CSE ND Technical Report

Serial No. TR-01-4

HIGH-LEVEL PROTOTYPING FOR THE HTMT PETAFLOP MACHINE

A Thesis

Submitted to the Graduate School

of the University of Notre Dame

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in Computer Science and Engineering

by

Lilia Vitalyevna Yerosheva, B.S.

Director: Dr. Peter M. Kogge, Director

Department of Computer Science and Engineering

Notre Dame, Indiana

April 2001

Abstract

The Hybrid Technology MultiThreaded (HTMT) project is an attempt to design a machine with radically new hardware technologies that will scale to a petaflop by the 2004 time frame. These technologies range from multi-hundred GHz CPUs built from superconductive RSFQ devices through active optical networks and 3D holographic memories to Processing-In-Memory (PIM) for active memories. The resulting architecture resembles a three level hierarchy of “networks of processing nodes” of different technologies and functionality. All this new technology, however, has a huge and unknown effect on software execution models for applications. This thesis discusses several potential HTMT models and how they can be prototyped and demonstrated using a combination of multithreaded Java and LAN-connected workstations.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Ever greater performance for large computation problems has been the driving force for the development of powerful supercomputers and massively parallel computers with hundreds or thousands of processors. In 1996 the Accelerated Strategic Computing Initiative (ASCI) program [1][2] represented the state of the art in high performance computers with several thousand microprocessors where the total performance peaks in the teraflop range (10^{12} flops/sec). Now, the development of technologies for the construction of massively parallel systems that could achieve 30 teraflops peak performance still is the main objective of the US Government in the High Performance Computing and Communications (HPCC) program [3][4]. This level of performance is required by many applications, such as weather modeling, simulation of physical phenomena (e.g., molecular dynamics, high-energy physics, plasma physics, astrophysics), aerodynamics (e.g., design of new aircraft), simulation of neural networks, simulation of chips, structural analysis, real-time image processing and robotics, artificial intelligence, seismology, animation, real-time processing of large databases, etc.

The major problem with the design of such systems is the latency in accessing the physically distributed memory and in communication between multiple processing nodes. As a result, software development for ASCI is challenging. If we look to the next level of performance, petaflops (10^{15} flops/sec), projections of machines using conventional CMOS technology indicate that even by the 2010 era, over a million 1GHz CPUs will be needed in a complex interconnection scheme with very long and complex latency paths.

1.1 Classical execution and programming models

The latency problem in the system architecture is closely tied to the details of execution model. The *execution model* for a given computer or system is a set of transformation rules that defines how the different subsystems of a computer interact in the process of executing a program. For example, for a von Neumann computer architecture model, such subsystems are memory and CPU, and the execution model is that a CPU sequentially generates a series of memory requests. The execution model establishes the operational relationship between the hardware level mechanisms and the programming model.

Programming models are typically designed with a specific hardware architecture in mind. A programming model is an interface provided to the user by the programming language, compiler, libraries, runtime system (or anything else that the user directly “programs”) which defines facilities that are visible to the program, and specifies to a programmer how to construct a program which manipulates these resources to do some application. The classical examples of programming models are shared memory and data parallel models for shared memory machines, and explicit message-passing and client-server models for distributed computing. The programming model must provide a way for the user to express parallelism (SIMD, MIMD), communication (shared memory, massage-passing), and synchronization (mutual exclusion, barriers, memory consistency) in the applications, and be mapped to an underlying architecture by hiding machine details.

1.2 HTMT - a new execution model

The Hybrid Technology Multi-Threaded architecture (HTMT) project [5] is a solution to the hardware problem in a way that, hopefully, simplifies the software problem by providing an execution model that makes latency management visible to the programer (allowing the control and optimization of the program execution on both hardware and software levels). The project focus lies in an attempt to design a machine with radically new device technologies with superior properties that will scale to a petaflop by 2005 or earlier, and to incorporate an execution model to simplify parallel system programming while expanding generality and applicability. The goal of the HTMT project is to have a working machine 5 to 10 years earlier than such performance levels might be possible with conventional CMOS technology. The computational performance will be dramatically improved through recent advances in Superconducting Rapid Single Flux Quantum logic (RSFQ) which will make 100 GHz clock rates. Processing-In-Memory (PIM), an intelligent RAM with significant amount of logic placed on a high-density memory part, will permit a huge increase in memory bandwidth while reducing local latencies in data accesses dramatically. Memory capacity will be increased through a new memory hierarchy merging advanced semiconductor high density memory (DRAM) and future optical 3-D holographic storage capable of storing 1 Petabytes or more. Interconnection bandwidth will be greatly enhanced by means of optical network (Data Vortex) with 100 Gbps channels. In terms of effects on software models, HTMT introduces multithreading in parallel CPUs, extremely deep memory hierarchies, and active memories to counteract the resulting extreme latencies. Each of these are new technologies by themselves at the current time, with little or no understanding of how they would interact.

The HTMT is a new class of parallel architectures for which no software is implemented at present time, no coherent and complete programming models are available, and no complete description of internal functions that will be needed for real program execution, available.

1.3 Thesis goals

The main directives for pursuing advances in information technology in the President’s program for the 21st century [23] included the detailed design studies and simulation of alternative high-end architectures, the detailed simulation of multiple new device types, and prototyping an implementation of critical elements of future generation high-end computer architectures and others.

These match the overall goal of this thesis work: to create a prototype for the HTMT - a new generation high-end computer scaled to petaflops. We need to describe the nature of the HTMT system and to define its place in between existing architectures, to survey existing programming models for constructing reasonable combinations in the prototype that match different key points of the HTMT architecture, particularly, in layers of the smart memory (PIM), and to explain how different memory hierarchy levels interact with themselves and the extremely fast superconductive processors, called SPELLs. Finally, we need to develop a framework, or a program execution flow scheme, that can serve as early software demonstrations to show new features introