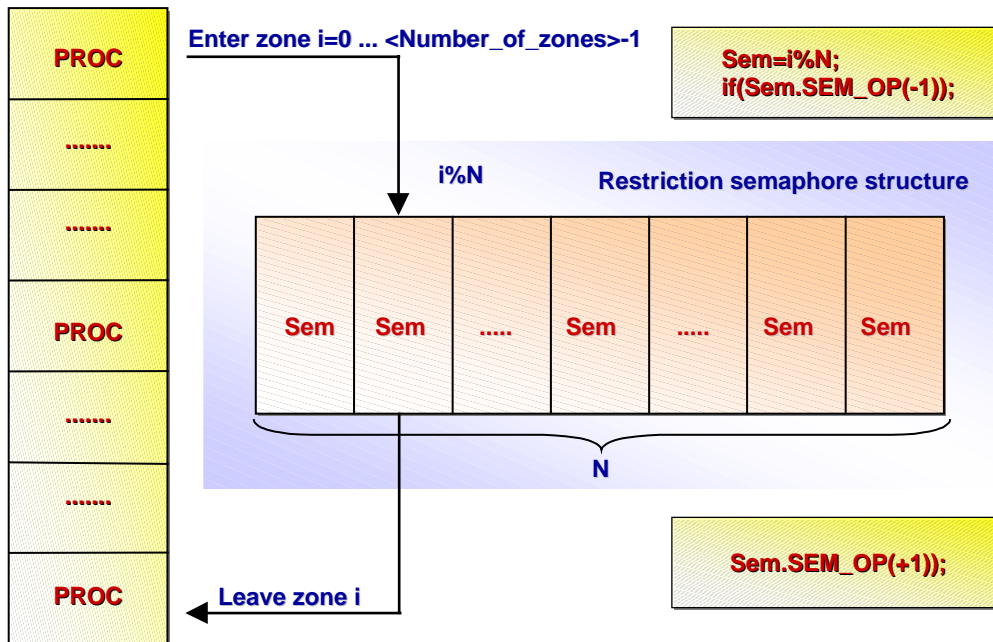


Figure 4.3. Reduced number of blocking semaphores for the shared zones

The dataflow engine itself is also synchronized through the semaphores. For this purpose we have built a common semaphore array, which consists of only twelve semaphores enumerated and explained below:

[DFMSrv]: Common Semaphores Array:		
[DFMSrv]: N#	Meaning	SemVal
[DFMSrv]: 0	for the DFM child PROCs blocking	1
[DFMSrv]: 1	for the PROCs msg pipe	1
[DFMSrv]: 2	for the CPU logs	1
[DFMSrv]: 3	for the statistic	1
[DFMSrv]: 4	for the Task Connection Zone	32767
[DFMSrv]: 5	for the Trace Plugging Area	32767
[DFMSrv]: 6	number of used entities in IORBPs	0
[DFMSrv]: 7	number of free entities in DB	500
[DFMSrv]: 8	number of free entities in OQ	5000
[DFMSrv]: 9	for the changes fixing	0
[DFMSrv]: 10	CPU PROCs awaker	0
[DFMSrv]: 11	OQ PROCs awaker	0

Semaphore 0 is used by the BMDFM external tracer. When the tracer disables the semaphore value it creates a frozen state of the entire dataflow engine, thus allowing step-by-step dataflow debugging.

Semaphore 1 blocks the output from the parallel dynamic scheduling processes to the server console. The output contains stall warnings and statistic information.

Semaphore 2 controls CPU and IORBP PROCs when they log their activities to the common log file. The log file is useful to analyze a dataflow graph of the running application.

Semaphore 3 enables/disables the PROCstat auxiliary process to collect a statistic from the dataflow kernel. Statistic information shows the peak and average use of dataflow resources.

Semaphore 4 is used by the external loader/listener pair when a user application connects or disconnects the dataflow engine.

Semaphore 5 is responsible for the registration of the external tracer in the trace plugging area.

Semaphore 6 stores the number of cells used in the I/O ring buffer ports. Storing of the resource usage value directly on the semaphore considerably simplifies synchronization issues between the parallel processes that try to occupy the resources and those that use them. Thus, an external task puts the marshaled clusters to the ring buffers increasing the semaphore value that signals an event for all waiting IORBP PROCs to fetch the cluster into the dataflow engine.

Semaphore 7 stores the number of free cells in the data buffer. As soon a DB cell is released all waiting IORBP PROCs are allowed to occupy the DB cell and decrement the semaphore value again.

Semaphore 8 stores the number of free cells in the operation queue. IORBP PROCs tries to put a new instruction into the OQ performing the SEM_OP(-1) operation. CPU PROCs execute the instructions releasing them from the queue. Release of the OQ cell is done through the SEM_OP(1) operation that in its turn allows new instructions to be put into the queue.

Semaphores 9 through 11 are the event signaling semaphores for the external tracer. They are used only when at least one tracing engine is connected to the dataflow engine. Semaphore 9 signals the tracer about changes in the dataflow engine, then the tracer signals back through the awakers to continue until the next change occurs.

4.3. Shared Memory Pool

The shared memory pool plays a key role in the entire BMDFM architecture because all BMDFM parallel processes synchronize their activities through the shared memory. According to our approach the shared memory pool stores synchronization semaphores, the complete data structure of the dataflow engine and data allocated by the user applications. Both the dynamic scheduling subsystem itself and running user applications can allocate/free memory blocks dynamically, therefore we designed our shared memory pool in a way to avoid a bottleneck of the memory allocation. Internally, the shared memory pool is divided into memory banks that can be accessed simultaneously by each BMDFM parallel process running its own copy of the reentrant shared memory pool driver. The shared memory pool architecture is shown in Figure 4.4.

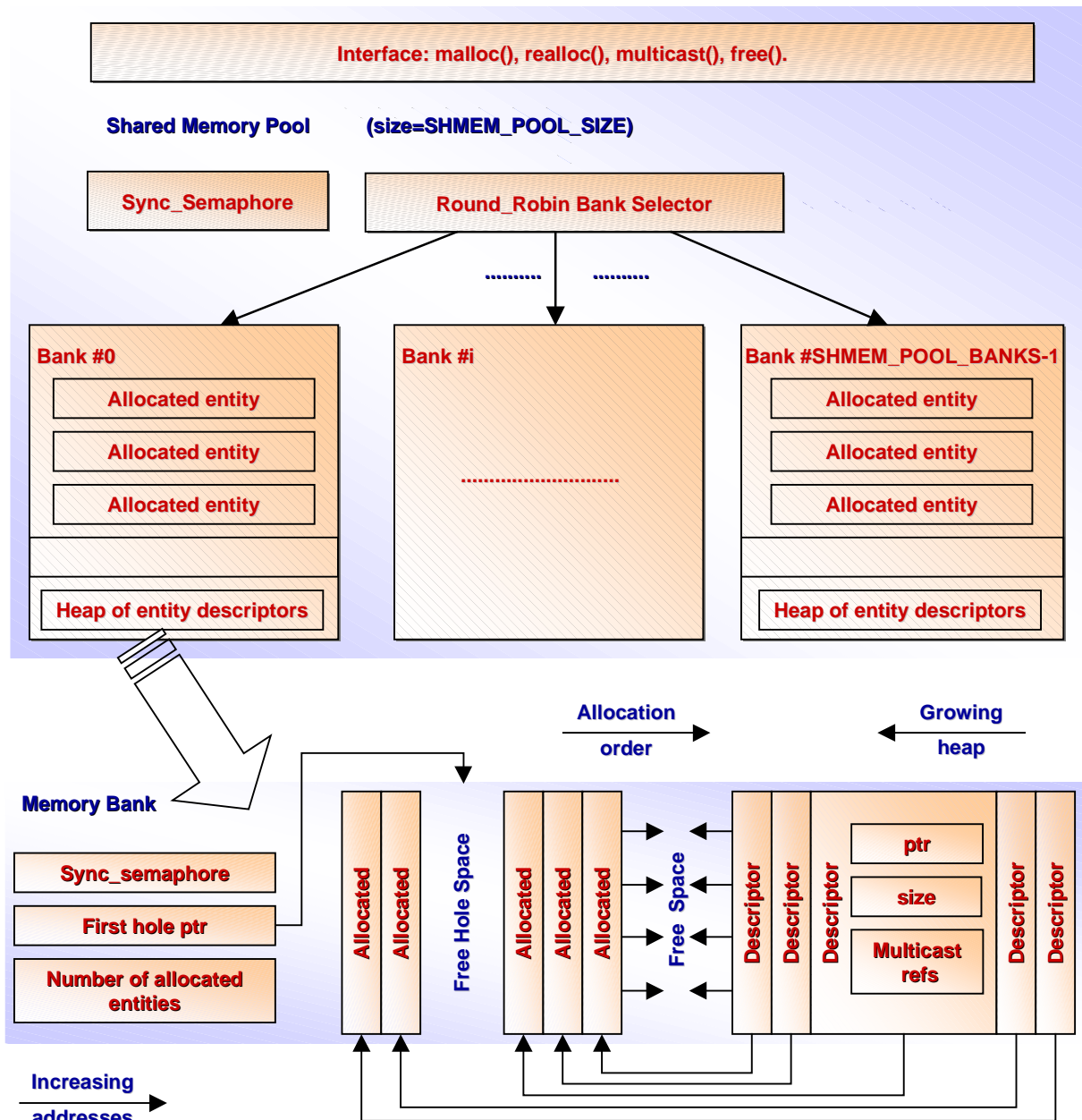


Figure 4.4. Architecture of the shared memory pool

The reentrant memory driver provides a conventional memory allocation interface explained in Table 4.1. Functions malloc(), realloc() and free() comprise a complete set to control a memory allocation process but in our implementation we use one additional function multicast() to assign multiple references to an allocated memory block. The function free() applied to the multicast memory block decrements a counter of the references, the actual memory free is done when the counter has reached a zero value.

Function call	Description
ptr=malloc(bytes)	Allocates a block specified by number of bytes.
new_ptr=realloc(ptr,bytes)	Reallocates the specified block according to the new size.
multicast(ptr)	Adds new reference to the block.
free(ptr)	Decrements the reference counter and/or frees the memory block.

Table 4.1. Function interface of the shared memory pool

The top structure of the shared memory pool contains a synchronization semaphore used to increment a round-robin bank selector in an exclusive mode. That is only one critical section of code the processes have to go through. However this section contains only an increment operation for the bank selector, so no serious bottleneck is present. Furthermore, each memory pool bank has its own synchronization semaphore, which ensures entry of only one process when the internal bank structure is being modified. This algorithm is illustrated below in a kind of pseudo-code.

```

Sync_semaphore.SEM_OP(-MAX_VAL); // enter pool critical section
RoundRobin_bank_selector=(RoundRobin_bank_selector+1)%SHMEM_POOL_BANKS;
bank_number=RoundRobin_bank_selector;
Sync_semaphore.SEM_OP(MAX_VAL); // leave pool critical section

bank_number.Sync_semaphore.SEM_OP(-MAX_VAL); // enter bank critical section
// ...
// perform memory allocation and modify the bank structure
// ...
bank_number.Sync_semaphore.SEM_OP(MAX_VAL); // leave bank critical section

```

Allocated blocks occupy memory space from lower addresses toward higher addresses. Each memory block has a corresponding entity descriptor containing a pointer to the allocated block, size of the block and the multicast reference counter. The heap of the entity descriptors grows from higher addresses to lower addresses until the memory space of the bank is exhausted. In our implementation we use a pointer to the first memory hole to speedup the allocation time for a new memory block. This additional pointer forces a search subroutine to start searching for a free space not from the beginning of a heap but from the first memory hole.

Synchronization semaphores are also operated in a read mode SEM_OP(-1)/SEM_OP(1) when the system collects a statistic information. A typical statistic report is shown below, from which we can recognize that the round-robin policy equally divides the load among the shared memory pool banks.

```

[MemPool]: ** STATUS OF THE SHARED MEMORY DRIVEN BY REENTERABLE CODE **
[MemPool]: Shared memory segment ID=4087.
[MemPool]: SHMEM_POOL_SIZE: 2147483648Bytes (10 BANK(S) of 214748352 each).
[MemPool]: Shared memory segment has been attached at 0x4000068000.
[MemPool]: Shared memory segment permissions are 0x01B4.
[MemPool]:<BANK#: Entities, FirstEntSpaceAfter, Free(Max), Fragmentation.>
[MemPool]: B#0: Ent=897, FA=798, Free=210477464B(210420120), Frag=0.03%.
[MemPool]: B#1: Ent=901, FA=797, Free=210358008B(210261568), Frag=0.05%.
[MemPool]: B#2: Ent=900, FA=805, Free=210385192B(210266224), Frag=0.06%.
[MemPool]: B#3: Ent=898, FA=792, Free=210148336B(210099184), Frag=0.02%.
[MemPool]: B#4: Ent=894, FA=790, Free=210321824B(210158824), Frag=0.08%.
[MemPool]: B#5: Ent=894, FA=811, Free=210782552B(210682200), Frag=0.05%.
[MemPool]: B#6: Ent=892, FA=794, Free=210493944B(210381120), Frag=0.05%.
[MemPool]: B#7: Ent=884, FA=794, Free=210630464B(210594624), Frag=0.02%.
[MemPool]: B#8: Ent=890, FA=797, Free=210582384B(210241864), Frag=0.16%.
[MemPool]: B#9: Ent=878, FA=794, Free=210404136B(210225776), Frag=0.08%.
[MemPool]: Memory Pool TOTAL:
[MemPool]: Number of entities allocated in the pool: 8928.
[MemPool]: Number of extra multicast references in the pool: 10.
[MemPool]: Free space in the pool: 2104584304Bytes.
[MemPool]: Largest free block in the pool: 210682200Bytes.
[MemPool]: Fragmentation of holes in the pool: 0.06%.

```

The proposed architecture of the shared memory pool is fully multithreaded and scalable for any number of running processes and shared memory pool banks. Generally, it performs well on all affordable SMP hardware we experimented with.

4.4. Non-Dead-Locking Policy

The problem of dead-locking is a very important issue for every parallel system that shares common resources. This section describes how we avoid this problem in our parallel architecture.

At first we create some prerequisite conditions during the static scheduling stage:

- The static scheduler checks for all uninitialized variables in a user application that are potential hazards for the dataflow engine. The uploading process will not start until all uninitialized states are fixed.

- The uploading process feeds the marshaled clusters into the dataflow engine in the sequence they would be executed single-threadedly. That prevents a situation where the dataflow resources are saturated with unresolved data waiting for dependencies from outside.

Secondly, the dataflow engine relies on a distributive semaphore allocation in the shared memory pool. Thus different shared resources do not use the same semaphores, which prevents dead-locking situation when a parallel processing is trying to block one resource keeping another resource blocked.

And finally, the dynamic scheduling subsystem uses one-way object locking policy shown in Figure 4.5.

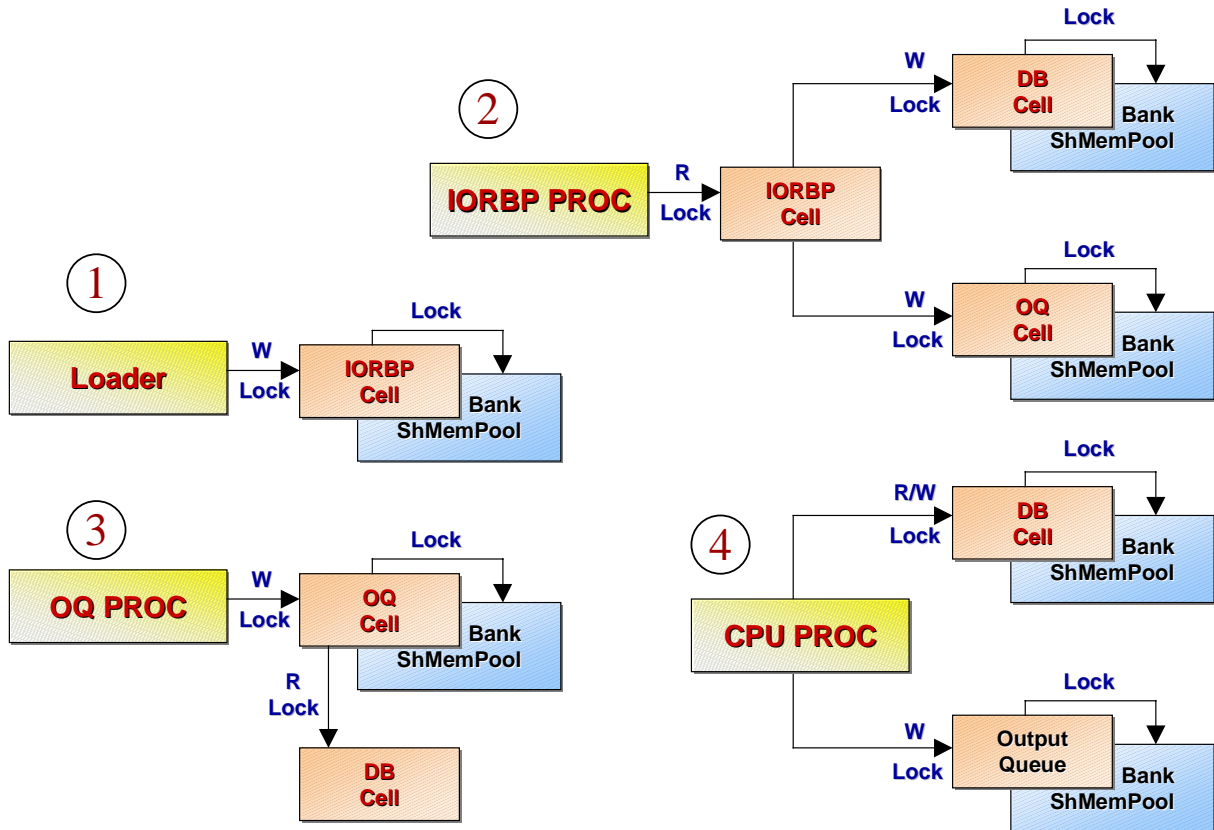


Figure 4.5. One-way object locking policy

There are four object locking algorithms of the dynamic scheduling. Obviously in each case we exclude a mutual blocking of resources:

1. The external task loader locks a cell in the I/O buffer ports for writing. If some stream data has to be allocated dynamically the shared memory pool bank is locked as well.
2. IORBP processes read the marshaled clusters from the I/O buffer cells locking the cell for reading. The cluster's data is targeted for the data buffer and the cluster's instructions are directed to the operation queue. Both types of cells are locked for writing. The shared memory bank could be locked for the streamed data allocation additionally.
3. OQ processes tag ready instructions in the operation queue. To tag the instruction it has to be locked for writing and all dependent operands, which are being checked in the data buffer, are locked for reading.
4. CPU processes execute ready instructions and put the results back to the data buffer or to the output queue. The output queue is locked for writing only. The DB cell is locked for reading when the required operands are being read and is locked for writing when the result is flushed to the DB. The shared memory pool bank is locked for writing.

4.5. Inter Process Communication

Three types of dynamic scheduling processes (IORBP PROC, OQ PROC and CPU PROC) communicate with each other through the communication channels. In the dataflow engine we use our own implementation of the

communication channels. We do not use the standard SVR4 IPC message queues on account of the following considerations:

- The standard SVR4 IPC message queues are restricted in size, maximal number of messages and structure of the message. These limits come from operating system configuration parameters.
- The message queues are too redundant for our needs. For example, the messages can be retrieved from the queue randomly by their reference numbers, which is not necessary for us.
- Extra functionality of the message queues predefines an approximate slowdown factor of 10 in comparison with the fastest communication through the shared memory as our experiments with TAU [147, 148] have shown.

Our implementation of the communication channels is a classic ring buffer structure in the shared memory synchronized by semaphores. As shown in Figure 4.6, two types of communication channels are used: unprotected write protected read (UWPR) and protected write protected read (PWPR).

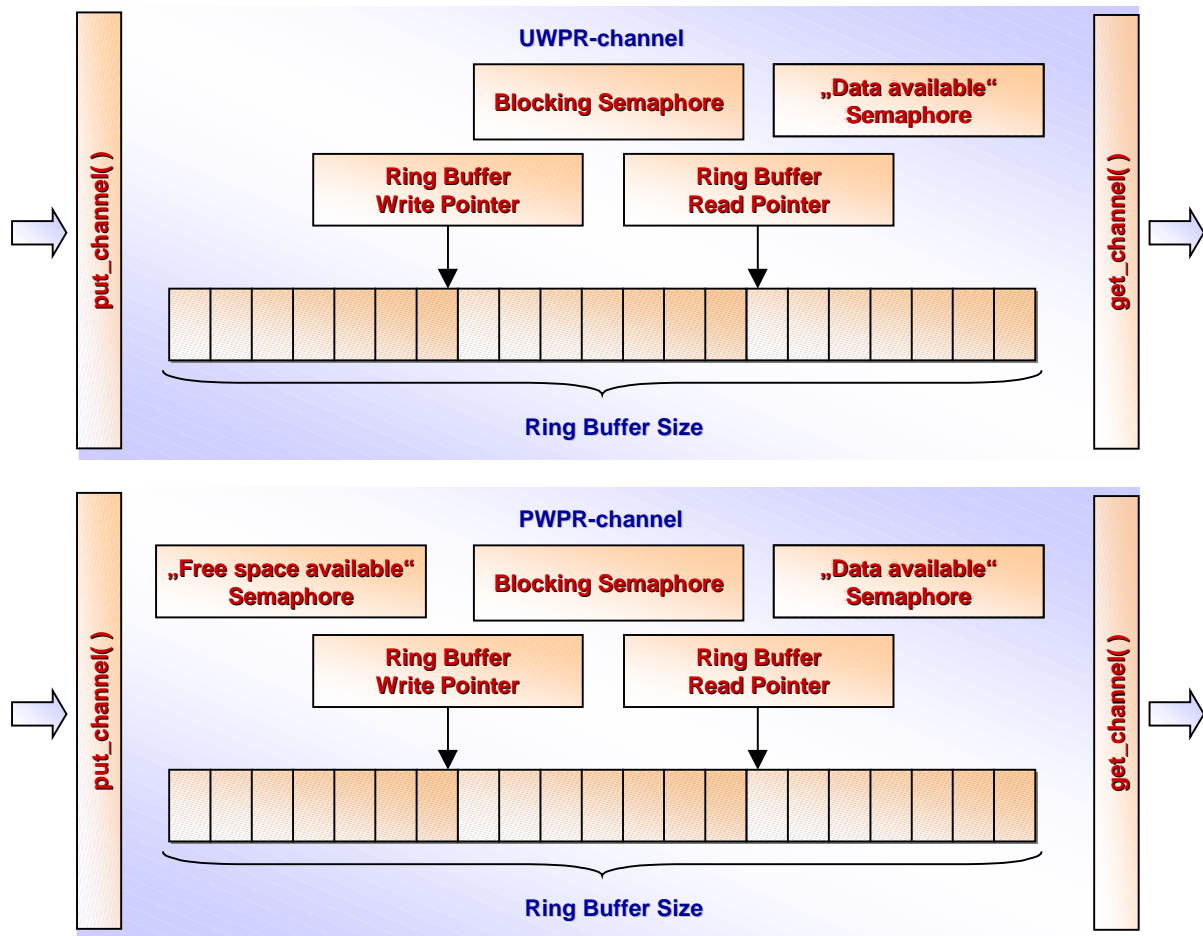


Figure 4.6. UWPR and PWPR channels

Both communication channel types use ring buffer pointers for read and write and the synchronization semaphores protecting the internal data structures. Because all communication channels are used in a predefined functionality (Figure 4.7) we omit write semaphore locking in most cases, reducing write synchronization efforts nearly to zero.

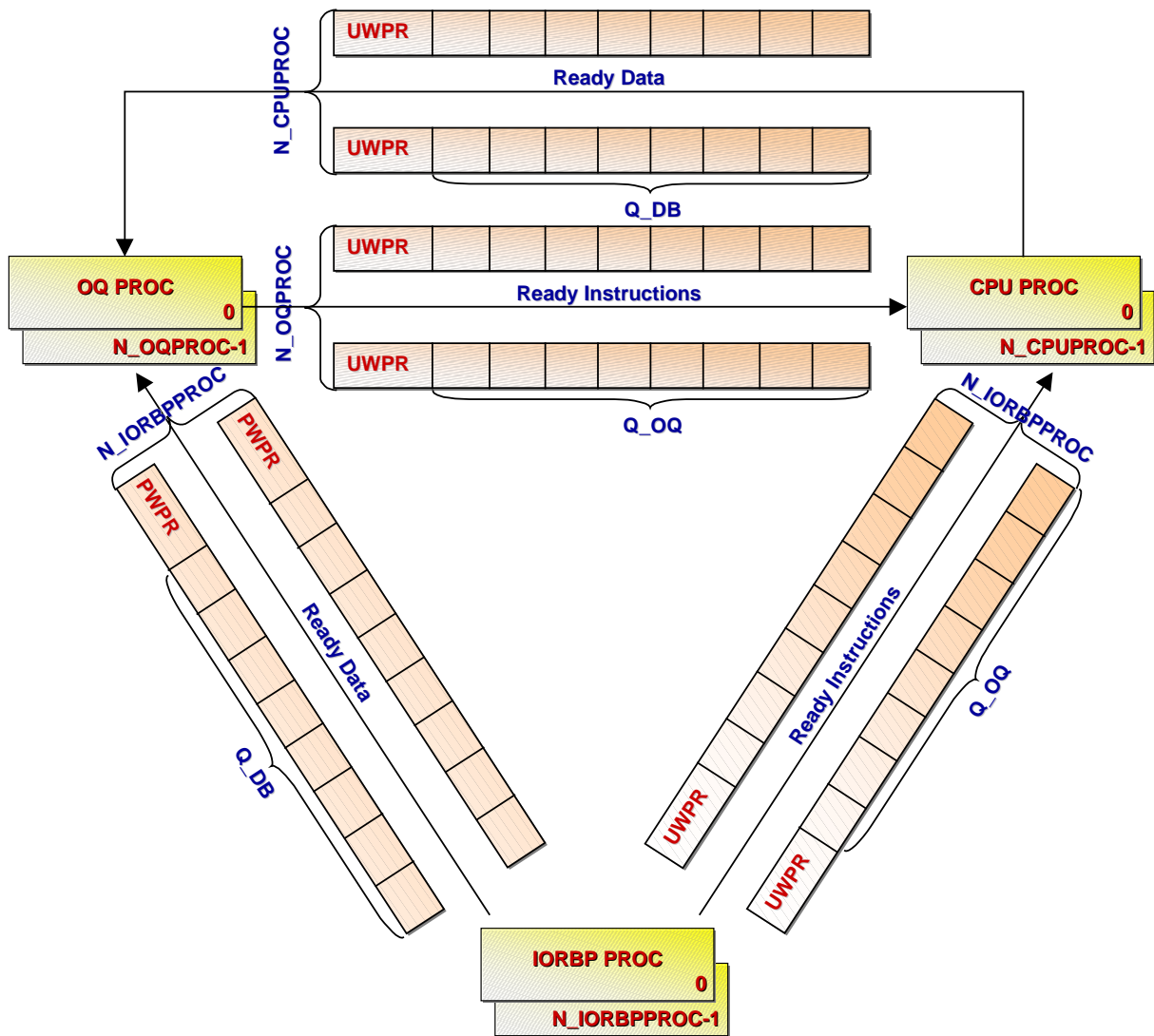


Figure 4.7. Communication between the scheduling processes

The BMDFM dynamic scheduling subsystem uses four communication directions: OQ PROCs to CPU PROCs, IORBP PROCs to CPU PROCs, CPU PROCs to OQ PROCs and IORBP PROCs to OQ PROCs.

OQ PROCs to CPU PROCs communication. This group of channels delivers addresses of the ready instructions tagged by the OQ PROCs. The size of each channel is equal to the size of the operation queue, so no data overlap is possible. Therefore we use the UWPR type of channel, which does not require checking of the “Free space available” semaphore.

IORBP PROCs to CPU PROCs communication. As in the previous case the purpose of these channels is to deliver addresses of ready instructions with the difference that they are tagged by the IORBP PROCs. This could happen if the instructions have all operands ready at the moment of relocating to the operation queue. Here we also use UWPR channels as no write synchronization is needed. The CPU PROCs listen to both groups of channels retrieving the addresses of ready instructions tagged by OQ and IORBP PROCs. Unfortunately, there is no chance to reduce synchronization efforts for reading.

CPU PROCs to OQ PROCs communication. These channels transfer addresses of the ready operands after they have been processed by the CPU PROCs. To avoid write synchronization the size of the channels is set to size of the DB. The UWPR channel type is applicable here as well.

IORBP PROCs to OQ PROCs communication. This group of channels delivers addresses of the ready operands put into the I/O ring buffer from outside. Practically, it is possible that the external task will provide ready data in the multiple contexts, so the total amount of data can exceed the size of DB. Therefore the IORBP PROCs to OQ PROCs communication is the only place where PWPR channels are used. The OQ PROCs listen to both groups of channels retrieving the addresses of ready data tagged by the CPU and IORBP PROCs.

The following pseudo-code demonstrates schematically how the channel write/read operations are implemented. Each process uses its own dedicated write channel that allows skipping the blocking lock while incrementing the ring buffer write pointer.

Initial values for the channel semaphores are set as follows:

- All blocking semaphores are set initially to one, permitting any incoming operations.
- “Data available” semaphores store the number of unread data inside the channel, initial zero values indicate that all channels are empty.
- “Free space available” semaphores initially are set to the size of the channel signaling write enable state.

```

put_channel(address){
    "Free_space_available".SEM_OP(-1); // only for PWPR channels

    ring_buffer[write_pointer]=address;
    write_pointer=(write_pointer+1)%buffer_size;

    "Data_available".SEM_OP(1);
}

get_channel(){
    "Data_available".SEM_OP(-1);

    Blocking_semaphore.SEM_OP(-1);
    address=ring_buffer[read_pointer];
    read_pointer=(read_pointer+1)%buffer_size;
    Blocking_semaphore.SEM_OP(1);

    "Free_space_available".SEM_OP(1); // only for PWPR channels

    return address;
}

```

As we can see there is no synchronization effort to write to the UWPR channel and minimal efforts to write to the PWPR channel. Having used the UWPR channel type in 80% of the cases we can state that we have an optimal solution when writing to the channel. Unfortunately, we have two synchronization points when reading from the channel.

4.6. Task Connection Zone

The task connection zone (TCZ) provides an interface to the multithreaded dataflow engine (Figure 4.8). TCZ can be configured for N_IORBP connection sockets defining the number of simultaneously connected external task loader/listener pairs (user applications). A user application is uploaded into the dedicated area of I/O ring buffer ports (IORBP) according to the round-robin policy. Further IORBP PROC processes fetch this information multithreadedly scanning all IORBP.

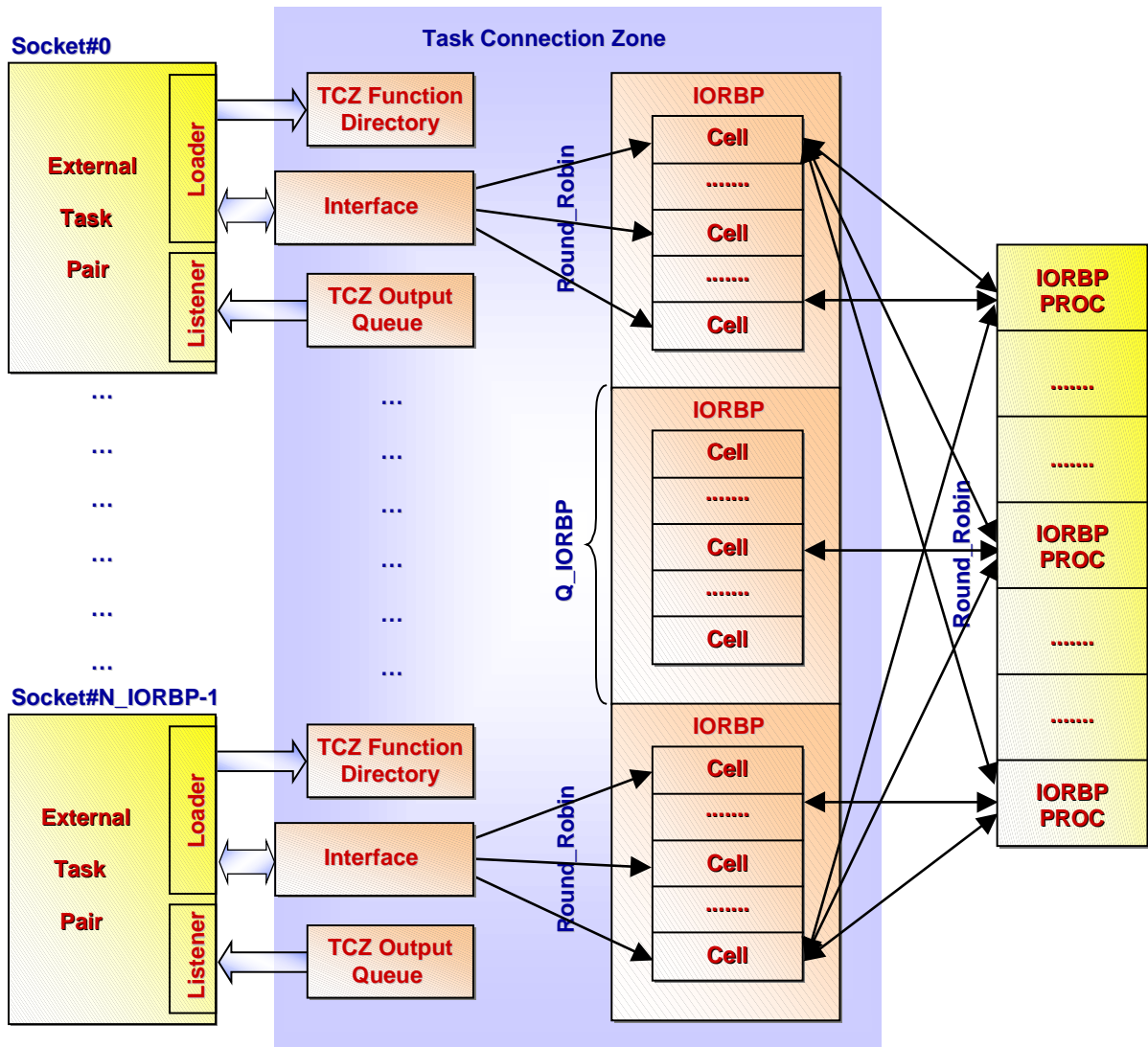


Figure 4.8. Task connection zone

Each connected user application (job) is associated with a session and communicates through a dedicated interface. Initially, the user application uploads all seamless byte code fragments into the TCZ function directory and all referenced variables to the data buffer (DB). Then during runtime the application uploads marshaled clusters, which contain data and instructions in a form of references to the DB and TCZ function directory, respectively. The application can also ask for some data that influences the uploading sequence. All these calls are performed through the dedicated interface described in Table 4.2. Such an uploading scheme has the following advantages:

- The marshaled clusters and seamless byte code fragments are prepared during the preprocessing and compilation stages. Thus no dynamic marshaling overhead is required.
- The size of the marshaled clusters and uploading traffic are considerably reduced because instead of the byte code fragments only references to the TCZ function directory are transferred.
- Space reservations in the DB for all application variables are done only once when the external connection is initialized, completely excluding associative searches at runtime.

Function call	Description
dfminit_upload(var_lst, fnc_lst);	Initializes a new session, allocates variables in DB, uploads seamless byte code fragments into the TCZ function directory.
dfmend_session();	Informs listener that the last chunk of code is uploaded, so the listener can wait until all data are received from the dataflow engine

	and finish.
dfmclose_session();	Waits for the listener. When the listener is done purges the occupied TCZ socket and closes the session.
dfmput_marshaled_cluster(cluster);	Delivers a marshaled cluster.
dfmput_idata(ivar);	Puts a variable with new context. Variable is initialized with an integer value.
dfmput_fdata(fvar);	Puts a variable with new context. Variable is initialized with a float value.
dfmput_sdata(svar);	Puts a variable with new context. Variable is initialized with a string value.
dfmput_zdata(zvar);	Puts a variable with the current context. Variable is not initialized (nil value) but in this way the variable (with latest context in DB) can receive destination attributes indicating that value is requested by the loader or listener.
dfmput_crelease(contexts);	Informs garbage collector about obsolete contexts. Those are marked for deletion.
dfmget_idata(var);	Loader requests an integer variable value and receives it when it is ready.
dfmget_sdata(var);	Loader requests a string variable value and receives it when it is ready. The loader can request only integer and string data that is sufficient to control the uploading sequence.

Table 4.2. Interface to the multithreaded dataflow engine

The last but also very important part of the task connection zone is a TCZ output queue shown in more detail in Figure 4.9. The purpose of the TCZ output queue is to have a kind of buffer where output data stream can be ordered after the out-of-order dataflow processing.

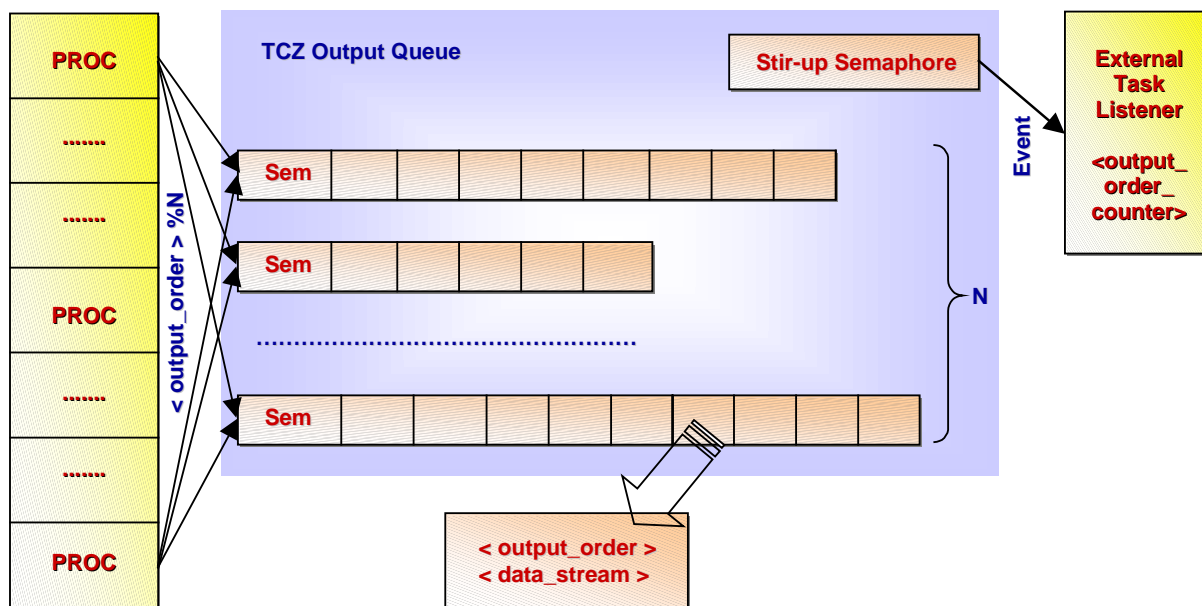


Figure 4.9. TCZ output queue

Normally, the output stream is a result of processing done by CPU PROCs. Because the data is processed on dataflow the results can be computed in an unpredictable order. To help making them ordered in the output stream all output data are accompanied by an output_order attribute. This attribute begins its life cycle from the marshaled cluster, then it is stored in the DB together with the variable value. As soon the output value becomes ready it is sent into the TCZ output queue.

The TCZ output queue is organized in a row manner. Each row has its own synchronization semaphore for the case of conflict when multiple parallel processes try to hold the row. CPU PROC selects a row, performing a trivial modulo operation $output_order \% N$, where N is a number of rows. The computed result is put into a free

cell of the row together with the output_order attribute firing the stir-up semaphore. To prevent redundant copying of the resulting stream the multicast() function of the shared memory pool is called. The external task listener waits when the stir-up semaphore is fired and fetches the data. Two fragments of code running by the CPU/IORBP PROC and the external listener, respectively, are shown below:

```

push_output_queue(DB_addr){ // CPU PROC or IORBP PROC

    row=DB[DB_addr].output_order%N;

    sem[row].SEM_OP(-MAX_VAL);
    put_row(row,multicast(DB[DB_addr].data));
    sem[row].SEM_OP(MAX_VAL);

    stir-up_semaphore.SEM_OP(1);
}

pull_output_queue(){ // External listener

    stir-up_semaphore.SEM_OP(-1);

    read_next=true;
    while(read_next){

        row=output_order_counter%N;

        sem[row].SEM_OP(-MAX_VAL);

        if(data=found_in_row(row,output_order_counter){
            stream.concatenate(data);
            output_order_counter++;
        }
        else
            read_next=false;

        sem[row].SEM_OP(MAX_VAL);
    }

    return stream;
}

```

4.7. I/O Ring Buffer Ports

I/O Ring Buffer Ports (IORBP) is an array of IORBP cells located in the task connection zone. The structure of the IORBP cell is shown in Figure 4.10. The IORBP cell contents is initialized by the external loader and further is fetched by IORPB PROC.

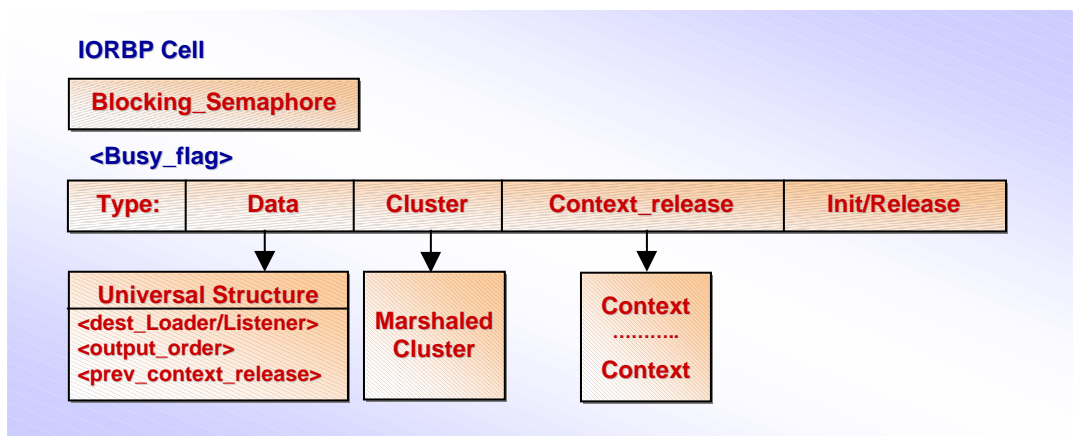


Figure 4.10. IORBP cell

Naturally, the IORBP cell has a blocking semaphore to protect the internal structure when it is shared by the parallel processes. A busy flag indicates that the contents is loaded and is valid otherwise the cell is considered to be free. Depending on the information type the cell can contain a marshaled cluster, variable value arranged in

the universal structure, a list of contexts to be released, or indication of session begin/close (Init/Release) associated with the connected user job. The universal structure is accompanied by the following “destination attributes”:

- <Dest_Loader/Listener> is a flag that can be set if the value is needed for the loader to control the uploading sequence or for the listener to be included into the output stream.
- <Output_order> is valid only if the <dest_Loader/Listener> flag is set for the listener. In this case the <output_order> means the order, in which data has to appear in the output stream.
- <Prev_context_release> points to the previous context number that automatically will be marked as obsolete for the garbage collector. If no active links to the obsolete context exist the context is removed from the dataflow engine.

Figure 4.11 shows the structure of the marshaled cluster. The marshaled cluster contains a group of functions (local function directory) and referenced variables (local variable directory). Each function is a reference to the TCZ function directory, thus it is a number associated with the seamless byte code fragment. As in case of the universal structure each function has “destination attributes” attached. Context variables from the local directory are targeted to the DB and the functions will be moved to the operation queue (OQ). Each function inherits context from its destination variable, the other way around each destination variable inherits the “destination attributes” from its function-producer. This is done to make the marshaled cluster structure maximally compact as the clusters are transferred and decoded at runtime.

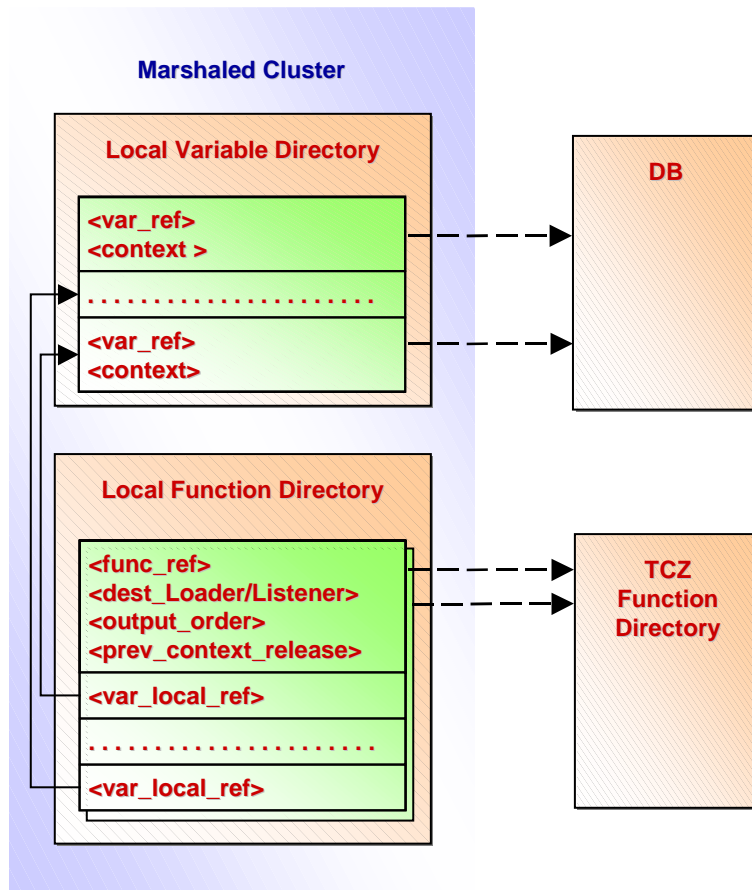


Figure 4.11. Structure of marshaled cluster

We think that this section is an appropriate place to describe the allocation algorithm used by the external loader occupying free IORBP cell. Because the same strategy is applied when finding free DB and OQ cells we describe the algorithm in a more common way, showing allocation of a shared cell in an abstract array:

```

ArrayFreeCells_semaphore.SEM_OP(-1); // synchronization, free cell exist,
while(true){                          // endless allocation loop is safe.

```

```

array_ptr=(array_ptr+1)%array_size; // Round-Robin iterate.
if(!array[array_ptr].busy){ // first speculative check.
    array[array_ptr].SEM_OP(-MAX_VAL); // lock the cell for modification.
    if(!array[array_ptr].busy){ // second expensive safe check.
        array[array_ptr].busy=true; // mark as occupied.
        break; // leave the allocation loop.
    }
    array[array_ptr].SEM_OP(MAX_VAL); // unlock the cell.
}
}
fill_contents(); // fill the data.
array[array_ptr].SEM_OP(MAX_VAL); // unlock the cell.

```

The important feature of the proposed algorithm is a speculative check that is done omitting a semaphore lock. Only if the first speculative check has passed the second expensive safe check will make a final decision. Taking into account that the semaphore locking is in average 100 times slower than a simple conditional, the proposed algorithm performs much faster as our experiments with TAU [147, 148] have shown.

4.8. Data Buffer

Data Buffer (DB) is an array of DB cells located in the shared memory pool. The structure of the DB cell, which is intended to store multiple contexts of a single variable, is shown in Figure 4.12. The DB cell contents is initialized by the IORBP PROCs and further is used as the storage for all active contexts of a variable value.

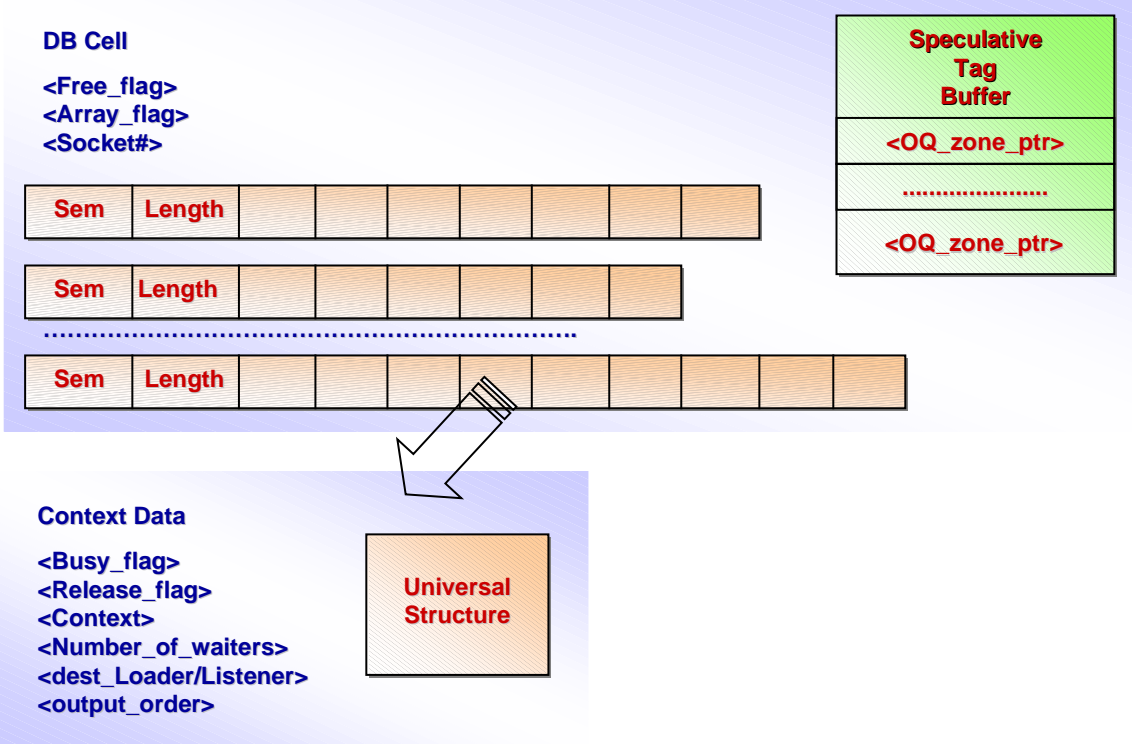


Figure 4.12. DB cell

According to the tagged-token dataflow every new variable assignment creates a copy of the variable in a new unique context. The previous contexts live until they are referenced by the unprocessed instructions. Later a garbage collector removes them from memory. Because multiple contexts can be processed at the same time the

DB cell structure is organized as an array of rows containing context data. Each row has a blocking semaphore to control data sharing. In addition to the main data storage the DB cell assists in a speculative tagging mechanism. The speculative tag buffer is modified every time a new context is introduced into the DB cell. OQ zone pointers are tagged for those zones where context dependent instructions are located. When the variable value is computed and turned into a ready state the instructions from the tagged OQ zones are checked for readiness.

All other components of the DB cell are explained below:

- `<Free_flag>` tags the DB cell as free. Reservation of the cells for a user application is done once during the initialization phase `dfm_init_upload()`.
- `<Array_flag>` is set when variable itself is an array. This defines a method how the universal structure has to be checked whether it is ready or not. The problem can appear if a CPU PROC tries to execute an instruction containing an indirect addressing. In this case the array index is computed at runtime and can reference the array member, which is not ready.
- `<Socket#>` is a task connection zone (TCZ) socket number that defines the user application this data belongs to. The socket number associates input/output streams of user application, it is also used as a filter to purge the socket of an interrupted application.
- `<Busy_flag>` marks the context data as busy. These flags are checked when a new context is being allocated under the same row.
- `<Release_flag>` means that context data can be removed from memory, as soon it is not referenced by any instruction (`<Number_of_waiters>` is equal to zero).
- `<Context>` is a unique number. This number is checked when a new context is being allocated under the same row.
- `<Number_of_waiters>` is the number of the context dependent instructions. A non-zero value means that the context is still in use.
- `<Dest Loader/Listener>` and `<output_order>` are the “destination attributes” defining a correct appearance of the results in the output stream.

When a new context is allocated and/or checked, the row and the locking semaphore is calculated by a simple modulo operation; an integer division defines the starting point for the search in the row (Figure 4.13.). Such a multiple context data structuring has two advantages:

- It allows access to neighbouring contexts independently without blocking.
- It has negligible dynamic scheduling overhead while allocating and searching for the context data.

DB Cell

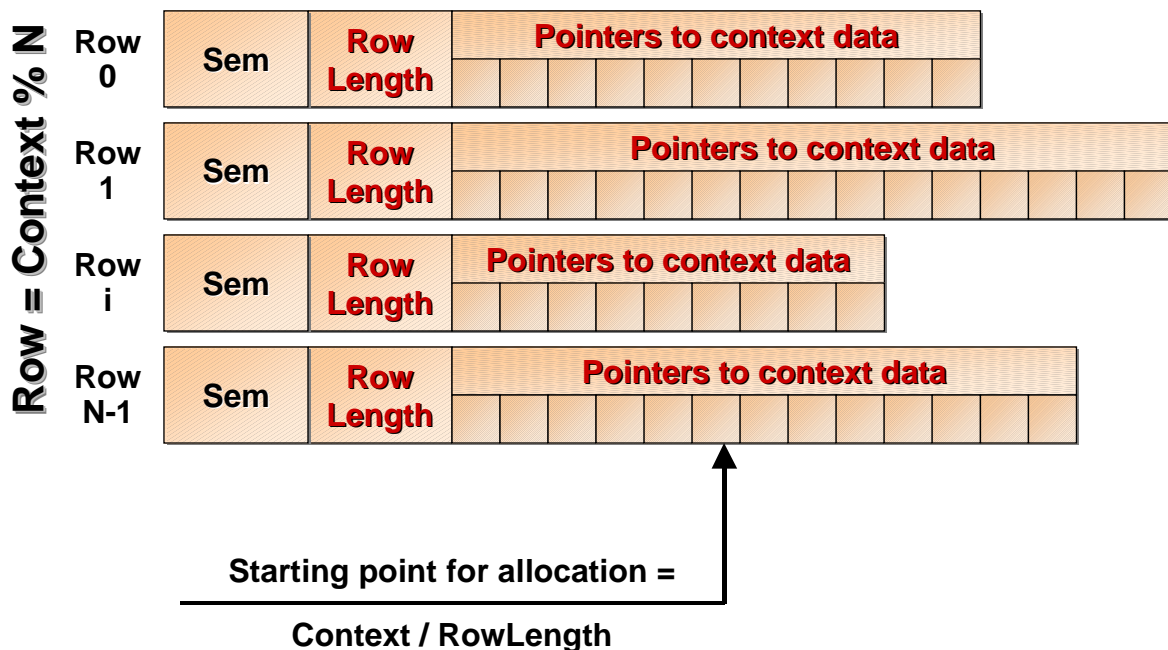


Figure 4.13. Multiple context data structuring

The following pseudo-code fragment illustrates an implementation of shared access to the context data:

```

row=Context%N; // select the row.
row.SEM_OP(-MAX_VAL); // lock the row.

posInRow=Context/RowLength; // starting point in the row.
found=false;

for(I=0;i<RowLength;i++){ // check if context already exists

    ContextData=row[posInRow];

    if(ContextData.Busy_flag && ContextData.Context==Context){
        found=true;
        break;
    }

    posInRow=(posInRow+1)%RowLength; // Round-Robin
}

if(!found){

    if(NoFreeSpaceInRow) // expand in chunks.
        RowLength=expandRow(ARRAYBLOCK_SIZE); // RowLength+=ARRAYBLOCK_SIZE.

    posInRow=Context/RowLength; // starting point in the row.

    while(1){ // reservation for new context.

        ContextData=row[posInRow];

        if(!ContextData.Busy_flag)
            break;

        posInRow=(posInRow+1)%RowLength; // Round-Robin
    }

    ContextData.Busy_flag=1; // now new context
    ContextData.Context=Context; // is placed to the row.
}

// ...
// modify the context data
// ...

row.SEM_OP(MAX_VAL); // unlock the row

```

At first we search whether the context is already in the row, assuming that its most probable position in the row is Context divided by RowLength. In case the context is not detected we reserve a new location for it, expanding the row if necessary. All shared memory dynamic structures expand in chunks, which makes shared memory reallocation not so intensive. The starting point for the new context in the row is also computed as the context divided by row length. In this way we increase probability of our assumption regarding starting position for all further searches.

4.9. Operation Queue

The Operation Queue (OQ) is an array of OQ cells located in the shared memory pool. The structure of the OQ cell is shown in Figure 4.14. The IORBP PROC daemons store the instructions in the OQ, then the OQ PROC daemons tag them as ready if all required operands are ready. Ready instructions are fetched and executed by the CPU PROCs. Physically, each instruction is a reference to one seamless byte code fragment from the TCZ function directory.

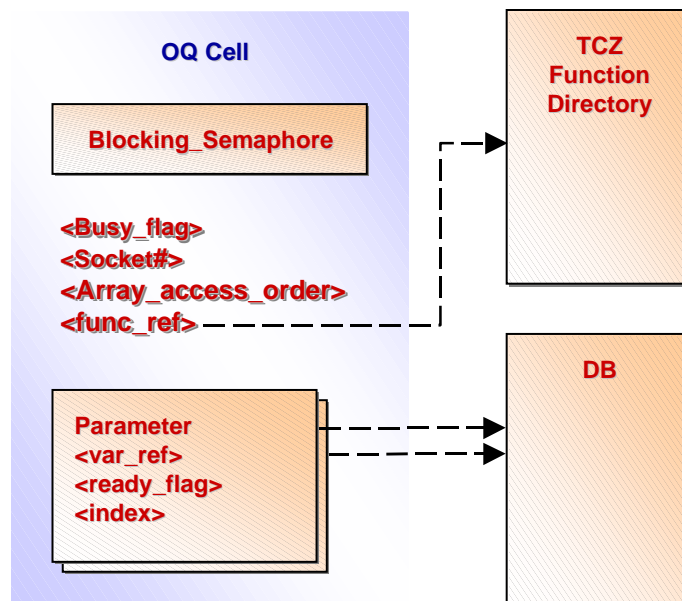


Figure 4.14. OQ cell

All other components of the OQ cell are explained below:

- <Busy_flag> is set if the OQ cell currently contains an active instruction. This flag is dropped only when the instruction is successfully executed.
- <Socket#> defines to which user application the instruction belongs. This is important when the BMDFM system runs many applications on one dataflow engine.
- <Array_access_order> is set when the instruction contains indirect addressing. In this case the array indices are computed at runtime to reference concrete array members.
- <var_ref> references the instruction's operands in the DB. This reference is already resolved and points directly to the context data in the DB cell.
- <ready_flag> marks each operand whether it is ready or not. The OQ PROC daemons check only for unready operands. The instruction is considered to be ready if all operands are ready.
- <index> specifies the array index additionally to the <var_ref>. Initially, the index is set to zero for all array operands. In a first approach such an operand is considered to be ready if zero member of the array is ready. A real array index is computed at runtime and can reference the array member, which is not ready. In this case first execution fails, the operand is reset to unready state but this time with the real index. Such a multi-pass checking scheme is applied to all indirect addressing instructions.

4.10. IORBP Scheduling Process

The IORBP PROC daemon runs an endless loop, in which all busy IORBP cells are analyzed for their types. Depending on the cell type four different actions are taken, respectively: session initialization, session deactivation, data allocation, processing of marshaled cluster and context release. The loop is controlled by semaphore number 6, which stores the number of busy cells. After the cell is processed it is relocked in writing mode to drop the <Busy_flag>. Note that relocking (SEM_OP(-MAX_VAL); SEM_OP(MAX_VAL-1); ... SEM_OP(1-MAX_VAL);) is a dead-lock safe form of (SEM_OP(-1); ... SEM_OP(1-MAX_VAL);) semaphore operation:

```
while(1){
    semaphore6.SEM_OP(-1);      // number of used entities in IORBPs
    iorbp=find_Busy();

    iorbp.sem.SEM_OP(-MAX_VAL); // lock cell for writing
    iorbp.sem.SEM_OP(MAX_VAL-1); // re-lock cell for reading
    switch(iorbp.Type){
        case Init:
            Socket#=dfmunit_upload(var_lst, fnc_lst);
    }
}
```

```

        break;
    case Release:
        dfmclose_session(Socket#);
        break;
    case Data:
        dfmput_data(var);
        break;
    case Cluster:
        dfmput_marshaled_cluster(cluster);
        break;
    case Context_release:
        dfmput_crelease(contexts);
    }

iorbp.sem.SEM_OP(1-MAX_VAL); // re-lock cell for writing
    Busy_flag=false;
iorbp.sem.SEM_OP(MAX_VAL); // unlock cell
}

```

Session initialization allocates application's variables in DB, uploads seamless byte code fragments into the TCZ function directory and registers user application in TCZ.

```

dfminit_upload(var_lst, fnc_lst){
    allocate_variables(var_lst);

    upload_TCZ_function_directory(fnc_lst);

    semaphore4.SEM_OP(-MAX_VAL); // lock Task Connection Zone for writing
        Socket#=register_application();
    semaphore4.SEM_OP(MAX_VAL); // unlock Task Connection Zone

    return Socket#;
}

```

Session deactivation similarly to the session initialization sequence purges DB and TCZ function directory and then releases the socket.

```

dfmclose_session(Socket#){
    purge_variables();

    purge_TCZ_function_directory();

    semaphore4.SEM_OP(-MAX_VAL); // lock Task Connection Zone for writing
        release_socket(Socket#);
    semaphore4.SEM_OP(MAX_VAL); // unlock Task Connection Zone
}

```

Data allocation delivers application's data to DB and performs several auxiliary procedures. Variable var is taken from the IORBP cell and moved to the context data of DB cell. The function call access_context_data() modifies an existing context or creates a new one from the scratch.

If context data is ready the OQ PROC is informed through the communication channel to check the dependent instructions. To reduce the number of speculative checks we use a restriction semaphore structure. It is probable that a request to check the current DB speculative tag buffer was already sent to one of the channels. In this case the restriction semaphore number DB_addr%size_of_restriction_structure is locked. If some of the OQ PROCs are currently checking all tags from the DB speculative tag buffer the restriction semaphore is unlocked again. Having experimented on TAU [147, 148], we estimate significant (10 times) relief in OQ PROCs load when using such a protection mechanism.

Ready data could also be requested by the loader or listener. After that the IORBP PROC performs garbage collection. If current context data is marked for deleting (Release_flag is true) and there are no dependent instructions (Number_of_waiters is zero) then the context is removed from memory. The same garbage collection is applied to the previous context data if specified in prev_context_release.

```

dfmput_data(var){

```



```

instruction.parameter[var_ref].index=0;
}
if(instruction.Ready)
    put_channel(instruction);           // IORBP to CPU communication
if(func_ref.prev_context_release){
    ContextData=access_context_data(prev); // previous context data
    ContextData.Release_flag=true;
    if(!ContextData.Number_of_waiters)
        ContextData.remove();           // remove from memory
}
}
}
}

```

Context release is a pure garbage collection procedure. It is called by the external loader after finishing a UDF uploading sequence. The contexts are collected in a list of all UDF's local variables to be removed from memory. If there are no dependent instructions then the contexts are removed.

```

dfmput_crelease(contexts){
    foreach context in contexts{           // iterate all contexts.
        ContextData=access_context_data(context); // context data in DB cell
        ContextData.Release_flag=true;
        if(!ContextData.Number_of_waiters)
            ContextData.remove();           // remove from memory
    }
}
}

```

4.11. OQ Scheduling Process

One of the serious dataflow problems is the dynamic scheduling overhead caused by dynamic operand matching and tagging. The BMDFM exploits a speculative tagging mechanism of the instructions as shown in Figure 4.15. Each data buffer (DB) cell additionally stores a limited number of tags to the instructions in the operation queue (OQ). Each tag speculatively represents several OQ cells where dependent instructions might be stored. Having experimented on TAU [147, 148], we estimate an optimal size for the DB cell speculative tag buffer, which is equal to $2*\sqrt{Q_DB}$. The OQ PROC interleaved search loop is triggered when a variable's data becomes ready in DB. The implemented mechanism provides the following benefits:

- The DB cell speculative tag buffer has a fixed size and will not be dynamically reallocated.
- The DB cell speculative tag buffer does not require any semaphore synchronization.
- The reduced size of the DB cell speculative tag buffer saves memory space and scanning time.
- Speculative checking loops do not block each other, detect required tagging with reasonable probability and keep tagging latency low.

**DB Cell
Speculative
Tag Buffer
<variable a>**

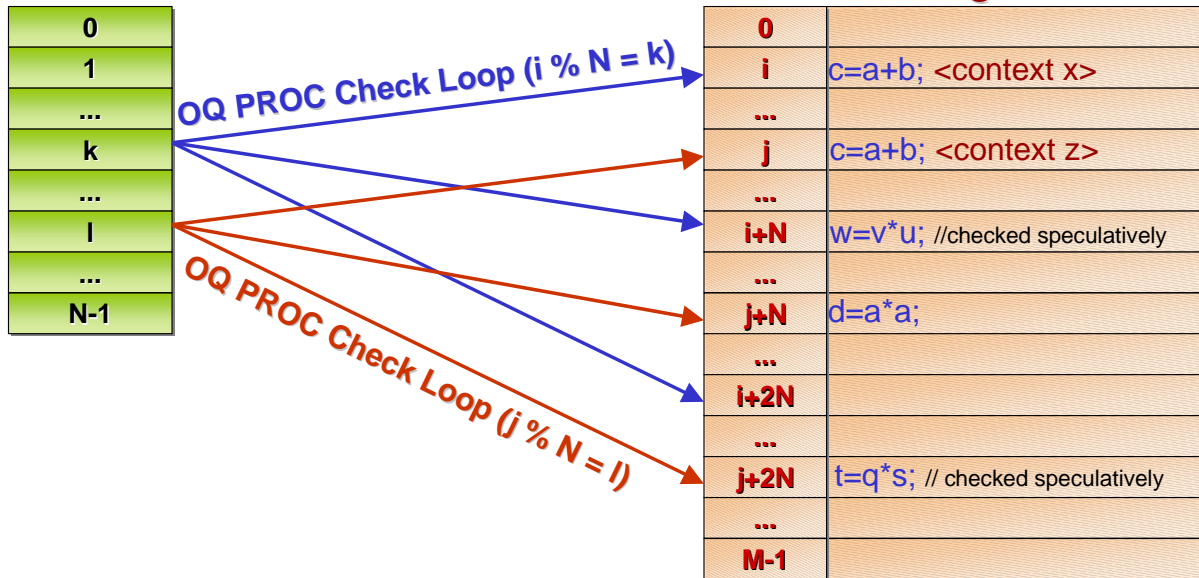


Figure 4.15. Speculative tagging of instructions

Each of the OQ PROC daemons runs an endless loop, listening to the IORBP and CPU communication channels. As soon the DB cell number with new ready data is received the speculative tagging algorithm is forced.

At first we reset the restriction semaphore structure to allow sending new messages to the communication channels. Then the DB cell speculative tag buffer is scanned for all fired tags. Each tag represents only a limited number of OQ cells to be checked, but they are interleaved along the OQ. The second nested loop performs this interleaved selection.

```

while(1){
    DB_cell=get_channel();           // listen to all IORBP and CPU channels.
    reset_Restriction_Semaphore_Structure(); // instructions are being checked
    // scan DB_cell speculative tag buffer
    for(i=0;i<DB_cell.speculative_tag_buffer_size;i++){
        if(DB_cell.speculative_tag_buffer[i]){
            DB_cell.speculative_tag_buffer[i]=0; // instructions are being checked
            // check speculatively across the OQ
            for(j=i;j<Q_OQ;j+=DB_cell.speculative_tag_buffer_size)
                check_instruction(OQ[j]);
        }
    }
}

```

The check_instruction() procedure itself is shown in the following code fragment. An instruction check is necessary only if the instruction is present in the OQ cell and it is not ready yet. We check it twice: the first speculative check significantly reduces the number of semaphore operations. Initially, we assume the instruction is ready, then this assumption is checked for each unready function's parameter. Finally, the instruction is tagged as ready if all operands are ready. If the check fails next time the ready operands will not be checked again. The ready instructions are sent to the CPU PROCs for execution through the communication channel.

```

check_instruction(instruction){

```

```

if(instruction.Busy_flag && !instruction.Ready){ // first speculative check
    instruction.sem.SEM_OP(-MAX_VAL); // lock OQ cell for writing.
    if(instruction.Busy_flag && !instruction.Ready){ // second safe check.
        instruction.Ready=true; // assume the instruction is ready
        foreach parameter in instruction.parameters
            if(!parameter.ready_flag){
                ContextData=access_context_data(parameter); // context data
                // instruction is ready if all operands are ready
                instruction.Ready&= // (Non-short-circuit evaluation)
                    (parameter.ready_flag=ContextData.Ready);
            }
        }
        instruction.sem.SEM_OP(MAX_VAL); // unlock OQ cell
        if(instruction.Ready)
            put_channel(instruction); // OQ to CPU communication
    }
}

```

4.12. CPU Executing/Scheduling Process

The CPU PROC daemons execute the instructions tagged as ready performing some scheduling task in addition. In the main loop an instruction is obtained from the communication channel and executed on a modified kernel of the virtual machine. In contrast to a standard kernel the modified kernel communicates directly with the shared memory pool. For that only three modifications are done:

- (get_var ...) and ([al]index ...) functions take variable values from the shared memory pool.
- ([al]setq ...) and (arsetq ...) assignment functions store variable values in the shared memory pool performing the additional scheduling.
- (asyncheap_ ...) function group allocates memory in the shared memory pool.

```

while(1){
    instruction=get_channel(); // listen to all IORBP and OQ channels.
    modified_VM.execute(instruction); // execute instruction
}

```

(get_var ...) and ([al]index ...) functions use instruction's parameter to refer to the context data in DB cell directly. The context data is locked for reading while copying them from the shared memory pool. In case the array is accessed with an unready member the parameter is reset back to the unready state and the current instruction is cancelled. No rollback actions are needed in the shared memory pool because the instruction works with local copies of the operands.

```

get_var__index(parameter,index){
    operand=parameters[parameter].var_ref;
    ContextData=DB[operand];
    ContextData.sem.SEM_OP(-1); // lock context data for reading
    if(ContextData.Ready){
        ContextData.sem.SEM_OP(1); // unlock context data.
        return copy_flp_data(); // return a copy of context data
    }
}

```

```

    }

    ContextData.sem.SEM_OP(1);          // unlock context data.

    parameters[parameter].Ready_flag=false; // reset ready parameter.
    parameters[parameter].index=index;     // set real index.

    throw exception(notReady);          // cancel instruction execution
}

```

([al]setq ...) and (arsetq ...) assignment functions also use instruction's parameter to refer to the context data in the DB cell directly. The context data is locked for writing while modifying them in the shared memory pool. After the context data are copied the following scheduling sequence is executed:

1. Data could be delivered to the loader or listener if requested.
2. The OQ PROC is informed through the communication channel to check the dependent instructions and tag them as ready. To reduce the number of speculative checks we use the restriction semaphore structure again.
3. For all parameters the number of dependencies is decremented by one (waiters--), immediately followed by an attempt to apply the garbage collection sequence to them.
4. Finally, the executed instruction is removed from the OQ, freeing space for other instructions.

```

setq_arsetq(parameter, index){

    operand=parameters[parameter].var_ref;
    ContextData=DB[operand];

    ContextData.sem.SEM_OP(-MAX_VAL); // lock context data for writing.
    ContextData=copy_flp_data();     // copy context data.
    ContextData.sem.SEM_OP(MAX_VAL);  // unlock context data

    switch(var.dest_Loader/Listener){
        case Loader:
            loader.dfmget_data(ContextData); // destination=loader
        case Listener:
            push_output_queue(ContextData, output_order); // listener output_queue
    }

    if(!check_Restriction_Semaphore_Structure()) // instructions are being
                                                    // already checked.
        put_channel(ContextData); // CPU to OQ communication.

    foreach parameter in parameters{ // iterate all parameters.

        ContextData=access_context_data(parameter); // context data in DB cell.

        ContextData.Number_of_waiters--; // number of dependencies

        if(ContextData.Release_flag && !ContextData.Number_of_waiters)
            ContextData.remove(); // remove from memory

    }

    instruction.remove(); // remove from memory

}

```

4.13. Complexity of the Dynamic Scheduling Subsystem

Many different methodologies exist, which help to estimate program complexity [77, 106]. Some of them recommend to count the code lines, others are based on calculations of conditional branches and function calls. According to our approach we think that synchronization points are most important for the dynamic scheduling subsystem.

Thus we estimate the complexity of the dynamic scheduler by the number of semaphore decrement operations where the process can be suspended into the idle state. The BMDFM system can be compiled in a debug mode (`_DEBUG_MODE_`), in which every synchronization point assigns the state number before entering a critical code section.

```

...
#ifdef _DEBUG_MODE_
    *state_ptr=76;
#endif
    SEM_CMD(dfmserver.semID_commonsem,4,-dfmserver.sem_maxval);
    if(cz_task->socket_used){
#ifdef _DEBUG_MODE_
        *state_ptr=77;
#endif
        free_string(&cz_task->fastlisp_errmsg);
    }
...

```

The above example code fragment defines two states. State number 76 is assigned when the system tries to enter the task connection zone (Semaphore 4). Synchronization state 77 appears while freeing memory block in the shared memory pool. All states together represent the state of the dataflow engine as shown below:

```

[Msg]: Current BM_DFM native processes states:
[Msg]:  N#          | CPUPROCs | OQPROCs | IORBPPROCs | PROCstat
[Msg]:  -----+-----+-----+-----+-----
[Msg]:      0          |      4   |      14  |           8 |      2
[Msg]:      1          |      4   |      14  |          17 |
[Msg]:      2          |      4   |      13  |           7 |
[Msg]:      3          |      4   |      14  |          17 |
[Msg]:      4          |     116  |      14  |          11 |
[Msg]:      5          |      4   |      13  |          17 |
[Msg]:      6          |      4   |      13  |          25 |
[Msg]:      7          |      4   |      14  |          11 |
[Msg]:      8          |     108  |      14  |          17 |
[Msg]: Current BM_DFM external processes states (continue):
[Msg]:  N#          | ExtTaskLd {PCount} | ExtTaskLs | ExtTrace
[Msg]:  -----+-----+-----+-----+-----
[Msg]:      0          |      35 {0000000025} |           8 |  _UNDEF_
[Msg]:      1          |  _UNDEF_ { _UNDEF_ } |  _UNDEF_ |  _UNDEF_

```

For each process the number of synchronization states is listed in Table 4.3.

Process	Number of states
IORBPPROC	92
OQPROC	16
CPUPROC	159
PROC stat	8
External task loader	56
External task listener	8
External tracer	9
TOTAL	348

Table 4.3. Program complexity of the dynamic scheduler

The time complexity expressed in the number of synchronization states depends on how many copies of the scheduling processes run in parallel. Thus it can be easily calculated using the following formula: $92 * N_{IORBPPROC} + 16 * N_{OQPROC} + 159 * N_{CPUPROC} + (56+8) * \text{Number_of_Loaders/Listeners}$. We assume that PROC stat and the external tracers are just a kind of auxiliary utilities, which can be run optionally.

4.14. Summary

We have designed an all-purpose SMP dataflow engine, which performs parallelization of sequential applications at runtime. Our architecture has the following features:

- **Multiple context data structuring.** This allows dataflow processing of the iterations in parallel, storing the iteration's data dynamically under different contexts. It is also a key point in resolving inter-procedural and cross-conditional dependencies.
- **Speculative tagging of instructions.** This is a solution how to significantly reduce dynamic scheduling overhead – the main problem of dataflow processing.

- **Parallel load of clusters.** This approach allows to avoid a bottleneck when the parallel dataflow machine is fed dynamically from the single threaded control virtual machine. In the proposed scheme the marshaled clusters are prepared statically on the compilation stage, which does not cause additional runtime overhead for marshaling.
- **Multithreading.** The dataflow engine functionality is fully distributed among the parallel processes. We have carefully tracked down all possible narrow/critical places in the code. All scheduling algorithms are multithreaded and do not have bottlenecks.
- **One-way locking policy.** There are four object locking algorithms in the dynamic scheduling subsystem. Obviously in each case we exclude a mutual blocking of the resources. In combination with distributive semaphore allocation in the shared memory pool we prevent a situation where the dataflow resources might be saturated.
- **Ordering results after out-of-order processing.** Because the data is processed on dataflow the results could be computed in an unpredictable order. Our dataflow architecture orders the output stream automatically in the TCZ output queue. Thus the output stream will always appear naturally ordered.
- **Multi-level granularity of parallelism.** The BMDFM virtual machine can define seamless macro-instructions, the bodies of which are prevented from dynamic scheduling. Exploitation of coarse-grain parallelism in addition to the fine-grain parallelism is more efficient as naturally less time is spent on dynamic scheduling.
- **Immediate garbage collector.** A postponed release of the shared resources is dangerous in case of shared memory dataflow processing. Our parallel algorithms are programmed to release unused resources immediately, which also prevents from having a kind of saturated dataflow.

Although the contribution of this thesis is the BMDFM dataflow architecture in its entirety, we would especially like to highlight the multiple context data structuring, speculative tagging of instructions and parallel load of clusters as our main contributions.

We have also estimated program complexity of the dynamic scheduling subsystem, which is expressed in 348 synchronization states.

Chapter 5

Static Scheduling Subsystem

- 5.1. Overview
- 5.2. Parallel Dataflow Code Style Restrictions
- 5.3. Static and Dynamic Type Casting
- 5.4. Code Reorganization
- 5.5. Generation of Marshaled Clusters
- 5.6. Uploading of Marshaled Clusters
- 5.7. Summary

5.1. Overview

In this chapter we describe all sequential stages necessary to transform and upload the conventional code into the dataflow engine. We call this flow a Static Scheduling of the code.

All transformations described in this chapter are performed fully automatically. Although the code examples are given in the virtual machine language notation, the same methodology can be applied to any other code semantics that formally consist of variable assignments, indirect addressing (indexed arrays), conditionals, loop processing and function declarations/invocations.

In the proposed static scheduling flow we have defined the following stages:

- **Checking for the parallel dataflow code style restrictions.** This formal checking procedure detects all code fragments that are dangerous for dataflow processing. In particular, it considers all kind of uninitialized variables suspending the dataflow.
- **Static and dynamic type casting.** This stage is optional and does not influence the static scheduling flow itself. Our implementation of the virtual machine language does not require explicit type declarations in the application's code. To reduce an overhead of the runtime type casting we analyze the code statically detecting the data types where it is possible.
- **Code reorganization.** Application code is split automatically into fragments, which have only one destination for a computed result per fragment. Such a fragment is supposed to be a seamless instruction for the dataflow processing.
- **Generation of marshaled clusters.** At this stage a sequential user application is split into code chunks called marshaled clusters. This transformation aims to prepare the application for a multithreaded dynamic uploading into the dataflow engine.
- **Uploading of marshaled clusters.** Marshaled cluster uploading is the final stage of the static scheduling flow. It is executed on the virtual front-end machine that acts as a von Neumann control machine for the dataflow engine.

5.2. Parallel Dataflow Code Style Restrictions

The purpose of checking for the parallel dataflow code style restrictions is to track down all variables that are used without prior initialization. Such an uninitialized variable endangers dataflow processing bringing it into an endless idle state (because the dataflow processing is controlled by firing of ready operands and instructions). To detect all uninitialized variables efficiently the checking algorithm always assumes a worse case of initialization as shown in Figure 5.1.

```
(progn
  (if (== a 0)
    (setq var 0) # initialization within a conditional
    nil
  )
)
```

```

(for i 1 1 n
  (setq var i) # initialization within a loop
)

(setq b (& a (setq var 1))) # initialization in the second argument
                             # of a boolean short-circuit operation.

(setq c var) # var is not initialized
)

```

Figure 5.1. Variable initialization within potentially unreachable code

Thus we can summarize that a variable is (still) considered to be uninitialized in the following cases:

- It was not initialized at all.
- It was previously initialized in a conditional statement.
- It was previously initialized within a loop (a worse assumption is that loop can iterate zero times as well).
- It was previously initialized in the second argument of a boolean and/or short-circuit operation that is not obligatorily evaluated.

The BMDFM restriction checking procedure additionally performs a few other checks that are implementation specific. A full report about the checked conditions can be seen in the logs:

```

* * * * *
* The BM_DFM CODE STYLE RESTRICTIONS Summary:
* ~~~~~
* o Variable names within the inclusive range of
* [TMP__000000000; TMP__999999999] are reserved.
* o 'SHADOW' is the reserved name for a UDF.
* o Array names should differ from ordinary variable names.
* o Every variable should be initialized before it is used.
* The following is an example of how to copy an array:
* ...
* (arsetq a 0 1)
* (arsetq a 1 5)
* (alsetq b (alindex a 2)) # instead of '(setq b a)'
* ...
* o The <step> and <limit> values of a <for> loop should be
* the integer numeric constants, function parameters or
* initialized variables which are not changed inside this
* <for> loop.
* o Second parameter of the booleans <or> and <and> should
* not include any assignments, I/O, conditional/
* iterational processing and UDF calls.
* NOTE: All conventional programs can be converted by a
* formal procedure to those that accept the above
* mentioned code style restrictions.
* * * * *

```

5.3. Static and Dynamic Type Casting

The BMDFM virtual machine is data type insensitive. The data type declarations can simply be omitted as shown below:

```

(progn
  (setq a "Dummy") # string
  (setq a 1.2) # float
  (setq a (+ a 2)) # casting to integer dynamically
)

```

To avoid the dynamic casting overhead the data types are analyzed at compile-time. After static type casting, the source code is modified slightly (Figure 5.2).

Source code	Modified code
(progn	(PROGN

(setq a (ival 1.2))	(SETQ@I A@I (IVAL@F 1.2))
(setq a (+ a 2))	(SETQ@I A@I (+@J A@I 2))
))

Figure 5.2. Static casting of data types

The casting procedure adds type suffixes to the functions and variables that specify their types. The suffixes are abbreviated from the standard type names:

- “I” stands for Integer.
- “F” stands for Float.
- “S” stands for String.
- “Z” for nil.
- “J” stands for Justified. This is a special type casting for functions that have more than one argument. In this case the function is justified if all argument types match or is not justified otherwise.

The suffixes provide additional information to the compiler and linker enabling optimized code generation. Having experimented on TAU [147, 148], we estimate a speed gain of 4-5 times when running statically casted VM code compared to dynamically casted VM code. Figures 5.3 and 5.4. explain the difference in the casting mechanisms on a physical level.

```

#if defined(OP_CODE__IADD)
void func__iadd(ULO *dat_ptr, struct fastlisp_data *ret_dat){
    SLO op_b;

    ret_ival(dat_ptr,&ret_dat->value.ival); // DYNAMIC CASTING for ARG 0.
    ret_ival(dat_ptr+1,&op_b); // DYNAMIC CASTING for ARG 1.

    if(noterror){
        ret_dat->single=1;
        ret_dat->type='I';
        ret_dat->value.ival+=op_b; // ADDITION
    }
    return;
}

void func__iadd_j(ULO *dat_ptr, struct fastlisp_data *ret_dat){
    ULO *tmp_ptr;
    SLO op_a;
    ret_dat->disable_ptr=1;

    tmp_ptr=((ULO**)dat_ptr); // ARG 0. NO RUNTIME CASTING NEEDED
    (*(fcall)*tmp_ptr)(tmp_ptr+1,ret_dat);
    op_a=ret_dat->value.ival;

    tmp_ptr=((ULO**)(dat_ptr+1)); // ARG 1. NO RUNTIME CASTING NEEDED
    (*(fcall)*tmp_ptr)(tmp_ptr+1,ret_dat);

    if(noterror)
        ret_dat->value.ival+=op_a; // ADDITION
    return;
}

#endif

OP_CODE_STRU OP_CODE[]={ // Internal registry for instructions
    // ...
#ifdef OP_CODE__IADD
    ,{OP_NAME__IADD.name,2,'I',"II",&func__iadd,&func__iadd_j,NULL,NULL,NULL}}
#endif
    // ...
};
const ULO OP_CODES=sizeof(OP_CODE)/sizeof(OP_CODE_STRU);

```

Figure 5.3. Physical meaning of justification

Native virtual machine functions use the same C interface, which is opened for user defined functions. The (+ ...) function has a dual implementation. Depending on the static casting the linker links a VM byte code with func__iadd() to resolve types at runtime or with func__iadd_j() otherwise.

Functions with only one argument have one generic implementation and four type dependent implementations. When linkage is applied to our VM code example the (IVAL@F ...) function is linked with the func__ival_f() implementation respectively.

```

#ifdef OP_CODE__IVAL

void func__ival(ULO *dat_ptr, struct fastlisp_data *ret_dat){
    ret_dat->single=1;
    ret_dat->type='I';

    ret_ival(dat_ptr,&ret_dat->value.ival); // DYNAMIC CASTING for ARG

    return;
}

void func__ival_i(ULO *dat_ptr, struct fastlisp_data *ret_dat){

    dat_ptr=((ULO**)dat_ptr);
    (*(fcall)*dat_ptr)(dat_ptr+1,ret_dat); // NO RUNTIME CASTING NEEDED

    return;
}

void func__ival_f(ULO *dat_ptr, struct fastlisp_data *ret_dat){
    ret_dat->disable_ptr=1;

    dat_ptr=((ULO**)dat_ptr);
    (*(fcall)*dat_ptr)(dat_ptr+1,ret_dat); // NO RUNTIME CASTING NEEDED

    if(noterror){
        ret_dat->type='I';
        if(ret_dat->value.fval<0.0)
            ret_dat->value.ival=(SLO)ceil((double)ret_dat->value.fval);
        else
            ret_dat->value.ival=(SLO)floor((double)ret_dat->value.fval);
    }
    return;
}

void func__ival_s(ULO *dat_ptr, struct fastlisp_data *ret_dat){
    ret_dat->disable_ptr=1;

    dat_ptr=((ULO**)dat_ptr);
    (*(fcall)*dat_ptr)(dat_ptr+1,ret_dat); // NO RUNTIME CASTING NEEDED

    if(noterror){
        ret_dat->type='I';
        ret_dat->value.ival=atol(ret_dat->svalue);
    }
    return;
}

void func__ival_z(ULO *dat_ptr, struct fastlisp_data *ret_dat){
    ret_dat->disable_ptr=1;

    dat_ptr=((ULO**)dat_ptr);
    (*(fcall)*dat_ptr)(dat_ptr+1,ret_dat); // NO RUNTIME CASTING NEEDED

    if(noterror){
        ret_dat->type='I';
        ret_dat->value.ival=0;
    }
    return;
}

#endif

OP_CODE_STRU OP_CODE[]={ // Internal registry for instructions
    // ...
#ifdef OP_CODE__IVAL
    ,{OP_NAME__IVAL.name,1,'I',"U",{&func__ival,&func__ival_i,&func__ival_f,
    &func__ival_s,&func__ival_z}}
#endif
    // ...
};

const ULO OP_CODES=sizeof(OP_CODE)/sizeof(OP_CODE_STRU);

```

Figure 5.4. Physical meaning of casting

5.4. Code Reorganization

The goal of code reorganization is to split an application code into fragments that have only one destination for a computed result per fragment. Such a fragment is supposed to be a seamless instruction for dataflow processing.

For a prefix notation the code reorganization mainly means moving the nested constructions to the upper nesting levels. The following example code demonstrates this idea. Additionally, variable and UDF names are prefixed with name of the parent function they are nested in. The most common parent function is the function “MAIN”.

Initial code
(setq a (setq b (setq c (+ 1 2))))
Preprocessed code
(SETQ@I MAIN:C@I (+@J 1 2))
(SETQ@I MAIN:B@I MAIN:C@I)
(SETQ@I MAIN:A@I MAIN:C@I)

Further we describe how the methodology of code reorganization is applied to the formal language constructions: UDF, I/O, conditionals and loops.

User defined functions.

- UDF invocation has to have its own destination variable. If it does not, then an artificial temporary variable TMP is automatically introduced (here and further on in the explanation text we use short names for the TMP variables, truncating superfluous zeros). UDF is always considered to be a coarse-grain function in contrast to all native VM functions, which are fine-grained. This is a very simple and at the same time very important rule that defines where a temporary TMP variable has to be automatically introduced. Suppose that in our example (Figure 5.5.) the variables TMP1 and TMP2 were not introduced and the coarse-grain UDF “true” performs heavy weight computations. Then the seamless expression (!= (true) (true)) will run on one CPU. In the other case, due to the artificially introduced variables the dataflow engine will schedule two invocations of “true” for multiple CPUs. Thus they will be executed in parallel.
- UDF declared at the virtual machine level always assigns variable TMP0 with its returned value. That instructs the dataflow engine to process returned values correctly. Note that this is valid only for programming model scheme A. A UDF declared on scheme B or scheme C (UDF is implemented in C language) is not preprocessed at all.

Initial code
(defun true (progn 1))
(setq false (!= (true) (true)))
Preprocessed code
(DEFUN MAIN:TRUE (SETQ@I MAIN:TRUE:TMP__000000000@I 1))
(SETQ@I MAIN:TMP__000000001@I (MAIN:TRUE))
(SETQ@I MAIN:TMP__000000002@I (MAIN:TRUE))
(SETQ@I MAIN:FALSE@I (!=@I MAIN:TMP__000000001@I MAIN:TMP__000000002@I))

Figure 5.5. Preprocessing of the UDF declaration and UDF invocation

Output.

- Functions, which generate an output, have to have their own destination variable. If they have not then an artificial temporary variable TMP is automatically introduced as shown in Figure 5.6.

Initial code
(outf "One = %d\n" 1)
(outf "Two = %d\n" (++ 1))

Preprocessed code
(SETQ@S MAIN:TMP__000000001@S (OUTF "One = %d\n" 1)) (SETQ@S MAIN:TMP__000000002@S (OUTF "Two = %d\n" (++@J 1)))

Figure 5.6. Preprocessing of the output functions

Input.

- In contrast to the output, which is processed in the dataflow engine, the input is organized in the control front-end VM. The control front-end VM communicates with the dataflow engine through the TCZ interface via `dfinget_data(var)/dfinput_data(var)`-like calls. Therefore input functions should have variables as arguments and destination. If they do not, then the artificial temporary variables are automatically introduced (Figure 5.7.). The same methodology is applied to file-based i/o.

Initial code
(setq number (++ (accept "Enter number: ")))
Preprocessed code
(SETQ@S MAIN:TMP__000000001@S "Enter number: ") (SETQ@S MAIN:TMP__000000002@S (ACCEPT MAIN:TMP__000000001@S)) (SETQ@I MAIN:NUMBER@I (++ MAIN:TMP__000000002@S))

Figure 5.7. Preprocessing of the input functions

Conditionals.

- Conditionals having no global constructions (`setq`, `arsetq`, `for`, `while`, `break`, `exit`) in conditional branches are not modified. Such a conditional is similar to the “?:” C equivalent. Otherwise the conditionals become global and are moved to the upper nested level. Figure 5.8 shows an example of two conditionals (“if a” is a global one and “if b” is in the local scope). In the preprocessed code the “if a” conditional is placed at the top level keeping the assignment of the variable “number” appropriate.

Initial code
(setq number (if a (if b 1 a) (setq b 1)))
Preprocessed code
(IF@J MAIN:A@I (SETQ@I MAIN:NUMBER@I (IF@J MAIN:B@I 1 MAIN:A@I)) (PROGN (SETQ@I MAIN:B@I 1) (SETQ@I MAIN:NUMBER@I 1)))

Figure 5.8. Preprocessing of conditionals in the global and local scopes

For- and while-loops.

- To process a loop the control front-end VM gets control variables from the dataflow engine and then controls the iteration process. Therefore artificial temporary variables modify the loop in a way that all loop controls are constants or variables (Figures 5.9 and 5.10.).

Initial code
(for i (- a b) 1 a (outf "%d\n" i))
Preprocessed code
(SETQ@I MAIN:TMP__000000001@I (-@J MAIN:A@I MAIN:B@I)) (FOR@J MAIN:I@I MAIN:TMP__000000001@I 1 MAIN:A@I (SETQ@S MAIN:TMP__000000002@S (OUTF "%d\n" MAIN:I@I)))

Figure 5.9. For-loop preprocessing

Initial code
(while (< a b) (progn (outf "%d\n" a) (setq a (++ a))))

Preprocessed code
<pre>(SETQ@I MAIN:TMP__000000001@I (<@I MAIN:A@I MAIN:B@I)) (WHILE@J MAIN:TMP__000000001@I (PROGN (SETQ@S MAIN:TMP__000000002@S (OUTF "%d\n" MAIN:A@I)) (SETQ@I MAIN:A@I (++@J MAIN:A@I)) (SETQ@I MAIN:TMP__000000001@I (<@I MAIN:A@I MAIN:B@I))))</pre>

Figure 5.10. While-loop preprocessing

Recursive call.

- In case of recursion a UDF calls itself from inside its body. This violates the rule that a UDF always has to assign its return value to the variable TMP0. To solve the problem we use the following trick. A copy of the recursive UDF is declared as a function “shadow” inside the UDF. Then the shadow function calls its parent UDF, and the UDF itself calls its own shadow copy. Such an intermediate presentation of recursion eliminates confusing the variable TMP0. Figure 5.11 shows the initial, intermediate and preprocessed code, respectively, for a classic recursive calculation of a factorial.

Initial code
<pre>(defun factorial_f (if (<= \$1 1.) 1. (*. \$1 (factorial_f (-. \$1 1.)))))</pre>
Intermediate code
<pre>(defun factorial_f (progn (defun shadow (if (<= \$1 1.) 1. (*. \$1 (factorial_f (-. \$1 1.))))) (if (<= \$1 1.) 1. (*. \$1 (shadow (-. \$1 1.))))))</pre>
Preprocessed code
<pre>(DEFUN MAIN:FACTORIAL_F (PROGN (DEFUN MAIN:FACTORIAL_F:SHADOW (PROGN (SETQ@I MAIN:FACTORIAL_F:SHADOW:TMP__000000002@I (<= MAIN:FACTORIAL_F:SHADOW:\$1 1.)) (IF@J MAIN:FACTORIAL_F:SHADOW:TMP__000000002@I (SETQ@F MAIN:FACTORIAL_F:SHADOW:TMP__000000000@F 1.) (PROGN (SETQ@F MAIN:FACTORIAL_F:SHADOW:TMP__000000001@F (MAIN:FACTORIAL_F (-. MAIN:FACTORIAL_F:SHADOW:\$1 1.))) (SETQ@F MAIN:FACTORIAL_F:SHADOW:TMP__000000000@F (*. MAIN:FACTORIAL_F:SHADOW:\$1 MAIN:FACTORIAL_F:SHADOW:TMP__000000001@F))))) (SETQ@I MAIN:FACTORIAL_F:TMP__000000002@I (<= MAIN:FACTORIAL_F:\$1 1.)) (IF@J MAIN:FACTORIAL_F:TMP__000000002@I (SETQ@F MAIN:FACTORIAL_F:TMP__000000000@F 1.) (PROGN (SETQ@F MAIN:FACTORIAL_F:TMP__000000001@F (MAIN:FACTORIAL_F:SHADOW (-. MAIN:FACTORIAL_F:\$1 1.))) (SETQ@F MAIN:FACTORIAL_F:TMP__000000000@F (*. MAIN:FACTORIAL_F:\$1 MAIN:FACTORIAL_F:TMP__000000001@F))))))</pre>

Figure 5.11. Preprocessing of recursion

5.5. Generation of Marshaled Clusters

Marshaled clusters are generated from the preprocessed user application after the code reorganization stage according to the following rule. A sequence of assignment statements can be merged to one cluster if this sequence is not interrupted by any global control statement (if, for, while, defun, input and file i/o, break, exit). Enumerated global statements are processed on the front-end VM controlling the marshaled cluster uploading process.

Figure 5.12 shows marshaled cluster containing three functions including one (Fnc #2) that contributes to the output stream.

Initial code
(setq a 1) (setq b 1) (outf "Result = %d\n" (+ a b))
Preprocessed code
(SETQ@I MAIN:A@I 1) (SETQ@I MAIN:B@I 1) (SETQ@S MAIN:TMP__000000001@S (OUTF "Result = %d\n" (+@J MAIN:A@I MAIN:B@I)))
Marshaled cluster
(marshaled_cluster (Vars_N#_Ref_Name_[Array] (0 0 "MAIN:A@I") (1 1 "MAIN:B@I") (2 14 "MAIN:TMP__000000001@S")) (Fnc (N# 0) (FLP (SETQ@I MAIN:A@I 1)) (FLP_COMPILED "D5 01 00 00" "01 00 00 00" "00 00 00 00" "D4 04 00 00" "00 00 00 00" "01 00 00 00" " I 00 00 00" "01 00 00 00") (Var_Ptrs 0)) (Fnc (N# 1) (FLP (SETQ@I MAIN:B@I 1)) (FLP_COMPILED "D5 01 00 00" "01 00 00 00" "00 00 00 00" "D4 04 00 00" "00 00 00 00" "01 00 00 00" " I 00 00 00" "01 00 00 00") (Var_Ptrs 1)) (Fnc (N# 2) (FLP (SETQ@S MAIN:TMP__000000001@S (OUTF "Result = %d\n" (+@J MAIN:A@I MAIN:B@I)))) (FLP_COMPILED "D5 01 00 00" "03 00 00 00" "00 00 00 00" "D4 05 00 00" "00 00 00 00" "01 00 00 00" " T 8 00 00" "02 00 00 00" "07 00 00 00" " S 00 00 00" "0C 00 00 00" " R e s u" " l t _ _ =" " _ _ % d 0A" "00 00 00 00" "D4 AC 00 00" "02 00 00 00" "03 00 00 00" " i 00 00 00" "01 00 00 00" " i 00 00 00" "02 00 00 00") (Inq_Dest Ls) (Var_Ptrs 2 0 1)))

Figure 5.12. Generation of marshaled cluster

Marshaled cluster consists of local variable directory (variables A, B and TMP1 in our example) and a group of functions (enumerated from 0 to 2). Physically, the functions are located in the TCZ function directory, they are only referenced from the cluster. Each function has a list of references (one reference per parameter) to the local variable directory (Var_Ptrs). Optionally, each function can have destination attributes (Inq_Dest) if the result is required by loader (Ld) or listener (Ls).

The structure of the marshaled cluster has the following advantages:

- A cluster is fully relocatable as it contains only function references to the TCZ function directory and variable references from the local variable directory to the DB.
- A cluster is maximally compact to be transferred and decoded at runtime.
- Each function is associated with a VM byte code fragment that can be more than one VM function (Fnc #2 has nested a+b). Further each cluster function becomes an atomic instruction in the dataflow engine.

5.6. Uploading of Marshaled Clusters

The control front-end virtual machine (the loader part of the external task listener/loader pair) uploads the marshaled clusters to the dataflow engine according to the generated control sequence. There is no special need to create a control sequence manually. After the preliminary preprocessing stages a user application is already split into marshaled clusters and control directives automatically. Figure 5.13 illustrates the uploading process. Global constructions (input, if/else, for/next) are processed on the control front-end VM in a classic von Neumann manner. The VM communicates with the dataflow engine through the bi-directional TCZ interface uploading the marshaled clusters and obtaining data necessary to control the uploading sequence.

The contexts of the marshaled clusters are modified dynamically when the clusters are delivered to the TCZ interface. The VM stores the current contexts of all application's variables in a table. The clusters are loaded with the current contexts, which are taken from the VM context table. Then contexts are incremented to the next unique values for those destination variables which are reassigned in the clusters.

The proposed scheme provides one serious advantage that considerably reduces dynamic scheduling overhead in the dataflow engine. The dataflow runtime engine is fed with the clusters dynamically in the order of application workflow. So the dataflow engine will stay away from the clusters, which are not in the current processing scope.

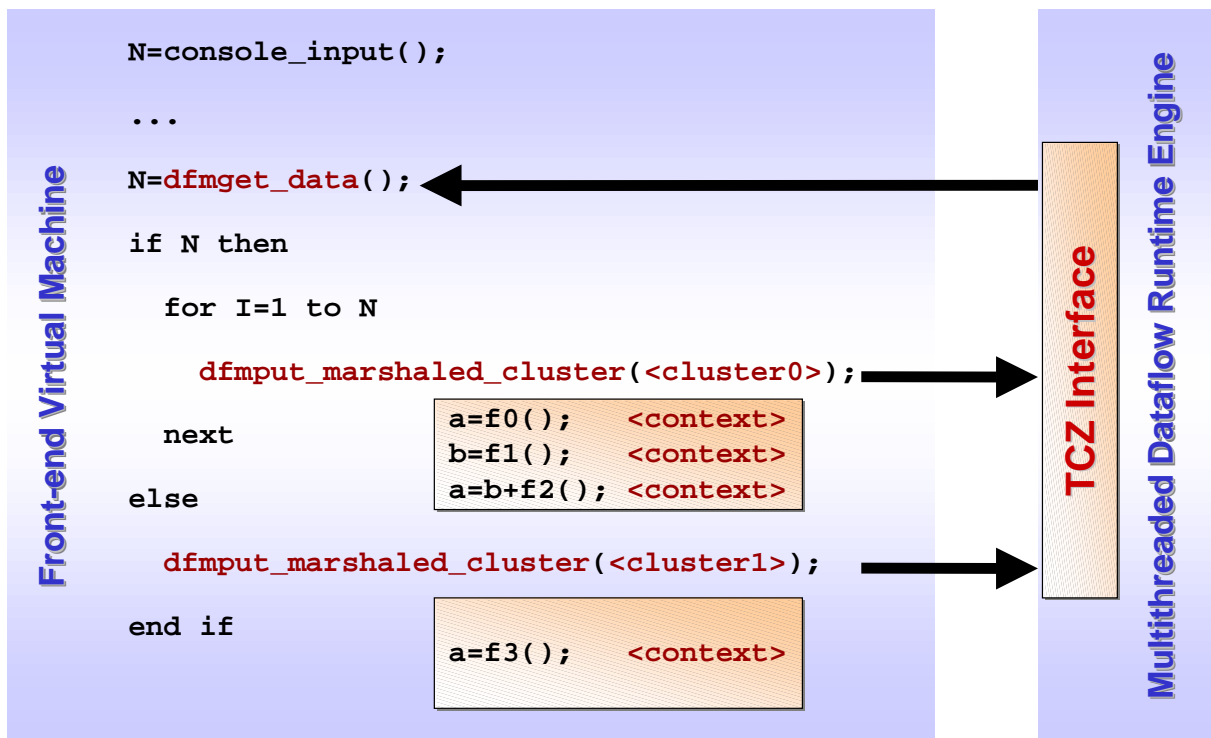


Figure 5.13. Dataflow engine controlled by the front-end VM

The front-end VM is a von Neumann control machine with the traditional program counter (PC) and stack pointer (SP). Table 5.1 summarizes the architecture of all VM registers. The register *context is associated with the context table of all application's variables. This table is initialized once at startup and exists during the whole application life cycle, therefore we do not store *context register in a stack while push/pop. The VM accumulator can store both integer and string values (accum_slo and accum_uch respectively). Additionally, three loop control integer registers are intended for organizing "for" iterations. The loop control, loop step and loop end

limit are stored correspondingly in the loop_slo, loopstep_slo and loopto_slo registers. Because an application can have multiple nested loops we save these registers on the stack when entering a new nested loop.

The last register which has to be mentioned is *var_lst. It points to the list of UDF local variables. When the control sequence enters a new local UDF the variable list of a parent UDF is saved on the stack. Furthermore, when the control sequence leaves the UDF all contexts from the current *var_lst are released through the dfmput_release() call of the TCZ interface, and the previous variable list is restored from the stack to the *var_lst.

Register	Description
*contexts	Current contexts of all application variables
accum_slo	Accumulator for integer values
accum_uch	Accumulator for string values
loop_slo	Loop control register
loopstep_slo	Loop step register
loopto_slo	Loop end limit register
*var_lst	Current UDF variable list
PC	Program counter
SP	Stack pointer

Table 5.1. Register architecture of the front-end control VM

Four groups of instructions are defined for the front-end control VM (all are explained in Table 5.2.):

- **Group 1** includes all instructions to communicate with the dataflow engine through the TCZ interface. Actually, they repeat functionality of the TCZ interface.
- **Group 2** is used to control the VM itself. These are classic jump and stack instructions.
- **Group 3** is dedicated to the VM input/output.
- **Group 4** organizes iteration control.

Mnemonic / OpGroup	Mnemonic / COP / Description	
DFM	1	PUTCLUSTER 50 Sends marshaled cluster to the dataflow engine
		PUTZDATA 70 Sends nil variable to the dataflow engine
		PUTIDATA 71 Sends integer variable to the dataflow engine
		PUTFDATA 72 Sends float variable to the dataflow engine
		PUTSDATA 73 Sends string variable to the dataflow engine
		GETIDATA 81 Gets integer data from the dataflow engine
		GETSDATA 83 Gets string data from the dataflow engine
CONTROL0	2	PUSHA 10 Pushes registers (except *contexts & *var_lst)
		POPA 11 Pops registers (except *contexts & *var_lst)
		ENTERRECURSION 12 Pushes *var_lst
		LEAVERECURSION 13 Pops *var_lst
		GOTO 14 Jumps unconditionally
		GOSUB 15 Calls subroutine
		RETURN 16 Returns from subroutine
IO	3	IFGOTO 17 Jumps conditionally
		ACCEPT 20 Inputs from stdin
		SCANCONSOLE 21 Inputs from console
		FILECREATE 22 Creates file
		FILEOPEN 23 Opens file
		FILEWRITE 24 Writes to file
		FILEREAD 25 Reads from file
CONTROL1		FILECLOSE 26 Closes file
		FILEREMOVE 27 Deletes file
	4	LOOP 90 Sets loop registers
	FOR 100 Organizes for loop	

	NEXT	101	Iterates
	END	200	Ends control sequence

Table 5.2. Instruction set matrix of the front-end control VM

Further we explain the uploading of marshaled clusters in three use cases: local UDF invocation, for-loop and while-loop. In each case we analyze the generated control sequence.

Local UDF invocation. Figure 5.14 shows a fragment of code, which calls a locally defined function. The target control sequence keeps the modified UDF in a cluster (CTRL 2). The UDF is invoked in the GOSUB statement of CTRL 5, prior to which all UDF local variables are saved on the stack (ENTER_RECURSION CTRL 4). The function in CTRL 6 copies the UDF return value to the variable “a” of the function main and restores the local variables by popping them from the stack (LEAVE_RECURSION CTRL 7).

Initial code
(defun true (progn 1))
(setq a (true))
Preprocessed code
(DEFUN MAIN:TRUE (SETQ@I MAIN:TRUE:TMP__000000000@I 1) (SETQ@I MAIN:A@I (MAIN:TRUE)))
Control sequence for the front-end VM
(CTRL (N# 1) (OpGroup 2) (COP 14) (GOTO 4) (REM "Pass over UDF 'MAIN:TRUE' body")) (CTRL (N# 2) (OpGroup 1) (COP 50) (dfmput_marshaled_cluster (Vars_N#_Ref_Name_[Array] (0 14 "MAIN:TRUE:TMP__000000000@I")) (Fnc (N# 0) (SETQ@I MAIN:TRUE:TMP__000000000@I 1) (Var_Ptrs 0)))) (CTRL (N# 3) (OpGroup 2) (COP 16) (RETURN) (REM "End of UDF 'MAIN:TRUE' body")) (CTRL (N# 4) (OpGroup 2) (COP 12) (ENTER_RECURSION) (Vars_N#_Ref_Name_[Array] (0 14 "MAIN:TRUE:TMP__000000000@I"))) (CTRL (N# 5) (OpGroup 2) (COP 15) (GOSUB 2) (REM "UDF 'MAIN:TRUE' call")) (CTRL (N# 6) (OpGroup 1) (COP 50) (dfmput_marshaled_cluster (Vars_N#_Ref_Name_[Array] (0 0 "MAIN:A@I") (1 14 "MAIN:TRUE:TMP__000000000@I")) (Fnc (N# 0) (ALSETQ MAIN:A@I MAIN:TRUE:TMP__000000000@I) (Var_Ptrs 0 1)) (REM "UDF 'MAIN:TRUE' returned value")) (CTRL (N# 7) (OpGroup 2) (COP 13) (LEAVE_RECURSION))

Figure 5.14. Control sequence template of local UDF invocation

For-loop. A typical control sequence of a “for” loop is shown in Figure 5.15. In the preamble section the VM registers are saved on the stack (PUSHA CTRL 4). The following code (CTRL 5 – 9) prepares loop controls retrieving loop_slo and loopto_slo values from the dataflow engine. The loop itself is organized on CTRL 10 – 13. The value of the loop control variable is delivered to the dataflow engine via dfmput_idata() call twice: inside the iteration (CTRL 11) and one time after the loop is finished (CTRL 14). The epilogue section restores the VM registers back from the stack (POPA CTRL 15).

Initial code
(for i a 1 b (outf "%d\n" a))
Preprocessed code
(FOR@J MAIN:I@I MAIN:A@I 1 MAIN:B@I (SETQ@S MAIN:TMP__000000001@S (OUTF "%d\n" MAIN:A@I)))
Control sequence for the front-end VM
(CTRL (N# 4) (OpGroup 2) (COP 10) (PUSHA)) (CTRL (N# 5) (OpGroup 1) (COP 70) (dfmput_zdata (VarRef 0) (VarName "MAIN:A@I") (Inq_Dest Ld)) (REM "<For> 'MAIN:I@I' loop initialization begins here")) (CTRL (N# 6) (OpGroup 1) (COP 81) (SubCOP 1) (<loop_slo> (dfmget_idata))) (CTRL (N# 7) (OpGroup 4) (COP 90) (SubCOP 2) (<loopstep_slo> 1)) (CTRL (N# 8) (OpGroup 1) (COP 70) (dfmput_zdata (VarRef 1) (VarName "MAIN:B@I") (Inq_Dest Ld))) (CTRL (N# 9) (OpGroup 1) (COP 81) (SubCOP 3) (<loopto_slo> (dfmget_idata))) (CTRL (N# 10) (OpGroup 4) (COP 100) (FOR <loop_slo> (STEP <loopstep_slo>) (TO <loopto_slo>) (BODY 14)) (REM "Controlled by 'MAIN:I@I' variable")) (CTRL (N# 11) (OpGroup 1) (COP 71) (SubCOP 1) (dfmput_idata <loop_slo> (VarRef 8) (VarName "MAIN:I@I"))) (CTRL (N# 12) (OpGroup 1) (COP 50) (dfmput_marshaled_cluster (Vars_N#_Ref_Name [Array] (0 0 "MAIN:A@I") (1 15 "MAIN:TMP__000000001@S")) (Fnc (N# 0) (SETQ@S MAIN:TMP__000000001@S (OUTF "%d\n" MAIN:A@I)) (Inq_Dest Ls) (Var_Ptrs 1 0)))) (CTRL (N# 13) (OpGroup 4) (COP 101) (SubCOP 1) (NEXT (BODY 10)) (REM "Controlled by 'MAIN:I@I' variable")) (CTRL (N# 14) (OpGroup 1) (COP 71) (SubCOP 1) (dfmput_idata <loop_slo> (VarRef 8) (VarName "MAIN:I@I")) (REM "<For> postloop 'MAIN:I@I' control variable value")) (CTRL (N# 15) (OpGroup 2) (COP 11) (POPA))

Figure 5.15. Control sequence template of for-loop

While-loop. The loop we consider is additionally complicated by a “break” termination (Figure 5.16.). It is nearly impossible to process terminators (break, exit) on the dataflow when several iteration contexts are already wandering across the dataflow engine. Therefore the only possible place is to embed them into a control sequence for the front-end control VM.

In the preamble section we compute the while-loop control condition (CTRL 4) and save the VM registers on the stack (PUSHA CTRL 5). The loop body (CTRL 6 – 17) retrieves the control condition value from the dataflow engine in every iteration (CTRL 6 – 7) and exits the loop (IF_NOT GOTO CTRL 8) when finished. The cluster in CTRL 9 performs mainly all loop computations. CTRL 10 – 15 are an implementation of the “break” termination. The next cluster CTRL 16 recalculates a new value for the while-loop control condition. Then the unconditional jump redirects back to the beginning of the loop (GOTO CTRL 17). Finally, the epilogue section restores the VM registers from the stack (POPA CTRL 18).

Initial code
<pre>(while (< a b) (progn (outf "%d\n" a) (setq a (++ a)) (if (> a 100) (break) nil)))</pre>
Preprocessed code
<pre>(SETQ@I MAIN:TMP__000000001@I (<@I MAIN:A@I MAIN:B@I)) (WHILE@J MAIN:TMP__000000001@I (PROGN (SETQ@S MAIN:TMP__000000002@S (OUTF "%d\n" MAIN:A@I)) (SETQ@I MAIN:A@I (++@J MAIN:A@I)) (SETQ@I MAIN:TMP__000000005@I (>@I MAIN:A@I 100)) (IF@J MAIN:TMP__000000005@I (BREAK) (SETQ@Z MAIN:TMP__000000004@Z NIL))) (SETQ@I MAIN:TMP__000000001@I (<@I MAIN:A@I MAIN:B@I))))</pre>
Control sequence for the front-end VM
<pre>(CTRL (N# 4) (OpGroup 1) (COP 50) (dfmput_marshaled_cluster (Vars_N#_Ref_Name_[Array] (0 0 "MAIN:A@I") (1 1 "MAIN:B@I") (2 14 "MAIN:TMP__000000001@I"))) (Fnc (N# 0) (SETQ@I MAIN:TMP__000000001@I (<@I MAIN:A@I MAIN:B@I)) (Var_Ptrs 2 0 1)))) (CTRL (N# 5) (OpGroup 2) (COP 10) (PUSHA)) (CTRL (N# 6) (OpGroup 1) (COP 70) (dfmput_zdata (VarRef 14) (VarName "MAIN:TMP__000000001@I") (Inq_Dest Ld)) (REM "<While> 'MAIN:TMP__000000001@I' loop body begins here")) (CTRL (N# 7) (OpGroup 1) (COP 81) (SubCOP 1) (<loop_slo> (dfmget_idata))) (CTRL (N# 8) (OpGroup 2) (COP 17) (SubCOP 1) (IF_NOT <loop_slo> (GOTO 18)) (REM "Exit <while> loop")) (CTRL (N# 9) (OpGroup 1) (COP 50) (dfmput_marshaled_cluster (Vars_N#_Ref_Name_[Array] (0 0 "MAIN:A@I") (1 15 "MAIN:TMP__000000002@S") (2 0 "MAIN:A@I") (3 17 "MAIN:TMP__000000005@I"))) (Fnc (N# 0) (SETQ@S MAIN:TMP__000000002@S (OUTF "%d\n" MAIN:A@I)) (Inq_Dest Ls) (Var_Ptrs 1 0)) (Fnc (N# 1) (SETQ@I MAIN:A@I (++@J MAIN:A@I)) (Var_Ptrs 2 0)) (Fnc (N# 2) (SETQ@I MAIN:TMP__000000005@I (>@I MAIN:A@I 100)) (Var_Ptrs 3 2))))</pre>

```

)
(CTRL (N# 10) (OpGroup 1) (COP 70)
  (dfmput_zdata (VarRef 17) (VarName "MAIN:TMP__000000005@I")
    (Inq_Dest Ld)
  )
)
(CTRL (N# 11) (OpGroup 1) (COP 81)
  (<accum_slo> (dfmget_idata))
)
(CTRL (N# 12) (OpGroup 2) (COP 17)
  (IF_NOT <accum_slo> (GOTO 15))
  (REM "Pass over 'MAIN:TMP__000000005@I' <if> conditional branch")
)
(CTRL (N# 13) (OpGroup 2) (COP 14)
  (GOTO 18) (REM "BREAK")
)
(CTRL (N# 14) (OpGroup 2) (COP 14)
  (GOTO 16)
  (REM "Pass over 'MAIN:TMP__000000005@I' <else> conditional branch")
)
(CTRL (N# 15) (OpGroup 1) (COP 50)
  (dfmput_marshaled_cluster
    (Vars_N#_Ref_Name_[Array] (0 16 "MAIN:TMP__000000004@Z"))
    (Fnc (N# 0)
      (SETQ@Z MAIN:TMP__000000004@Z NIL)
      (Var_Ptrs 0)
    )
  )
)
(CTRL (N# 16) (OpGroup 1) (COP 50)
  (dfmput_marshaled_cluster
    (Vars_N#_Ref_Name_[Array]
      (0 0 "MAIN:A@I")
      (1 1 "MAIN:B@I")
      (2 14 "MAIN:TMP__000000001@I")
    )
    (Fnc (N# 0)
      (SETQ@I MAIN:TMP__000000001@I (<@I MAIN:A@I MAIN:B@I))
      (Var_Ptrs 2 0 1)
    )
  )
)
(CTRL (N# 17) (OpGroup 2) (COP 14) (SubCOP 1)
  (GOTO 6)
  (REM "Continue <while> 'MAIN:TMP__000000001@I' loop,"
    " <while> loop body ends here"
  )
)
(CTRL (N# 18) (OpGroup 2) (COP 11)
  (POPA)
)
)

```

Figure 5.16. Control sequence template of while-loop

5.7. Summary

This chapter analyzes the static scheduling subsystem of BMDFM. The purpose of the static scheduling is to preprocess and to reorganize conventional input code into a set of marshaled clusters and a control sequence. The static scheduler is designed as a hybrid architecture, in which dataflow engine is controlled by a von Neumann front-end VM running the control sequence and uploading generated marshaled clusters into the dataflow engine. Two topics are described in details:

- At first, we explain all code transformations that are necessary to generate the target marshaled clusters and the control sequence.
- Secondly, we discuss the architecture of the front-end VM that has a minimal and sufficient set of facilities to control the dataflow engine.

The chapter contributes with the following ideas:

- The proposed methodology of the code preprocessing/reorganization is applicable to any conventional code semantics that formally consist of variable assignments, conditional processing, loop iterations and function declarations/invocations.

- There is no need for a special dataflow language. The marshaled clusters and control sequence are derived from a user application's code automatically.
- To reduce dynamic scheduling overhead in a runtime dataflow engine the dataflow engine is fed with the clusters dynamically in the order of the application workflow. So the dataflow engine will stay away from the clusters, which are not in the current processing scope.

Chapter 6

Transparent Dataflow Semantics

6.1. Overview

6.2. Conventional Programming Paradigm

6.3. Synchronization of Asynchronous Coarse-Grain and Fine-Grain Functions

6.4. Ordering Non-Standard Stream in Out-of-Order Processing

6.5. Speculative Dataflow Processing

6.6. Summary

6.1. Overview

In this chapter we would like to put emphasis on the main feature of the proposed architecture – to provide a conventional programming paradigm at the top level. We call this ability of BMDFM transparent dataflow semantics, the convenience of which justifies all the effort we spent designing the complex architecture behind it. We think that, because the complexity is hidden, transparent dataflow semantics is a key point defining the applicability of the BMDFM system.

The mentioned transparency makes the discussion in this chapter relatively short.

6.2. Conventional Programming Paradigm

BMDFM provides a conventional programming environment with transparent dataflow semantics. No directives for parallel execution are required! A user understands BMDFM as a virtual machine, which runs every statement of an application program in parallel having all parallelization and synchronization mechanisms fully transparent. The statements of an application program are normal operators, which any single threaded program might consist of - they include variable assignments, conditional processing, loops, function calls, etc.

Suppose we have the code fragment shown below:

```
(setq a (udf1 x))    # a=udf1(x);  udf stands for user defined
(setq b (udf2 x))    # b=udf2(x);  function
(setq b (++ b))      # b++;
(outf "a = %d\n" a) # printf("a = %d\n",a);
(outf "b = %d\n" b) # printf("b = %d\n",b);
```

The two first statements are independent, so the dataflow engine can run them on different processors. The two last statements can also run in parallel but only after “a” and “b” are computed. The dataflow engine recognizes dependencies automatically because of its ability to build a dataflow graph dynamically at run time (in contrast to a static Petri Net). Additionally, the dataflow engine orders the output stream to output the results sequentially. Thus even after the out-of-order processing the results will appear in a natural way.

Suppose that above code fragment now is nested in a loop:

```
(for i 1 1 N (progn  # for(i=1;i<=N;i++){
  (setq a (udf1 x))
  (setq b (udf2 x))
  (setq b (++ b))
  (outf "a = %d\n" a)
  (outf "b = %d\n" b)
}) # }
```

The dataflow engine will keep variables “a” and “b” under unique contexts for every iteration. Actually, these are different copies of the variables in the shared memory pool. A context variable exists until it is referenced by instruction consumers. Later non-referenced contexts will be garbage collected at runtime. Therefore the dataflow

engine can exploit both local parallelism within the iteration and global parallelism as well running multiple iterations simultaneously.

In comparison with the traditional directive-based paradigm for SMP (OpenMP) our approach has the following advantages:

- According to OpenMP the user explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel. OpenMP implementations do not check for dependencies, conflicts, deadlocks, race conditions or other problems that result in incorrect program execution. The user is responsible for ensuring that the application using the OpenMP API constructs executes correctly. The BMDFM dataflow engine takes care of synchronization at runtime automatically. No locks, semaphores, mutexes, barriers or others are needed.
- Although the fork-join model can be useful for solving a variety of problems, it is somewhat tailored for large array-based applications. BMDFM is not specialized for a range of tasks.
- The OpenMP principle is based on barrier points, in which all team threads should be synchronized (“nowait” directive will not help very much). Thus a system can idle for the longest thread. The dataflow engine loads available CPUs in a natural and balanced manner.
- In OpenMP any unsynchronized calls to output functions may result in output in which data written by different threads appears in non-deterministic order. Similarly, unsynchronized calls to input functions may read data in non-deterministic order. In BMDFM this problem is solved. If non-deterministic order of i/o is really necessary the user can write his own simple asynchronous i/o UDF.

6.3. Synchronization of Asynchronous Coarse-Grain and Fine-Grain Functions

In addition to the pure computations, an application very often accesses different peripheral devices, performs graphical output or reads/writes dynamically allocated memory. All these asynchronous resources have a similar access policy through interface functions: to obtain a descriptor first and then to refer the resource using this descriptor. In the case of dynamic memory a memory address is used. In this section we discuss how to synchronize the asynchronous interface functions on dataflow having no special synchronization directives but using only return values of the functions. We generalize it into a methodology for coarse-grain and fine-grain access types.

Suppose we have the source code fragment shown below that allocates three memory blocks dynamically (p, c0 and c1). Block “p” from the producer loop is used in two consumer loops. Formally, we can say that memory release calls are also a kind of consumer of the memory blocks.

<pre> p=malloc(<array_size>); // allocation c0=malloc(<array_size>); c1=malloc(<array_size>); for(i=0;i<N;i++) // loop-producer *(p+i)=...; for(i=0;i<N;i++) // loop-consumer 0 *(c0+i)=...*(p+i)...; for(i=0;i<N;i++) // loop-consumer 1 *(c1+i)=...*(p+i)...; free(p); // memory release free(c0); free(c1); </pre>

In the proposed programming environment the corresponding coarse-grain functions will be generated or created manually, which are regarded as the seamless instructions of the virtual machine.

<pre> p_udf(*p){ // function-producer for(i=0;i<N;i++) *(p+i)=...; return p; // repeats the address } c0_udf(*c0, *p){ // function-consumer 0 for(i=0;i<N;i++) *(c0+i)=...*(p+i)...; return c0; // repeats the address } </pre>
--

```

}
c1_ufd(*c1, *p){           // function-consumer 1
  for(i=0;i<N;i++)
    *(c1+i)=...*(p+i)...;
  return c1;               // repeats the address
}

```

Then these coarse-grain seamless functions are called on the VM level and synchronized on the dataflow engine due to artificial dependencies created via simple boolean bitwise operations.

```

(setq p_sync (& 0
  (setq p (asyncheap_malloc <array_size>)) # allocation
))
(setq c0_sync (& 0
  (setq c0 (asyncheap_malloc <array_size>))
))
(setq c1_sync (& 0
  (setq c1 (asyncheap_malloc <array_size>))
))
(setq p_sync (& 0
  (setq p (p_ufd (| p p_sync))) # calls function-producer
))
(setq p_sync (setq c0_sync (& 0
  (setq c0 (c0_ufd (| c0 c0_sync) p)) # calls function-consumer 0
)))
(setq p_sync (setq c1_sync (& 0
  (setq c1 (c1_ufd (| c1 c1_sync) p)) # calls function-consumer 1
)))
(asyncheap_free (| p p_sync)) # memory release
(asyncheap_free (| c0 c0_sync))
(asyncheap_free (| c1 c1_sync))

```

The synchronization rules are trivial:

- Every pointer variable has a corresponding synchronization “_sync” variable.
- A pointer to a read-only area is passed as a direct argument. Thus parallel independent calls are possible.
- A pointer to a modified area is passed as a dependent pair of the pointer and “_sync” variables.
- After the call is finished both “_sync” variables (for all pointers involved in the call) and the pointer variables (only for pointers to the modified areas) are touched. Thus the asynchronous coarse grain producer/consumer functions will be synchronized due to the fact that BMDFM is capable of using the transparent dataflow semantics.

To exploit the fine-grain parallelism of loops the same synchronization rules are used. Below it is shown of how this applies to the c0_ufd consumer loop function. The function itself is modified slightly to make the loop interleaved via “start” and “step” arguments.

```

c0_ufd_(start, step, *c0, *p){ // modified function-consumer 0.
  for(i=start;i<N;i+=step) // loop is interleaved
    *(c0+i)=...*(p+i)...;
  return 0; // now returns zero
}

```

On the VM level the function call is synchronized through the “c0_sync” and “p_sync” variables.

```

(setq nthreads (n_cpuproc)) # number of configured CPU PROCs

(setq c0 (| c0 c0_sync)) # prologue
(for thread 1 1 nthreads
  (setq c0_sync (+ c0_sync
    (c0_ufd_ (-- thread) nthreads c0 p) # calls modified function-consumer 0
  ))
)
(setq p_sync c0_sync) # epilogue
(setq c0 (| c0 c0_sync))

```

To remove dependencies of the `c0_ufd_` function calls the preprocessor reorganizes the code introducing a temporary variable `TMP1`. Now the time consuming calls of `c0_ufd_` function are context independent and can be executed in parallel at the dataflow level.

Initial code
<pre>(for thread 1 1 nthreads (setq c0_sync (+ c0_sync (c0_ufd_ (-- thread) nthreads c0 p))))</pre>
Preprocessed code
<pre>(FOR@J MAIN:THREAD@I 1 1 MAIN:NTHREADS@I (PROGN (SETQ@I MAIN:TMP__000000001@I (MAIN:C0_UDF_ (--@J MAIN:THREAD@I) MAIN:NTHREADS@I MAIN:C0@I MAIN:P@I)) (SETQ@I MAIN:C0_SYNC@I (+@J MAIN:C0_SYNC@I MAIN:TMP__000000001@I))))</pre>

6.4. Ordering Non-Standard Stream in Out-of-Order Processing

The dataflow runtime engine processes data in an out-of-order fashion but the output results have to be ordered as if the application were processed on a single processor. The dataflow engine recognizes and orders only standard output streams automatically.

A non-standard case can be easily solved as well due to the transparent dataflow semantics. Suppose we have a function `devicewrite()` that writes data to a special device by chunks. In the function `main` an array of data chunks is processed first and then delivered to `devicewrite()`.

<pre>devicewrite(*chunk){ // non-standard output function. // ... // performs non-standard output return; } *process(*chunk){ // chunk process function. // ... // performs chunk processing return chunk; } **chunks[N]; // array of data chunks for(i=0;i<N;i++) // processes and outputs array of chunks devicewrite(process(chunks[i]));</pre>
--

A user has to create his own interface function with a dummy argument “enable”, which repeats its value as the result of the interface function (very often in a real implementation such an argument has a meaning of the opened device descriptor).

<pre>devicewrite_udf(enable, *chunk){ // modified non-standard output function. devicewrite(chunk); // calls non-standard output function. return enable; // repeater }</pre>

At the virtual machine level the interface function is called as any other conventional function. An additional synchronization variable “wenable” is reassigned every time the interface function is called. Thus the problem is reduced to a common case of data dependencies.

Because the preprocessor uses a policy that all UDF invocations have to have their own destination variables, an artificial temporary variable `TMP1` is automatically introduced. Now the dataflow engine calls `process()` functions in a parallel out-of-order manner but the non-standard output stream will be ordered due to the data dependencies.

The same technique can be applied to all cases where out-of-order processing has to be ordered.

Initial code
<pre>(setq wenable 1) # write enable (for i 1 1 N (setq wenable (devicewrite_udf wenable # repeats write enable (process (index chunks (-- i))))</pre>

```

))
)
Preprocessed code
(SETQ@I MAIN:WENABLE@I 1)
(FOR@J MAIN:I@I 1 1 MAIN:N@I (PROGN
  (SETQ@I MAIN:TMP__00000001@I # Out-of-order processing
    (MAIN:PROCESS (INDEX@I MAIN:CHUNKS@I (--@J MAIN:I@I)))
  )
  (SETQ@I MAIN:WENABLE@I # Ordered on dataflow due to the data dependencies
    (MAIN:DEVICEWRITE_UDF MAIN:WENABLE@I MAIN:TMP__00000001@I)
  )
))

```

6.5. Speculative Dataflow Processing

Speculative processing enables parallel computations in the branches of code in advance hoping that results of the processed branches will be necessary. Speculative processing is useful because the computing resources can process a dependent code section instead of idle until the dependency is resolved. We propose very simple and elegant technique of how to process speculatively on dataflow:

- A conditional variable, which creates dependency, is allocated in the shared memory pool. A reference address is used instead of the variable that makes dataflow dependency disappeared (the address itself is data that is ready).
- A conditional that checks the conditional variable is moved into the seamless user defined functions. Such a conditional controls whether the UDF body will be processed.
- Finally, all UDF-instructions are uploaded into the dataflow engine to be processed speculatively. Optionally, the control front-end VM can restrict the number of uploaded instructions keeping the dataflow resources not overloaded.

The following example demonstrates a pseudo-code fragment with loop-carried control dependency. Each next process() UDF can not be uploaded into the dataflow engine because of the status value yielded by the previous call of the same UDF.

Callee (a seamless UDF processed on the Multithreaded Dataflow Engine)
<pre> boolean process(*chunk){ status; // a UDF that processes a chunk // and returns 'true' or 'false' // processing... // ==> status=true_false; return status; } </pre>
Caller (code processed by the Von-Neumann Control Front-end VM)
<pre> **chunks[N]; // array of data chunks for(i=0;i<N;i++){ status=process(chunks[i]); if(!status) // loop-carried control dependency break; } </pre>

The modified speculative version of the same example is shown below. Now the UDF body will be processed only if global_status is still true. Multiple UDF invocations can be uploaded into the dataflow engine as they do not depend from each other anymore. Resulting status is accumulated in variable “status” through the logical multiplication. A simple modulo operation (%1000) restricts the number of speculative instructions uploaded into the dataflow engine. After speculative processing of the previous 1000 instructions the resulting status is checked to determine whether the next speculative group of instructions has to be uploaded.

Callee (a seamless UDF processed on the Multithreaded Dataflow Engine)
<pre> boolean process(*chunk, *global_status){ if(global_status){ // global status is in shared memory. </pre>

```

    // processing... // actual processing might not happen
    // ==> global_status=true_false;
}
return global_status;
}
Caller (code processed by the Von-Neumann Control Front-end VM)
**chunks[N]; // array of data chunks

*global_status=allocate_in_shared_memory(true);
status=true;

for(i=0;i<N;i++){

    status&=process(chunks[i],global_status);

    if(!(i%1000)) // upload every 1000 process()
        if(!status) // instructions into the DataFlow
            break; // Engine speculatively
}

// artificial dependency to complete the loop first and then deallocate
deallocate_in_shared_memory(global_status*(status|1));

```

6.6. Summary

In this chapter we define the transparent dataflow semantics as a conventional programming paradigm on top of the dataflow. No synchronization and parallelization directives are needed.

The directive-based approach is intended for numeric loop computations, thus it is restricted to the numeric computation area. Our approach presumes that any application, presented in a conventional form, will be processed in parallel automatically. Such an application could come from the symbol processing area, adaptive algorithms or whatever (also including numeric computation as well).

Within the transparent dataflow semantics we explain the methodology of how to synchronize asynchronous coarse/fine-grain functions, how to order non-standard streams in out-of-order processing and how to process speculatively. This makes the proposed paradigm completed and applicable for all-purpose computing.

The transparent dataflow semantics paradigm is intended for SMP. An ideal SMP hardware has to be UMA capable but unfortunately the current technologies can ensure only NUMA functionality. Therefore we still rely on large-grain seamless VM instructions to achieve higher performance.

Chapter 7

Evaluation

- 7.1. Overview
- 7.2. Test Environment
- 7.3. NAS Parallel Benchmarks
- 7.4. Irregular Test
- 7.5. Summary

7.1. Overview

In this chapter we evaluate the efficiency of BMDFM. We run NAS Parallel Benchmarks 2.3, which are widely recognized as a standard indicator of performance. We also evaluate the dynamic scheduling overhead introduced by the dataflow runtime engine.

The biggest advantage of the dataflow runtime engine, however, is to parallelize dynamic adaptive (irregular) applications, which can not be analyzed statically. We also test this capability as well running an IP core generator based on a VHDL preprocessor.

7.2. Test Environment

The BMDFM system is available for commodity SMP platforms. The BMDFM code is written in ANSI C, so every machine supporting ANSI C and `shmctl()/semctl()` UNIX SVR4 IPC calls should be able to run BMDFM. The official BMDFM web site [14] provides already ported fully multithreaded versions for:

- Intel/Linux/32bit, Intel/FreeBSD/32bit,
- IA-64/Linux/64bit, AMDx86-64/Linux/64bit,
- Alpha/Tru64OSF1/64bit, Alpha/Linux/64bit, Alpha/FreeBSD/64bit,
- PA-RISC/HP-UX/32bit, PA-RISC/HP-UX/64bit,
- SPARC/SunOS/32bit, SPARC/SunOS/64bit,
- MIPS/IRIX/32bit, MIPS/IRIX/64bit,
- RS6000/AIX/32bit, RS6000/AIX/64bit,
- PowerPC/MacOS/32bit,
- Intel/Win32-SFU, Intel/Win32-UWIN
- and a limited single threaded version for Intel/Win32.

For testing purposes we chose the 8-way POWER4 IBM p690 SMP server running AIX 5.1. In all our tests we measure real times to calculate speedups. Additionally, to measure the dynamic scheduling overhead and to count number of the processed instructions we use the following BMDFM embedded facilities, respectively:

- Statistics collector.
- Process logging.

Statistics collector.

Each of the dynamic scheduling processes has its own signal handler that collects CPU execution times via the standard UNIX call `getrusage()`.

```
void resgettime_handl(){
    // ...
    getrusage(RUSAGE_SELF, &res_used);
    // ...
    return "[PROC#%ld]: USRs=%d, USRsus=%d, SYSs=%d, SYSsus=%d.\n";
}
signal(dfmsrv.resgettime_sig, (void (*)(int))&resgettime_handl);
```

The BMDFM server sends a signal to all dynamic scheduling processes and then summarizes the obtained information into a report. We use this report to measure the dynamic scheduling overhead.

```
[SmartHistory]: Assuming {2}: GET COUNT
[SysMsg]: Sending a SIG_GET_TIME to the CPU PROCs...
pipe[CPUPROC#0]: USRs=12, USRus=270000, SYSS=1, SYSus=510000.
pipe[CPUPROC#2]: USRs=13, USRus=270000, SYSS=0, SYSus=580000.
pipe[CPUPROC#4]: USRs=15, USRus=110000, SYSS=3, SYSus=600000.
pipe[CPUPROC#1]: USRs=12, USRus=360000, SYSS=1, SYSus=400000.
pipe[CPUPROC#3]: USRs=12, USRus=870000, SYSS=1, SYSus=500000.
pipe[CPUPROC#5]: USRs=12, USRus=590000, SYSS=0, SYSus=790000.
pipe[CPUPROC#6]: USRs=12, USRus=600000, SYSS=1, SYSus=400000.
pipe[CPUPROC#7]: USRs=12, USRus=150000, SYSS=1, SYSus=380000.
[SysMsg]: Sending a SIG_GET_TIME to the OQ PROCs...
pipe[OQPROC#0]: USRs=0, USRus=120000, SYSS=0, SYSus=410000.
pipe[OQPROC#1]: USRs=0, USRus=160000, SYSS=0, SYSus=460000.
pipe[OQPROC#2]: USRs=0, USRus=140000, SYSS=0, SYSus=600000.
pipe[OQPROC#3]: USRs=0, USRus=180000, SYSS=0, SYSus=520000.
[SysMsg]: Sending a SIG_GET_TIME to the IORBP PROCs...
pipe[IORBPPROC#0]: USRs=0, USRus=50000, SYSS=0, SYSus=110000.
pipe[IORBPPROC#3]: USRs=0, USRus=40000, SYSS=0, SYSus=130000.
pipe[IORBPPROC#6]: USRs=0, USRus=100000, SYSS=0, SYSus=140000.
pipe[IORBPPROC#7]: USRs=0, USRus=40000, SYSS=0, SYSus=90000.
pipe[IORBPPROC#2]: USRs=0, USRus=90000, SYSS=0, SYSus=130000.
pipe[IORBPPROC#1]: USRs=0, USRus=20000, SYSS=0, SYSus=160000.
pipe[IORBPPROC#4]: USRs=0, USRus=160000, SYSS=0, SYSus=180000.
pipe[IORBPPROC#5]: USRs=0, USRus=70000, SYSS=0, SYSus=270000.
[SysMsg]: !!!!! GENERAL BENCHMARKS OF THE PERFORMANCE !!!!!
[Msg]: All times are given in seconds below.
[DFMSrv]: CPU PROCs concurrency factor:
[DFMSrv]: 1.153800000000E+02/1.871000000000E+01=6.166755745591E+00.
[DFMSrv]: Parallelizing index:
[DFMSrv]: 4.370000000000E+00/7.400000000000E-01=5.905405405405E+00.
[DFMSrv]: BM_DFM Server total reached concurrency:
[DFMSrv]: 1.197500000000E+02/1.871000000000E+01=6.400320684126E+00.
[DFMSrv]: Estimation of the CPU PROCs run-time workload:
[DFMSrv]: Abs. time range: min=1.338000000000E+01, max=1.871000000000E+01.
[DFMSrv]: Square root of dispersion = 1.644184828418E+00.
[DFMSrv]: Normalized standard deviation = 4.030557260178E-02 (4.03%).
[DFMSrv]: Estimation of the OQPROCs run-time workload:
[DFMSrv]: Abs. time range: min=5.300000000000E-01, max=7.400000000000E-01.
[DFMSrv]: Square root of dispersion = 8.042853971073E-02.
[DFMSrv]: Normalized standard deviation = 6.210698047160E-02 (6.21%).
[DFMSrv]: Estimation of the IORBPPROCs run-time workload:
[DFMSrv]: Abs. time range: min=1.300000000000E-01, max=3.400000000000E-01.
[DFMSrv]: Square root of dispersion = 7.495832175282E-02.
[DFMSrv]: Normalized standard deviation = 1.191090732984E-01 (11.91%).
```

The CPU time of a process is calculated according to expression (1), where USR_s is user time in seconds, USR_{us} is additional user time in microseconds, SYS_s – system time in seconds and SYS_{us} is additional system time in microseconds. The expressions below show the calculations for:

- CPU PROC concurrency factor (2).
- Parallelization index of the scheduling processes (3).
- Total reached concurrency (4).
- Square root of dispersion (5).
- Normalized standard deviation (6) as an integral characteristic of the load balance.

$$X_i = USR_s + USR_{us} / 1000000 + SYS_s + SYS_{us} / 1000000 \quad (1)$$

$$T_{rc} = \frac{\sum_{i=0}^{K+M+L-1} X_{i(CPU_PROC, OQ_PROC, IORBP_PROC)}}{X_{\max(CPU_PROC, OQ_PROC, IORBP_PROC)}} \quad (4)$$

$$C_f = \frac{\sum_{i=0}^{K-1} X_{i(CPU_PROC)}}{X_{\max(CPU_PROC)}} \quad (2)$$

$$D_x = \sqrt{\frac{1}{N} \sum_{i=0}^N (X_i - \bar{X})^2}, \text{ where } \bar{X} = \frac{1}{N} \sum_{i=0}^N X_i \quad (5)$$

$$P_i = \frac{\sum_{i=0}^{M+L-1} X_{i(OQ_PROC, IORBP_PROC)}}{X_{\max(OQ_PROC, IORBP_PROC)}} \quad (3)$$

$$V_x = \frac{D_x}{X * \sqrt{N}} * 100\% \quad (6)$$

Process logging.

The dataflow engine is able to generate a computational dataflow graph that we use to count the number of processed instructions. All execution processes (CPU PROCs) and I/O ring buffer processes (IORBP PROCs) write their activities into a log file (example fragment is shown below). Every entry in the log file has common clock synchronization. The TaskPort_ID tag unifies the entries belonging to the same application. IORBP PROC entries show data written to the shared memory pool, CPU PROC entries show executed instructions and types of results. Operand references are written in the following format:

<DB_addr:Row:Offset_in_Row>[Index](#Context){Type} /* Var_Names */

```
[CPUPROC#17]: BEGIN at (sec=984321321, usec=96606)
  TaskPort_ID = #2
  FstLispCode = `(SETQ@S MAIN:TMP__000000001@S (OUTF "**** t00: 'A*x^2+B*x+C=0'
    square equations calculation ***\n\n" 0))'
  AddressRefs = <18:0:0>[0](#0){UCH59} /* MAIN:TMP__000000001@S */
  OutputOrder = {0(Ls)}
END_OF_CPUPROC_ENTRY at (sec=984321321, usec=103984)
[CPUPROC#14]: BEGIN at (sec=984321321, usec=99479)
  TaskPort_ID = #2
  FstLispCode = `(SETQ@S MAIN:TMP__000000002@S "How many equations do you want
    to solve: ")'
  AddressRefs = <19:0:0>[0](#0){UCH41} /* MAIN:TMP__000000002@S */
  OutputOrder = {(Ld), 1(Ls)}
END_OF_CPUPROC_ENTRY at (sec=984321321, usec=108828)
[IORBPPROC#18]: BEGIN at (sec=984321323, usec=98113)
  TaskPort_ID = #2
  SrcAddrRefs = <20:0:0>[0](#0) = "5" /* MAIN:TMP__000000003@S */
END_OF_IORBPPROC_ENTRY at (sec=984321323, usec=98797)
[CPUPROC#19]: BEGIN at (sec=984321323, usec=109027)
  TaskPort_ID = #2
  FstLispCode = `(SETQ@I MAIN:NUMB@I (IVAL@S MAIN:TMP__000000003@S))'
  AddressRefs = <14:0:0>[0](#0){SLO8}, <20:0:0>[0](#0){UCH1}
    /* MAIN:NUMB@I, MAIN:TMP__000000003@S */
  OutputOrder = {(Ld)}
END_OF_CPUPROC_ENTRY at (sec=984321323, usec=114636)
[IORBPPROC#21]: BEGIN at (sec=984321323, usec=124816)
  TaskPort_ID = #2
  SrcAddrRefs = <11:0:0>[0](#0) = 1 /* MAIN:I@I */
END_OF_IORBPPROC_ENTRY at (sec=984321323, usec=125041)
```

7.3. NAS Parallel Benchmarks

NAS Parallel Benchmarks [16, 112] were derived from Computational Fluid Dynamics (CFD) code. They were designed to compare the performance of parallel computers and are widely recognized as a standard indicator of computer performance. NPB consists of five kernels and three simulated CFD applications derived from important classes of aerophysics applications. These five kernels mimic the computational core of five numerical methods used by CFD applications. The simulated CFD applications reproduce much of the data movement and computation found in full CFD code.

- **BT** is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions. The resulting systems are a Block-Tridiagonal of 5x5 blocks and are solved sequentially along each dimension.
- **SP** is a simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system has Scalar Pentadiagonal bands of linear equations that are solved sequentially along each dimension.

- **LU** is a simulated CFD application that uses a symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block Lower and Upper triangular systems.
- **FT** contains the computational kernel of a 3D fast Fourier Transform (FFT)-based spectral method. FT performs three one-dimensional (1-D) FFTs, one for each dimension.
- **MG** uses a V-cycle MultiGrid method to compute the solution of the 3-D scalar Poisson equation. The algorithm works continuously on a set of grids. It tests both short and long distance data movement.
- **CG** uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. This kernel tests unstructured grid computations and communications by using a matrix with randomly generated locations of entries.
- **IS** is an Integer Sort kernel.
- **EP** is an Embarrassingly Parallel benchmark. It generates pairs of Gaussian random deviates according to a specific scheme. The goal is to establish the reference point for peak performance of a given platform.

Our starting point is the C implementation of NAS Parallel Benchmarks 2.3, automatically translated into the BMDFM programming model scheme C. The following code fragment demonstrates a typical transformation of parallel loops for the SSOR initialization of the LU benchmark. Each parallel loop is presented in an interleaved fashion and located in the `_wrk()` function. The corresponding `_int()` interface function is a kind of bridge between the C implementation and the virtual machine. Then C functions are called and synchronized on the VM level.

```

void lu_ssor_init_wrk(SLO step, SLO interleave, SLO ISIZ1, SLO ISIZ2,
DFL ****a, DFL ****b, DFL ****c, DFL ****d){
  SLO i,j,k,m;
  for(i=step;i<ISIZ1;i+=interleave)
    for(j=0;j<ISIZ2;j++)
      for(k=0;k<5;k++)
        for(m=0;m<5;m++)
          a[i][j][k][m]=b[i][j][k][m]=c[i][j][k][m]=d[i][j][k][m]=0.;
  return;
}

void lu_ssor_init_int(ULO *dat_ptr, struct fastlisp_data *ret_dat){
  SLO step,interleave,ISIZ1,ISIZ2;
  DFL ****a,****b,****c,****d;
  ret_ival(dat_ptr,&step);
  ret_ival(dat_ptr+1,&interleave);
  ret_ival(dat_ptr+2,&ISIZ1);
  ret_ival(dat_ptr+3,&ISIZ2);
  ret_ival(dat_ptr+4,(SLO*)&a);
  ret_ival(dat_ptr+5,(SLO*)&b);
  ret_ival(dat_ptr+6,(SLO*)&c);
  ret_ival(dat_ptr+7,(SLO*)&d);
  if(noterror){
    lu_ssor_init_wrk(step,interleave,ISIZ1,ISIZ2,a,b,c,d);
    ret_dat->single=1;
    ret_dat->type='I';
    ret_dat->value.ival=0;
  }
  return;
}

INSTRUCTION_STRU INSTRUCTION_SET[]={
  // ...
  {"LU_SSOR_INIT",8,'I',"IIIIIIII",&lu_ssor_init_int},
  // ...
};
const ULO INSTRUCTIONS=sizeof(INSTRUCTION_SET)/sizeof(INSTRUCTION_STRU);

(setq a (| sync_a a))
(setq b (| sync_b b))
(setq c (| sync_c c))
(setq d (| sync_d d))
(for thread 1 1 threads
  (setq sync_a (| sync_a
    (lu_ssor_init (-- thread) threads ISIZ1 ISIZ2 a b c d)
  ))
)
(setq sync_b (setq sync_c (setq sync_d sync_a)))
(setq a (| sync_a a))
(setq b (| sync_b b))
(setq c (| sync_c c))

```

```
(setq d (| sync_d d))
```

Table 7.1 and Figure 7.1 show the execution times and speedups of a 64-bit BMDFM running the NAS Parallel Benchmarks (version 2.3, class A and class C) on an 8-way POWER4 IBM p690 SMP server. BMDFM is configured on scheme C for N_CPUPROC = 8, N_IORBPPROC = 8 and N_OQPROC = 4.

Benchmark	Class	Baseline time / s	BMDFM time / s	Speedup
EP	A	140.00	17.90	7.82
	C	2266.00	286.47	7.91
CG	A	10.36	2.57	4.03
	C	2556.00	339.58	7.53
IS	A	9.16	1.78	5.15
	C	361.42	58.90	6.14
MG	A	20.18	3.31	6.10
	C	516.12	70.14	7.36
FT	A	27.95	3.94	7.09
	C	1575.56	215.97	7.30
BT	A	860.84	159.54	5.40
	C	20764.48	3039.68	6.83
SP	A	410.32	115.50	3.55
	C	9888.60	1708.99	5.79
LU	A	490.38	91.07	5.38
	C	13400.40	1798.41	7.45

Table 7.1. Execution times and speedups for NAS PB 2.3 (classes A, C) on 8-way SMP

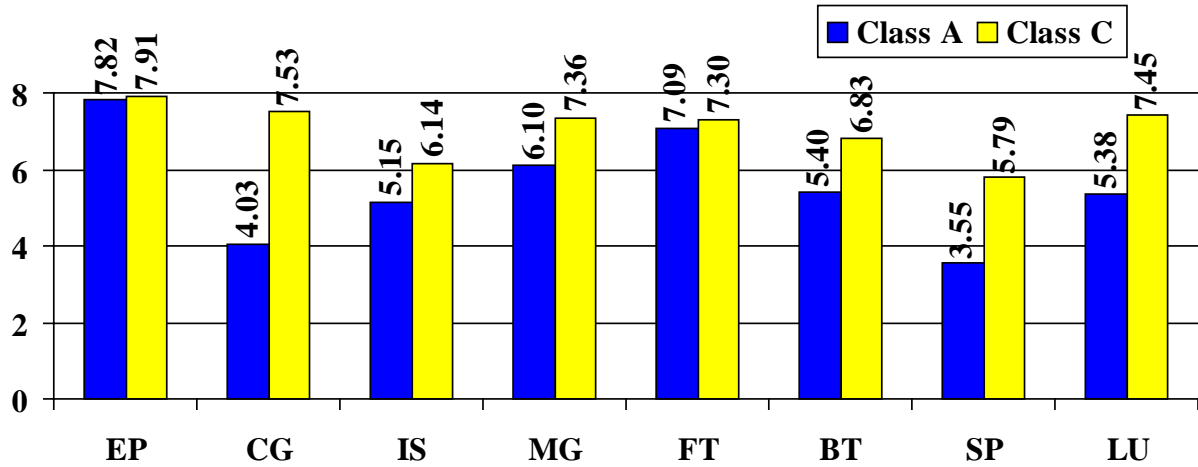


Figure 7.1. Speedups for NAS PB 2.3 (classes A, C) on 8-way SMP

These experimental results demonstrate good average speedup and prove the efficiency of the proposed SMP dataflow hybrid architecture. Performance tends to be better for the NAS PB class C as this class operates on bigger arrays that influences larger granularity of the processed coarse-grain instructions.

We also estimated the dynamic scheduling overhead of the dataflow runtime engine, collecting CPU times used by the dynamic scheduling processes. Table 7.2 summarizes the number of instructions processed in each test and gives CPU times spent for their dynamic scheduling. Then we calculate normalized scheduling overhead (SO) per instruction. This parameter differs for all tests, which can be explained by the different complexity of the algorithms and dependencies among the processed instructions. The average value of the normalized scheduling overhead is 27 μ s per virtual machine instruction, which is considerably less compared to the computational weight of the generated instructions (5.3% for class A and 1.1% for class C – weight of instructions is bigger in class C). Therefore we conclude the proposed SMP dataflow hybrid architecture has negligible dynamic scheduling overhead.

Benchmark	Class	Number of Instructions	Scheduling Overhead (SO) time / s	SO per Instruction time / μ s	SO/Baseline ratio %
EP	A	312	0.007	22.436	0.00500
	C	312	0.005	16.026	0.00022
CG	A	58012	1.132	19.513	10.92664
	C	274372	6.265	22.834	0.24511
IS	A	1708	0.067	39.227	0.73144
	C	1708	0.054	31.616	0.01494
MG	A	21782	0.935	42.925	4.63330
	C	100049	4.234	42.319	0.82035
FT	A	2954	0.093	31.483	0.33274
	C	8994	0.234	26.017	0.01485
BT	A	2140724	52.675	24.606	6.11902
	C	5154915	202.854	39.352	0.97693
SP	A	11726139	45.031	3.840	10.97461
	C	27680328	530.763	19.175	5.36742
LU	A	2227548	42.739	19.187	8.71549
	C	5632798	170.720	30.308	1.27399
Average	A			25.402	5.30478
	C			28.456	1.08923
	Average			26.929	(3.1970)

Table 7.2. Number of instructions and scheduling overhead for NAS PB 2.3 (classes A, C) on 8-way SMP

7.4. Irregular Test

The biggest advantage of the dataflow engine is to parallelize dynamic adaptive (irregular) applications, which can not be analyzed statically. We also test this capability as well running an IP core generator based on VHDL preprocessor.

The idea of the IP core generator is to have a library of configurable VHDL files (.vhl files stand for VHDL + LISP). Each such a file consists of native VHDL code mixed with ‘macros’, which are written in the VM language. The VHDL preprocessor searches for all macros, evaluates them and substitutes the macros with calculated result, in other words VHDL preprocessor converts .vhl files from the source directory to .vhd files in a target directory. After a single file is processed, the VHDL preprocessor performs a second pass, searching for all used components in the “COMPONENT” statements and diving deeply into the hierarchy of referenced components recursively. When all of the hierarchy is processed the target directory will contain pure VHDL code of the generated IP core.

Such a non-trivial test is very interesting from the parallelization point of view because the dataflow engine has to be able to exploit inter-procedural and cross-conditional parallelism at runtime.

The following code fragments demonstrate configurable VHDL code before and after preprocessing (wordwidth = 32), which do not reference any hierarchy of components.

Initial configurable .vhl code
<pre> LIBRARY IEEE; USE IEEE.std_logic_1164.ALL; USE IEEE.std_logic_signed.ALL; USE IEEE.std_logic_arith.ALL; ENTITY rom IS PORT(in_adr : IN STD_LOGIC_VECTOR(1 DOWNTO 0); out_d : OUT STD_LOGIC_VECTOR(`(-- wordwidth)` DOWNTO 0)); END rom; ARCHITECTURE str OF rom IS BEGIN run: PROCESS(in_adr) BEGIN BEGIN `(progn </pre>

```

(defun coeff_gen
  (progn
    (setq wordwidth $1)
    (setq i $2)
    (setq n $3)
    (setq ret_line "\"0")
    (setq weight 0)
    (for j 2 1 wordwidth (progn
      (setq weight (<< weight 1))
      (setq weight (++ weight))
    ))
    (setq coeff (ival (*. weight (cos (/ (*. (pi) (++ (* 2 i))) (* 2 n))))))
    (setq j (<< 1 (- wordwidth 2)))
    (while (>= j 1) (progn
      (setq ret_line (cat ret_line (if (> (/ coeff j) 0) "1" "0")))
      (setq coeff (% coeff j))
      (setq j (>> j 1))
    ))
    (cat (cat ret_line "\"; -- R") (cat (str i) (cat "(" (cat (str n) ")"))))
  )
)
(outf " IF in_adr = \"00\" THEN\n" 0)
(outf " out_d <= %s\n" (coeff_gen wordwidth 0 8))
(outf " ELSIF in_adr = \"01\" THEN\n" 0)
(outf " out_d <= %s\n" (coeff_gen wordwidth 1 8))
(outf " ELSIF in_adr = \"10\" THEN\n" 0)
(outf " out_d <= %s\n" (coeff_gen wordwidth 3 8))
(outf " ELSIF in_adr = \"11\" THEN\n" 0)
(outf " out_d <= %s\n" (coeff_gen wordwidth 2 8))
" END IF;\n"
)' END PROCESS run;
END str;

```

Preprocessed target pure .vhd code

```

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_signed.ALL;
USE IEEE.std_logic_arith.ALL;

ENTITY rom IS
  PORT(in_adr : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        out_d : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
END rom;

ARCHITECTURE str OF rom IS
BEGIN
  run: PROCESS(in_adr)
  BEGIN
    IF in_adr = "00" THEN
      out_d <= "01111101100010100101111100111110"; -- R0(8)
    ELSIF in_adr = "01" THEN
      out_d <= "01101010011011011001100010100011"; -- R1(8)
    ELSIF in_adr = "10" THEN
      out_d <= "00011000111110001011100000111100"; -- R3(8)
    ELSIF in_adr = "11" THEN
      out_d <= "01000111000111001101100111001110"; -- R2(8)
    END IF;
  END PROCESS run;
END str;

```

Now let us have a look at the next code fragments of the top level VHDL model. After the preprocessing (wordwidth = 32 and use_rom = true) the target code will contain COMPONENT rom, which recursively forces to preprocess previously discussed code of the “rom” component.

Initial configurable .vhl code

```

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_signed.ALL;
USE IEEE.std_logic_arith.ALL;

ENTITY device IS
  PORT(a,b : IN STD_LOGIC_VECTOR('(if use_rom 1 (-- wordwidth))' DOWNTO 0);
        s: OUT STD_LOGIC_VECTOR('(-- wordwidth)' DOWNTO 0));
END device;

ARCHITECTURE str OF device IS
'(if use_rom

```

```

        (progn
          (outf " COMPONENT rom\n" nil)
          (outf " PORT(in_adr : IN STD_LOGIC_VECTOR(1 DOWNTO 0);\n" nil)
            (outf " out_d : OUT STD_LOGIC_VECTOR(%ld DOWNTO 0);\n"
              (-- wordwidth))
            (outf " END COMPONENT;\n" nil)
            (outf " SIGNAL opa, opb : STD_LOGIC_VECTOR(%ld DOWNTO 0);\n"
              (-- wordwidth))
            ""
          )
        )
      )'BEGIN
    `(if use_rom
      (progn
        (outf " u0: rom\n" nil)
        (outf " PORT MAP(in_adr => a, out_d => opa);\n" nil)
        (outf " u1: rom\n" nil)
        " PORT MAP(in_adr => b, out_d => opb);\n"
      )
    )
    )' s <= `(if use_rom "op" "")`a + `(if use_rom "op" "")`b;
  END str;

```

Preprocessed target pure .vhd code

```

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_signed.ALL;
USE IEEE.std_logic_arith.ALL;

ENTITY device IS
  PORT(a,b : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        s: OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
END device;

ARCHITECTURE str OF device IS
  COMPONENT rom
    PORT(in_adr : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
          out_d : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
  END COMPONENT;
  SIGNAL opa, opb : STD_LOGIC_VECTOR(31 DOWNTO 0);
BEGIN
  u0: rom
    PORT MAP(in_adr => a, out_d => opa);
  u1: rom
    PORT MAP(in_adr => b, out_d => opb);
  s <= opa + opb;
END str;

```

The IP core generator (VHDL preprocessor) itself is also implemented in pure VM language. It consists of five functions:

- (rd_file) reads a file. It takes a file name as an argument and returns the file contents.
- (wr_file) writes a file. It takes the file name and file contents as arguments.
- (fst_lsp) encapsulates the (mapcar) function. It takes an expression of the VM language, evaluates it through the (mapcar) and returns the result of the calculation.
- (process_file) takes a raw file and returns a preprocessed file where all macros are evaluated.
- (process_hierarchy) is a recursive function that calls (process_file) first and invokes itself recursively for all found COMPONENT statements. Function main calls (process_hierarchy) for the top VHDL model.

The first three functions are trivial, so we explain only the (process_file) and (process_hierarchy) functions below:

```

(process_file) real code
(defun process_file
  (progn
    (setq flp_statement $1) # parameters like (setq wordwidth 32) (setq use_rom 1)
    (setq file_contents $2) # raw file contents.
    (setq new_contents "") # new contents to concatenate.
    (while (len file_contents) # iterate through the raw contents.
      (if (setq pos (at "" file_contents)) # find a macro.
        (progn
          (setq new_contents (cat new_contents (left file_contents (idecr pos))))
          (setq file_contents (right1 file_contents pos))
          (setq pos (at "" file_contents))
          (setq flp_statement1 (cat flp_statement (left file_contents (idecr pos))))
        )
      )
    )
  )

```


1	223 running single-threadedly		1.00
2		131.18	1.67
4		69.69	3.20
8		34.31	6.50

Table 7.3. Execution times and speedups for irregular test

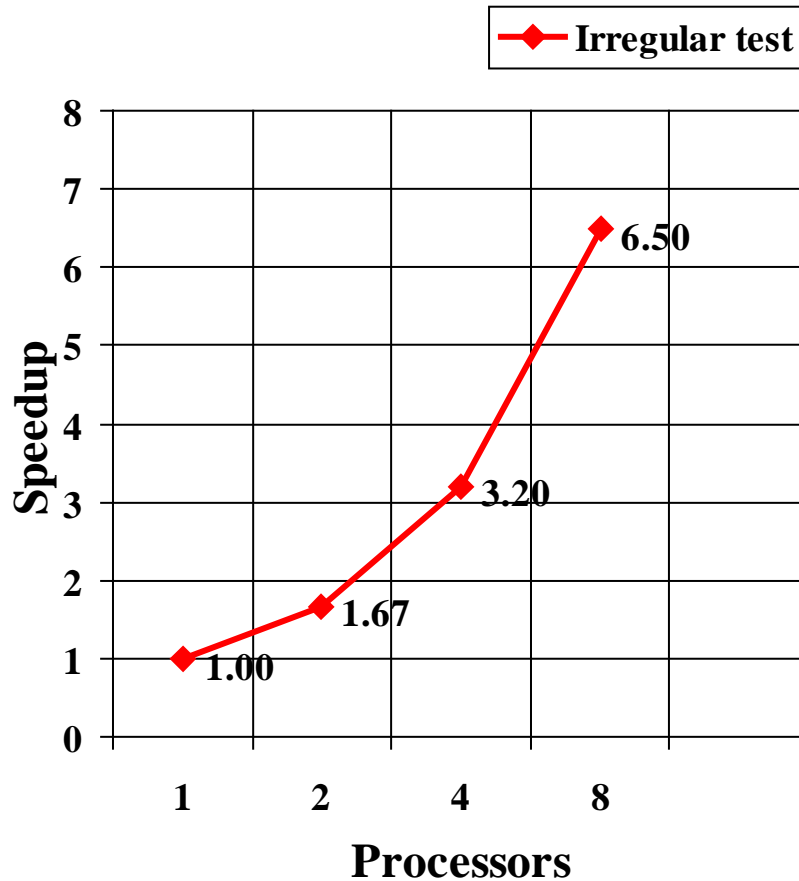


Figure 7.2. Speedups for irregular test

As we can see, the dataflow engine demonstrates good runtime parallelization capabilities, taking into account that the sequential application is written in a conventional programming style and does not require any parallelization/synchronization directives.

7.5. Summary

Having tested the performance of BMDFM on the 8-way POWER4 IBM p690 SMP server we have concluded that in general it performs very well, demonstrating nearly linear scalability on both numeric processing and irregular applications.

We have also estimated the dynamic scheduling overhead, which is 27 μ s per virtual machine instruction. This value is considerably less compared to computational weight of the generated instructions (5.3% for class A and 1.1% for class C of NAS Parallel Benchmarks 2.3 – weight of instructions is bigger in class C).

The dataflow engine also demonstrates good runtime parallelization capabilities, running dynamic adaptive (irregular) applications, which can not be analyzed statically. We also test this capability as well running an IP core generator based on VHDL preprocessor.

Chapter 8

Conclusion

8.1. Overall Summary

8.2. Future Directions

8.1. Overall Summary

Runtime parallelization definitely provides a wider range of possibilities compared to compile-time methods. Compilers cannot apply many interesting optimizations that depend on knowledge of dynamic information. Compile-time optimizations cannot be applied to situations where the time it takes to complete an operation varies at runtime. Additionally, compilers can perform only limited inter-procedural and cross-conditional optimizations because they often cannot determine which way a conditional will go or cannot optimize across a function call.

To complement existing compiler-optimization methods we have proposed a hybrid dataflow parallelization environment BMDFM that creates a data-dependence graph and exploits parallelism of a user application program at run time.

Our architectural approach has the following important features that are not present in known runtime parallelization projects:

- The dataflow runtime engine is not aggressively optimized for the applications in some specific area such as numeric processing, for example. It can solve inter-procedural and cross-conditional dependencies as well.
- The dynamic scheduling subsystem is decentralized and is executed in parallel on the same multiprocessors that run the application itself. This approach eliminates the situation where the task scheduling becomes a bottleneck of the entire computing process.
- An application comprises the conventional virtual machine language and classical C. There is no special language to control dataflow. The application program itself controls dataflow fully automatically and transparently.
- From the point of view of dataflow programming our approach excludes the problem of a single assignment paradigm. We think that our way of dataflow programming with a conventional algorithmic language can remove the known gap of missing programming methodology for dataflow.

It is known that the main weakness of dataflow is relatively high dynamic scheduling overhead because each dynamic instruction requires dynamic operand matching. We spent a great deal of effort designing and optimizing our dataflow engine to reduce this overhead. We would especially like to highlight the multiple context data structuring, speculative tagging of instructions and parallel load of clusters as our main contributions in this field. Additionally, we have estimated program and time complexities of the proposed dataflow dynamic scheduling subsystem.

Having tested the performance of BMDFM on the 8-way POWER4 IBM p690 SMP server, we have concluded that in general it performs very well, demonstrating nearly linear scalability on both numeric processing and irregular applications.

Indeed, the main advantage of the proposed architecture is to provide a conventional programming paradigm at the top level. We call this ability of BMDFM transparent dataflow semantics, the convenience of which justifies all the efforts we spent designing the complex architecture behind it. We think that because the complexity is hidden the transparent dataflow semantics is a key point defining applicability of the BMDFM system.

* * *

The BMDFM is publicly available for commodity SMP platforms. BMDFM can be understood as a virtual machine, which provides a conventional functional programming model using transparent dataflow semantics with negligible dynamic scheduling overhead.

We believe that our approach is a big step toward exploring a better parallel programming/compiling technology. Thanks to the effective architectural combination of SMP and dataflow we were able to consider a much simpler parallelization technique for programmers.

8.2. Future Directions

Several directions for follow-up work have been identified as this thesis was written.

- Further improvements can be applied to some of the techniques that we have already used in a limited fashion. We also would consider aggressive optimization of the system for certain SMP architectures such as a popular scalable ccNUMA.
- One other direction is to add a pure C or Fortran interface to the system. This work has already been started in regards to designing the external translators from high-level languages into the VM/C language. Moreover, we have already used them in this work for testing but we did not describe this functionality as it is out of the scope of the thesis.

List of Defined Terms

ARRAYBLOCK_SIZE – size of a minimal data chunk, in which data is allocated in the shared memory pool.

Artificial temporary variable – a variable introduced by the static scheduler to resolve data dependencies in the dataflow engine.

Binary Modular DataFlow Machine (BMDFM) – a hybrid dataflow runtime parallelization environment for shared memory multiprocessors.

BMDFM server – the main BMDFM module, which starts and shuts down the dataflow engine.

Common semaphores – an array of main semaphores that synchronizes the dataflow engine.

Context – a unique number that specifies a copy of data in the tagged-token dataflow engine.

Context data – a piece of data within the specified context.

Control sequence – a code for the front-end virtual machine automatically generated from the user application. Control sequence controls uploading of the marshaled clusters into the dataflow engine.

CPU PROC – a BMDFM CPU executing/scheduling process executing ready instructions and performing garbage collection in the shared memory pool.

Dataflow engine – multithreaded dataflow engine that creates a data-dependence graph and exploits parallelism of a user application programs at run time.

Distributed semaphores – semaphores that are interleaved along the shared memory pool to control multiple resources of the same type.

Dynamic scheduler – multiple processes performing dataflow emulation and scheduling the dataflow engine.

Front-end virtual machine – a von Neumann control machine for the dataflow engine. The control machine does not execute the byte code of an application but it uploads the marshaled clusters dynamically to the dataflow engine.

Function – an atomic unit for the virtual machine.

Input/Output Ring Buffer Ports (IORBP) – a data structure in the shared memory pool to which the front-end virtual machine uploads the marshaled clusters and data. Consists of the separate cells.

Instruction – a seamless atomic unit for the dataflow engine that is a BMDFM byte code fragment.

IORBP PROC – a BMDFM IORBP scheduling process moving clustered data and instructions to DB and OQ respectively and performing garbage collection in the shared memory pool.

Loader/listener pair – an external process performing the static scheduling of the user application and then emulating the front-end virtual machine.

Local function directory – a directory for all functions of the marshaled cluster pointing to the TCZ function directory.

Local variable directory – a directory for all variables of the marshaled cluster pointing to the data buffer.

Machine instruction database – a registry for all available VM seamless functions.

Marshaled cluster – an atomic data chunk that is uploaded to the dataflow engine seamlessly. It consists of local variable directory and local function directory.

N_CPUPROC – number of BMDFM CPU executing/scheduling processes.

N_IORBP – number of input/output ring buffer ports.

N_IORBPPROC – number of BMDFM IORBP scheduling processes.

N_OQPROC – number of BMDFM OQ scheduling processes.

Operation Queue (OQ) – a data structure in the shared memory pool where all instructions are stored. Consists of the separate cells.

OQ PROC – a BMDFM OQ scheduling process performing the speculative tagging and garbage collection in the shared memory pool.

Port – a single entity in the trace plugging area associated with one connected external tracer.

PROC – a BMDFM process.

PROC local memory – a local storage for the seamless coarse-grain user defined functions.

PROC stat – a BMDFM auxiliary process collecting statistic information on the dataflow engine.

Q_DB – size of data buffer.

Q_IORBP – size of input/output ring buffer ports.

Q_OQ – size of operation queue.

Scheme A – a programming model with fine-granularity of parallelism.

Scheme B – a programming model with coarse-grain UDFs that are defined on VM level.

Scheme C – a programming model with coarse-grain UDFs that are defined in C language.

Shared memory pool – a common storage shared by all BMDFM processes.

SHMEM_POOL_BANKS – number of memory banks in the shared memory pool.

SHMEM_POOL_SIZE – shared memory pool size.

Socket – a single entity in the task connection zone associated with one connected external task loader/listener pair.

Speculative tagging – a dynamic scheduling algorithm that tags instructions to be ready for execution. Speculative mechanism reduces the dynamic scheduling overhead.

Static scheduler – a process that converts the user application into the control sequence and marshaled clusters.

Task Connection Zone (TCZ) – a structure in the shared memory pool where all connected external task loader/listener pairs are registered.

TCZ function directory – a shared memory pool storage for all instructions of the connected user application.

TCZ output queue – a shared memory pool structure where the output stream is ordered after out-of-order processing.

Trace Plugging Area (TPA) – a structure in the shared memory pool where all connected external tracers are registered.

Tracer – an external process for debugging the out-of-order execution in the dataflow engine.

Transparent dataflow semantics – a paradigm that provides a conventional programming style on top of a dataflow machine.

Universal structure – a predefined VM data structure to store variables that enables changing data types dynamically and having a single value or an array with different types of members in order to support lists and trees.

User Defined Function (UDF) – a function defined on VM level or in C language.

Virtual Machine (VM) – a runtime kernel that runs BMDFM byte code. A single threaded VM runs in a protected address space. The multithreaded VM is modified slightly to store all dynamic data in the shared memory pool.

References

- [1] W.B.Ackerman, J.B.Dennis. VAL - A Value-Oriented Algorithmic Language. Technical Report 218, Laboratory for Computer Science, MIT, 1979.
- [2] T.Agerwala, Arvind, Data flow systems, IEEE Computer 15 (Feb. 1982), pp. 10-13.
- [3] H.Akkary, M.Driscoll. A Dynamic Multithreading Processor. In MICRO-31, December 1998.
- [4] S.P.Amarasinghe, J.M.Anderson, M.S.Lam, C.W.Tseng, The SUIF Compiler for Scalable Parallel Machines, Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing, July, 1995.
- [5] J.A.M.Anderson, Automatic Computation and Data Decomposition for Multiprocessors, Technical Report CSL-TR-97-719, Computer Systems Laboratory, Dept. of Electrical Eng. and Computer Sc., Stanford University, 1997.
- [6] T.Anderson, B.Bershad, E.Lazowska, H.Levy, Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, Proceedings of the 13th. ACM Symposium on Operating System Principles (SOSP), October 1991.
- [7] T.E.Anderson, FastThreads User's Manual, Department of Computer Science and Engineering, University of Washington, January 1990.
- [8] T.E.Anderson, E.D.Lazowska, Quartz: A Tool for Tuning Parallel Program Performance, Department of Computer Science and Engineering, University of Washington, September 1989.
- [9] J.M.Anderson, S.P.Amarasinghe, M.S.Lam. Data and Computation Transformations for Multiprocessors. Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing, Jul. 1995.
- [10] Arvind, D.E.Culler, Dataflow architectures, Ann. Review in Comput. Sci. 1 (1986), pp. 225-253.
- [11] Arvind, V.Kathail, A multiple processor dataflow machine that supports generalized procedures, Proc. 8th ISCA, May 1981, pp. 291-302.
- [12] Arvind, R.S.Nikhil, Executing a program on the MIT tagged-token dataflow architecture, Lect. Notes Comput. Sc. 259 (1987), pp. 1-29.
- [13] E.Ayguade, X.Martorell, J.Labarta, M.Gonzalez, N.Navarro. Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study. ICPP'99, Sep. 1999.
- [14] BMDFM The Official Site. <http://www.bmdfm.de>
- [15] J.Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Comm. ACM 21 (1978), pp. 613-641.
- [16] D.Bailey, T.Harris, W.Saphir, R.Wijngaart, A.Woo, M.Yarrow, The NAS Parallel Benchmarks 2.0, Technical Report NAS-95-020, NASA, December 1995.
- [17] U.Banerjee. Loop Parallelization. Kluwer Academic Pub., 1994.
- [18] P.M.C.C.Barahona, J.R.Gurd, Simulated performance of the Manchester multi-ring dataflow machine, Proc. 2nd ICPC, Sep. 1985, pp. 419-424.
- [19] P.M.C.C.Barahona, J.R.Gurd, Processor allocation in a multi-ring dataflow machine, J. Parall. Distr. Comput. 3 (1986), pp. 67-85.
- [20] U.Banerjee. Dependence Analysis for Supercomputing. Kluwer Pub., 1989.
- [21] M.Beck, T.Ungerer, E.Zehender, Classification and performance evaluation of hybrid dataflow techniques with respect to matrix multiplication, Proc. GI/ITG Workshop PARS'93, Apr. 1993, pp. 118-126.
- [22] B.Bershad, E.Lazowska, H.Levy, Presto: A System for Object-oriented Parallel Programming, Software - Practice and Experience, vol. 18, no. 8, pp. 713-732, 1988.
- [23] L.Bic, M.Al-Mouhamed, The EM-4 under implicit parallelism, Proc. 1993 Int'l Conf. Supercomputing, Jul. 1993, pp. 21-26.
- [24] W.Blume, R.Eigenmann, K.Faigin, J.Grout, J.Hoeinger, D.Padua, P.Petersen, W.Pottenger, L.Rauchwerger, P.Tu, S.Weatherford. Effective automatic parallelization with Polaris. International Journal of Parallel Programming, May 1995.
- [25] R.D.Blumofe, C.F.Joerg, B.C.Kuzmaul, C.E.Leiserson, K.H.Randall, Y.Zhou, Cilk: An Efficient Multithreaded Runtime System, Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95), Santa Barbara, California, July 19-21, 1995.
- [26] A.P.W.Boehm, Y.M.Teo, Resource management in a multi-ring dataflow machine, Proc. CONPAR88, Sep. 1988, pp. B 191-200.

- [27] C.J.Brownhill, A.Nicolau, S.Novack, C.D.Polychronopoulos. Achieving Multi-level Parallelization. Proc. Of ISHPC'97, Nov. 1997.
- [28] R.Buehrer, K.Ekanadham, Incorporating data flow ideas into von Neumann processors for parallel processing, IEEE Trans. Computers C-36 (1987), pp. 1515-1522.
- [29] A.J.Catto, J.R.Gurd, Resource management in dataflow, Proc. Conf. Functional Programming Languages and Comput. Arch., Oct. 1981, pp. 77-84.
- [30] A.J.Catto, J.R.Gurd, C.C.Kirkham, Non-deterministic dataflow programming, Proc. 6th ACM European Conf. Comput. Arch., Apr. 1981, pp. 435-444.
- [31] R.Chandra, A.Gupta, J.L.Hennessy, Data Locality and Load Balancing in COOL, Fourth ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming (PPoPP), pp. 249-259, May 1993.
- [32] R.Chandra, A.Gupta, J.L.Hennessy, Integrating Concurrency and Data Abstraction in the COOL Parallel Programming Language, Technical Report CSL-TR-92-511, Computer Systems Laboratory, Stanford University, February 1992.
- [33] A.Charlesworth, STARFIRE: Extending the SMP Envelope, IEEE Micro, Jan/Feb 1998.
- [34] A.Chien, V.Karamcheti, J.Plevyak, The Concert System - Compiler and Runtime Support for Efficient Fine-grained Concurrent Object-oriented Programs, Department of Computer Science, University of Illinois, Urbana, IL, Tech. Rep. UIUCDCS-R-93-1815, 1993.
- [35] J.Chow, W.L.Harrison III, Switch Stacks: A Scheme for Micro-tasking Nested Parallel Loops, Center for Supercomputing Research and Development (CSR), University of Illinois at Urbana-Champaign, 1990.
- [36] M.Cintra, J.F.Martinez, J.Torrellas, Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors, Proceedings of the 27th Annual International Symposium of Computer Architecture, pp. 13-24, Vancouver, BC, June 2000.
- [37] R.P.Colwell, R.L.Steck, A 0.6um BiCMOS processor with dynamic execution, Proc. Intl. Solid State Circuits Conf., Feb. 1995.
- [38] D.Cortessi, A.Evans, W.Ferguson, J.Hartman, Topics in IRIX Programming, Doc. num. 007-2478-004, Silicon Graphics, Inc., <http://techpubs.sgi.com>, 1996.
- [39] D.Cortessi, A.Evans, W.Ferguson, J.Hartman, Topics in IRIX Programming, Doc. num. 007-2478-006, Silicon Graphics, Inc., <http://techpubs.sgi.com>, 1998.
- [40] D.Craig, An Integrated Kernel- and User-Level Paradigm for Efficient Multiprogramming Support, M.S. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.
- [41] D.E.Culler, S.Goldstein, K.E.Schauser, T.vonEicken, TAM - A compiler controlled threaded abstract machine, J. Parall. Distr. Comput., 18 (1993), pp.347-370.
- [42] D.E.Culler, G.M.Papadopoulos, The explicit token store, J. Parall. Distr. Comput. 10 (1990), pp. 289-308.
- [43] D.E.Culler, A.Sah, K.E.Schauser, T.vonEicken, J.Wawrzynek, Fine-grain parallelism with minimal hardware support: A compiler-controlled Threaded Abstract Machine, Proc. 4th Int'l Conf. Arch. Support for Programming Languages and Operating Systems, April 1991, pp. 164-175.
- [44] L.Dagum, R.Menon. OpenMP: An Industry Standard API for Shared Memory Programming. IEEE Computational Science & Engineering, 1998.
- [45] J.B.Dennis, First version of a data-flow procedure language, Lect. Notes Comput. Sc. 19 (1974), pp. 362-376.
- [46] J.B.Dennis, The varieties of data flow computers, Proc. 1st Int'l Conf. Distributed Comp. Sys., Oct. 1979, pp. 430-439.
- [47] J.B.Dennis, Dataflow computation: A case study, Computer architecture - Concepts and systems (V.M.Milutinovic, ed.), North-Holland, 1988, pp. 354-404.
- [48] J.B.Dennis, D.P.Misunas, A preliminary architecture for a basic data-flow processor, Proc. 2nd ISCA, Jan. 1975, pp. 126-132.
- [49] K.Diefendorff, Power4 Focuses on Memory Bandwidth, Microprocessor Report, October 6, 1999, pp. 11-17.
- [50] Digital Equipment Corporation / Compaq Computer Corporation, AlphaServer 8x00 Technical Summary, http://www.digital.com/alphaserver/alphasrv8400/8x00_summ.html, 1999.
- [51] Digital Equipment Corporation / Compaq Computer Corporation, AlphaServer GS60/ GS140 and 8200/8400 Systems, technical summary, <http://www.digital.com/alphaserver/products.html>, 1999.
- [52] Digital Equipment Corporation / Compaq Computer Corporation, Digital UNIX: Assembly Language Programmer's Guide, Maynard, Massachusetts, March 1996.
- [53] Digital Equipment Corporation / Compaq Computer Corporation, Digital UNIX: Calling Standard for Alpha Systems, Maynard, Massachusetts, March 1996.

- [54] Digital Equipment Corporation / Compaq Computer Corporation, Digital UNIX: Guide to DEC threads, Maynard, Massachusetts, December 1997.
- [55] Digital Equipment Corporation / Compaq Computer Corporation, Digital UNIX: Programmer's Guide, Maynard, Massachusetts, March 1996.
- [56] J.H.Edmondson, P.Rubinfeld, R.Preston, V.Rajagopalan, Superscalar Instruction Execution in the 21164 Alpha Microprocessor, IEEE Micro, 15(2), pp. 33-43, April 1995.
- [57] R.Eigenmann, J.Hoeflinger, D.Padua. On the Automatic Parallelization of the Perfect Benchmarks. IEEE Trans. On parallel and distributed systems, 9(1), Jan. 1998.
- [58] D.R.Engler, G.R.Andrews, D.K.Lowenthal, Filaments: Efficient Support for Fine-Grain Parallelism, Technical Report 93-13a, Department of Computer Science, University of Arizona, Tucson, 1993.
- [59] P.Evripidou, J.L.Gaudiot, The USC decoupled multilevel data-flow execution model, Advanced Topics in Data-Flow Computing (J.L.Gaudiot, L.Bic, eds.), Prentice Hall, 1991, pp. 347-379.
- [60] J.T.Feo, D.C.Cann, R.R.Oldehoeft. A report on the Sisal language project. J. Parallel Distrib. Comput., 10:349-366, 1990.
- [61] M.Franklin, G.S.Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. IEEE Transactions on Computers, 45(5), May 1996.
- [62] J.L.Gaudiot, L.Bic, Advanced Topics in Data-Flow Computing, Prentice Hall, 1991.
- [63] A.Geist, A.Beguelin, J.Dongarra, W.Jiang, B.Manckek, V.Sunderam, PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Network Parallel Computing, MIT Press, 1994.
- [64] M.Girkar, M.R.Haghighat, P.Grey, H.Saito, N.Stavrakos, C.D.Polychronopoulos, Illinois-Intel Multithreading Library: Multithreading Support for Intel Architecture Based Multiprocessor Systems, Intel Technology Journal, Q1 issue, February 1998.
- [65] M.Girkar, C.Polychronopoulos. Optimization of Data/Control Conditions in Task Graphs. Proc. 4th Workshop on Languages and Compilers for Parallel Computing, Aug. 1991.
- [66] J.R.W.Glauert, J.R.Gurd, C.C.Kirkham, Evolution of dataflow architecture, Proc. IFIP WG 10.3 Workshop on Hardware Supported Implementation on Concurrent Languages in Distributed Systems, March 1984, pp. 1-18.
- [67] J.R.W.Glauert, J.R.Gurd, C.C.Kirkham, I.Watson, The dataflow approach, Distributed Computing (F.B.Chambers, D.A.Duce, G.P.Jones, eds.), Academic Press, 1984, pp. 1-53.
- [68] S.Gopal, T.Vijaykumar, J.Smith, G.Sohi. Speculative Versioning Cache. In Proceedings of the Fourth International Symposium on High-Performance Computer Architecture, February 1998.
- [69] K.P.Gostelow, R.E.Thomas, A view of dataflow, Proc. National Comp. Conf., Jun. 1979, pp. 629-636.
- [70] K.P.Gostelow, R.E.Thomas, Performance of a simulated dataflow computer, IEEE Trans. Computers C-29 (1980), pp. 905-919.
- [71] V.G.Grafe, J.E.Hoch, The Epsilon-2 multiprocessor system, J. Parall. Distr. Comput. 10 (1990), pp. 309-318.
- [72] V.G.Grafe, J.E.Hoch, G.S.Davidson, V.P.Holmes, D.M.Davenport, K.M.Steele, The Epsilon project, Advanced Topics in Data-Flow Computing (J.L.Gaudiot, L.Bic, eds.), Prentice Hall, 1991, pp. 175-205.
- [73] M.Gupta, R.Nim. Techniques for Speculative Run-Time Parallelization of Loops. In Supercomputing '98, November 1998.
- [74] M.R.Haghighat, C.D.Polychronopoulos. Symbolic Analysis for Parallelizing Compilers. Kluwer Academic Publishers, 1995.
- [75] M.W.Hall, J.M.Anderson, S.P.Amarasinghe, B.R.Murphy, S.W.Liao, E.Bugnion, M.S.Lam, Maximizing Multiprocessor Performance with the SUIF Compiler, IEEE Computer, December 1996.
- [76] M.W.Hall, B.R.Murphy, S.P.Amarasinghe, S.Liao, M.S.Lam. Inter-procedural Parallelization Analysis: A Case Study. Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing (LCPC95), Aug. 1995.
- [77] M.H.Halstead Elements of Software Science, Operating and Programming Systems Series, Vol. 7, New York, NY: Elsevier, 1977.
- [78] L.Hammond, M.Willey, K.Olukotun. Data Speculation Support for a Chip Multiprocessor. In Proceedings of ASPLOS-VIII, October 1998.
- [79] L.S.Hammond, Hydra: A Chip Multiprocessor with Support for Speculative Thread-Level Parallelization. Dissertation, Department of Electrical Engineering, Stanford University, March 2002.
- [80] H.Han, G.Rivera, C.W.Tseng. Software Support for Improving Locality in Scientific Codes. 8th Workshop on Compilers for Parallel Computers (CPC'2000), Jan. 2000.
- [81] H.Honda, M.Iwata, H.Kasahara. Coarse Grain Parallelism Detection Scheme of Fortran programs. Trans. IEICE, J73-D-I(12), Dec. 1990.

- [82] IBM RISC System/6000 Technology, IBM International Technical Support Organization SA23-2619.
- [83] IEEE Computer Society, POSIX System Application Program Interface: Threads Extension [C Language] POSIX 1003.4. Available from the IEEE Standards Department.
- [84] R.A.Iannucci, Toward a dataflow/von Neumann hybrid architecture, Proc. 15th ISCA, May 1988, pp. 131-140.
- [85] J.Kahle. Power4: A Dual-CPU Processor Chip. Microprocessor Forum '99, October 1999.
- [86] S.Karmesin, J.Crotinger, J.Cummings, S.Haney, W.Humphrey, J.Reynders, S.Smith, T.J.Williams. Array Design and Expression Evaluation in POOMA II. In D.Caromel, R.R.Oldehoeft, M.Tholburn, editors, Computing in Object-Oriented Parallel Environments, volume 1505 of Lecture Notes in Computer Science, pages 231-238. Springer-Verlag, 1998.
- [87] H.Kasahara, M.Obata, K.Ishizaka. Automatic Coarse Grain Task Parallel Processing on SMP using OpenMP. Waseda University, LCPC 2000.
- [88] H.Kasahara. A Multi-grain Parallelizing Compilation Scheme on OSCAR. Proc. 4th Workshop on Languages and Compilers for Parallel Computing, Aug. 1991.
- [89] H.Kasahara, M.Okamoto, A.Yoshida, W.Ogata, K.Kimura, G.Matsui, H.Matsuzaki, H.Honda. OSCAR Multi-grain Architecture and Its Evaluation. Proc. International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, Oct. 1997.
- [90] H.Kasahara, H.Honda, M.Iwata, M.Hirota. A Macro-dataflow Compilation Scheme for Hierarchical Multiprocessor Systems. Proc. Int'l. Conf. on Parallel Processing, Aug. 1990.
- [91] H.Kasahara. Parallel Processing Technology. Corona Publishing, Tokyo, Jun. 1991.
- [92] H.Kasahara, H.Honda, S.Narita. Parallel Processing of Near Fine Grain Tasks Using Static Scheduling on OSCAR. Proc. IEEE ACM Supercomputing'90, Nov. 1990.
- [93] T.Knight. An Architecture for Mostly Functional Languages. In Proceedings of the ACM Lisp and Functional Programming Conference, pages 500-519, August 1986.
- [94] Y.Kodama, Y.Koumura, M.Sato, H.Sakane, S.Sakai, Y.Yamaguchi, EMC-Y: Parallel processing element optimizing communication and computation, Proc. 1993 Int'l Conf. Supercomputing, Jul. 1993, pp. 167-174.
- [95] V.Krishnan, J.Torrellas. The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors. In International Conference on Parallel Architectures and Compilation Techniques (PACT), October 1999.
- [96] Kuck & Associates, Inc., Experiences With Visual KAP and KAP/Pro Toolset Under Windows NT, Technical Report, Nov. 1997.
- [97] A.Kumar, The HP PA-8000 RISC CPU, IEEE Micro, 17 (Mar./Apr. 1997), pp. 27-32.
- [98] M.S.Lam. Locality Optimizations for Parallel Machines. Third Joint International Conference on Vector and Parallel Processing, Nov. 1994.
- [99] J.Laudon, D.Lenoski, The SGI Origin: A ccNUMA Highly Scalable Server, Proceedings of the 24th. Annual International Symposium on Computer Architecture, pp. 241-251, Denver, Colorado, June 1997.
- [100] B.Lee, A.R.Hurson, Dataflow architectures and multithreading, IEEE Computer 27 (Aug. 1994), pp. 27-39.
- [101] P.F.Leggett, 1998, CAPTools Communication Library (CAPLib), Technical report, CMS Press, Paper No. 98/IM/37.
- [102] P.Marcuello, A.Gonzalez. Clustered Speculative Multithreaded Processors. In Proc. of the ACM Int. Conf. on Supercomputing, June 1999.
- [103] B.Marsh, M.Scott, T.LeBlanc, E.Markatos, First-Class User-Level Threads, Proceedings of the 13th. ACM Symposium on Operating System Principles (SOSP), October 1991.
- [104] X.Martorell, E.Ayguade, N.Navarro, J.Corbala, M.Gonzalez, J.Labarta, Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors, Proceedings of the 13th ACM International Conference on Supercomputing (ICS'99), Rhodes, Greece, June 1999.
- [105] H.M.Mathis, J.D.McCalpin, M.C.Chiang, F.P.O'Connell, P.Buckland, IBM eServer pSeries 690. Configuring for Performance, IBM Server Group, 2002.
- [106] T.J.McCabe, A Complexity Measure, IEEE Trans. Soft. Eng., Vol. 2, No. 6, pp. 308-320, 1976.
- [107] C.W.McCurdy, R.Stevens, H.Simon, W.Kramer, D.Bailey, W.Johnston, C.Catlett, R.Lusk, T.Morgan, J.Meza, M.Banda, J.Leighton, J.Hules, Creating Science-Driven Computer Architecture: A New Path to Scientific Leadership, Computing Sciences Directorate Ernest Orlando Lawrence Berkeley National Laboratory, Argonne National Laboratory, US department of Energy (DOE), 2003.
- [108] Message Passing Interface Forum, MPI: A Message Passing Interface Standard, The International Journal of Supercomputer Applications and High Performance Computing 8, 1994.

- [109] Message Passing Interface Forum, MPI-2: Extensions to the Message-Passing Interface, University of Tennessee, Knoxville, July 1997.
- [110] J.E.Moreira, C.D.Polychronopoulos. Autoscheduling in a Shared Memory Multiprocessor. CSRD Report No.1337, 1994.
- [111] S.Murer, J.A.Feldman, C.C.Lim, M.M.Seidel, pSather: Layered Extensions to an Object-Oriented Language for Efficient Parallel Computation, International Computer Science Institute, University of California at Berkeley, Technical Report 93-028, December 1993.
- [112] NAS Parallel Benchmarks 2.3. <http://www.nas.nasa.gov/NAS/NPB/>
- [113] R.S.Nikhil, Arvind, Can dataflow subsume von Neumann computing?, Proc. 16th ISCA, May 1989, pp. 262-272.
- [114] M.Ojstersek, V.Zumer, P.Kokol, Data flow computer models, Proc. CompEuro '87, May 1987, pp. 884-885.
- [115] M.Okamoto, K.Aida, M.Miyazawa, H.Honda, H.Kasahara. A Hierarchical Macro-dataflow Computation Scheme of OSCAR Multi-grain Compiler. Trans. IPSJ, 35(4):513-521, Apr. 1994.
- [116] K.Olukotun, B.A.Nayfeh, L.Hammond, K.Wilson, K.Chang. The Case for a Single-Chip Multiprocessor. In Proceedings of ASPLOS-VII, October 1996.
- [117] OpenMP Fortran/C Application Program Interface. Version 2.0 March 2002. <http://www.openmp.org>
- [118] J.Oplinger, D.Heine, M.S.Lam. In Search of Speculative Thread-Level Parallelism. In Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99), October 1999.
- [119] The POWER4 Processor. Introduction and Tuning Guide, IBM Red Books SG24-7041-00.
- [120] PROMIS. <http://www.csr.d.uiuc.edu/promis/>
- [121] D.Padua, M.Wolfe. Advanced Compiler Optimizations for Supercomputers. C.ACM, 29(12):1184-1201, Dec. 1986.
- [122] G.M.Papadopoulos, Implementation of a general-purpose dataflow multiprocessor, Tech. Report TR-432, MIT Laboratory for Computer Science, Cambridge, Ma., August 1988.
- [123] G.M.Papadopoulos, D.E.Culler, Monsoon: An explicit token-store architecture, Proc. 17th ISCA, Jun. 1990, pp. 82-91.
- [124] G.M.Papadopoulos, K.R.Traub, Multithreading: A revisionist view of dataflow architectures, Proc. 18th ISCA, May 1991, pp. 342-351.
- [125] Parafraze2. <http://www.csr.d.uiuc.edu/parafraze2/>
- [126] I.Park, M.J.Voss, R.Eigenmann, Compiling for the New Generation of High Performance SMPs, Technical Report, Nov. 1996.
- [127] P.Petersen, D.Padua. Static and Dynamic Evaluation of Data Dependence Analysis. Proc. Int'l conf. on supercomputing, Jun. 1993.
- [128] O.Pochayevets, A.Bode, H.Eichele, Efficient Implementation of a Hybrid Dataflow Machine on Shared Memory Symmetric Multiprocessors, Proceedings of the 1st international conference ACSN'2003, pp. 86-90, Lviv 2003.
- [129] Polaris. <http://polaris.cs.uiuc.edu/polaris/>
- [130] C.D.Polychronopoulos, D.J.Kuck, Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers, IEEE Transactions on Computers, C-36(12), December 1987.
- [131] M.L.Powell, S.R.Kleiman, S.Barton, D.Shah, D.Stein, M.Weeks, SunOS Multithread Architecture, Proceedings of the USENIX Winter '91 Conference, Dallas, Texas, 1991.
- [132] W.Pugh. The OMEGA Test: A Fast and Practical Integer Programming Algorithm for Dependence Alysis. Proc. Supercomputing'91, 1991.
- [133] L.Rauchwerger, N.M.Amato, D.A.Padua. Run-Time Methods for Parallelizing Partially Parallel Loops. Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain, pages 137-146, Jul. 1995.
- [134] J.V.W.Reynders, P.J.Hinker, J.C.Cummings, S.R.Atlas, S.Banerjee, W.F.Humphrey, S.R.Karmesin, K.Keahey, M.Srikant, M.Tholburn. Pooma. In G.V.Wilson, P.Lu, editors, Parallel Programming Using C++. MIT Press, 1996.
- [135] R.S.Nikhil. Id Language Reference Manual Version 90.1. Technical Report CSG Memo 284-2, Laboratory for Computer Science, MIT, 1991.
- [136] G.Rivera, C.W.Tseng. Locality Optimizations for Multi-Level Caches. Super Computing '99, Nov. 1999.
- [137] L.Roh, W.A.Najjar, Design of a storage hierarchy in multithreaded architectures, Proc.MICRO-28, 1995, pp. 271-278.
- [138] SUN Microsystems Inc., The Ultra Enterprise 10000 Server, Technical White Paper, 1997.

- [139] SUN Microsystems Inc., Pthreads and Solaris Threads: A Comparison of two user level threads APIs, SunSoft, Revision A, May 1994.
- [140] SUN Microsystems Inc., The SUN Enterprise Cluster Architecture, Technical White Paper, October 1997.
- [141] S.Sakai, Synchronization and pipeline design for a multithreaded massively parallel computer, *Advanced Topics in Dataflow Computing and Multithreading* (L.Bic, J.L.Gaudiot, G.R.Gao, eds.), IEEE Computer Society Press, 1995, pp. 55-74.
- [142] S.Sakai, K.Okamoto, H.Matsuoka, H.Hirono, Y.Kodama, M.Sato, Super-threading: Architectural and software mechanisms for optimizing parallel computation, *Proc. 1993 Int'l Conf. Supercomputing*, Jul. 1993, pp. 251-260.
- [143] S.Sakai, Y.Yamaguchi, K.Hiraki, Y.Kodama, T.Yuba, An architecture of a dataflow single chip processor, *Proc. 16th ISCA*, May 1989, pp. 46-53.
- [144] A.V.S.Sastry, L.M.Patnaik, J.Silc, Dataflow architectures for logic programming, *Electrotechnical Review* 55 (1988), pp. 9-19.
- [145] J.A.Sharp, *Data flow computing*, Ellis Horwood Ltd. Publishers, 1985.
- [146] A.Shaw, Arvind, R.P.Johnson, Performance tuning scientific codes for dataflow execution, *Proc. PACT'96*, Oct. 1996, pp. 198-207.
- [147] S.Shende, A.D.Malony, J.Cuny, K.Lindlan, P.Beckman, S.Karmesin. Portable Profiling and Tracing for Parallel Scientific Applications using C++. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 134-145. ACM, 1998.
- [148] S.Shende, A.D.Malony, S.Hackstadt. Dynamic Performance Callstack Sampling: Merging TAU and DAQV. In B.Kaegstroem et al., editors, *Applied Parallel Computing, PARA'98, Lecture Notes in Computer Science*, No. 1541, pages 515-520. Springer-Verlag, 1998.
- [149] J.Silc, B.Robic, The review of some data flow computer architectures, *Informatica* 11 (1987), pp. 61-66.
- [150] Silicon Graphics Computer Systems (SGI), MIPSpro C and C++ Pragmas, Doc. num. 007-3587-001, <http://techpubs.sgi.com>, 1998.
- [151] Silicon Graphics Computer Systems (SGI), IRIX 6.4/6.5 manual pages: mp(3F) & mp(3C), IRIX online manuals, also in <http://techpubs.sgi.com>, 1997-1999.
- [152] Silicon Graphics Computer Systems (SGI), MIPSpro Auto-Parallelizing Option Programmer's Guide, Doc. num. 007-3572-002, <http://techpubs.sgi.com>, 1998.
- [153] Silicon Graphics Computer Systems (SGI), MIPSpro Fortran 77 Programmer's Guide, Doc. num. 007-2361-006, <http://techpubs.sgi.com>, 1998.
- [154] Silicon Graphics Computer Systems (SGI), Origin and Onyx2 Theory of Operations Manual, Doc. num. 007-3439-002, <http://techpubs.sgi.com>, 1997.
- [155] Silicon Graphics Computer Systems (SGI), Origin2000 and Onyx2 Performance Tuning and Optimization Guide, Doc. num. 007-3430-002, <http://techpubs.sgi.com>, 1998.
- [156] Silicon Graphics Computer Systems (SGI), REACT Real-Time Programmer's Guide, Doc. num. 007-2499-006, <http://techpubs.sgi.com>, 1998.
- [157] D.F.Snelling, G.K.Egan, A comparative study of data-flow architectures, Tech. Report UMCS-94-4-3, University of Manchester, Department of Computer Science, 1994.
- [158] G.S.Sohi, S.Breach, T.N.Vijaykumar. Multi-scalar Processors. In *Proceedings of ISCA 22*, pages 414-425, June 1995.
- [159] V.P.Srini, An architectural comparison of dataflow systems, *IEEE Computer* 19 (1986), pp. 68-88.
- [160] J.G.Steffan, T.C.Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallellization. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.
- [161] J.G.Steffan, C.B.Colohan, A.Zhai, T.C.Mowry, A Scalable Approach to Thread-Level Speculation, *Proceedings of the 27th Annual International Symposium of Computer Architecture*, pp. 1-12, Vancouver, BC, June 2000.
- [162] D.Stoutamire, pSather 1.0 Manual, International Computer Science Institute, University of California at Berkeley, Technical Report 95-058, October 1995.
- [163] J.Strohschneider, B.Klauer, K.Waldschmidt, An associative communication network for fine and large grain dataflow, *Proc. Euromicro Workshop on Parallel and Distr. Processing*, 1995, pp. 324-331.
- [164] J.Strohschneider, B.Klauer, S.Zickenheimer, K.Waldschmidt, Adarc: A fine grain dataflow architecture with associative communication network, *Proc. 20th Euromicro Conf.*, Sep. 1994, pp. 445-450.
- [165] S.A.Thoreson, A.N.Long, J.R.Kerns, Performance of three dataflow computers, *Proc. 14th Ann. Comp. Sc. Conf.*, Feb. 1986, pp. 93-99.

- [166] K.R.Traub, G.M.Papadopoulos, M.J.Beckerle, J.E.Hicks, J.Young, Overview of the Monsoon project, Proc. 1991 Int'l Conf. Comput. Design, 1991, pp. 150-155.
- [167] P.C.Treleaven, D.R.Brownbridge, R.P.Hopkins, Data-driven and demand-driven computer architectures, Computing Surveys 14 (1982), pp. 93-143.
- [168] M.Tremblay. MAJC: Microprocessor Architecture for Java Computing. HotChips '99, August 1999.
- [169] J.Y.Tsai, J.Huang, C.Amlo, D.J.Lilja, P.C.Yew. The Super-threaded Processor Architecture. IEEE Transactions on Computers, Special Issue on Multithreaded Architectures, 48(9), September 1999.
- [170] P.Tu, D.Padua. Automatic Array Privatization. Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing, 1993.
- [171] A.Tucker, A.Gupta, Process Control and Scheduling Issues for Multi-programmed Shared-Memory Multiprocessors, Proceedings of the 12th. ACM Symposium on Operating System Principles (SOSP), December 1989.
- [172] A.Tucker, Efficient Scheduling on Multi-programmed Shared-Memory Multiprocessors, Ph.D. Thesis, Stanford University, December 1993.
- [173] D.M.Tullsen, S.J.Eggers, H.M.Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In Proceedings of ISCA 22, pages 392-403, June 1995.
- [174] S.Vajracharya, S.Karmesin, P.Beckman, J.Crottinger, A.Malony, S.Shende, R.Oldehoeft, S.Smith, SMARTS: Exploiting Temporal Locality and Parallelism through Vertical Execution, Los Alamos National Laboratory, Los Alamos, NM, U.S.A.
- [175] I.Watson, J.R.Gurd, A prototype data flow computer with token labeling, Proc. National Comp. Conf., Jun. 1979, pp. 623-628.
- [176] B.Weissman, Active Threads: an Extensible and Portable Light-Weight Thread System, International Computer Science Institute, University of California at Berkeley, Technical Report 97-036, September 1997.
- [177] M.Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley, 1996.
- [178] M.Wolfe. Optimizing Supercompilers for Supercomputers. MIT Press, 1989.
- [179] K.C.Yeager, The MIPS R10000 super-scalar microprocessor IEEE Micro, 16 (Apr. 1996), pp. 28-40.
- [180] A.Yoshida, K.Koshizuka, M.Okamoto, H.Kasahara. A Data-Localization Scheme among Loops for each Layer in Hierarchical Coarse Grain Parallel Processing. Trans. of IPSJ, 40(5), May. 1999.
- [181] A.C.Yuceturk, B.Klauer, S.Zickenheimer, R.Moore, K.Waldschmidt, Mapping of neural networks onto dataflow graphs, Proc. 22nd Euromicro Conf., Sep. 1996, pp. 51-57.
- [182] Y.Zhang, L.Rauchwerger, J.Torrellas. Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors. In Fifth International Symposium on High-Performance Computer Architecture (HPCA), pages 135-141, January 1999.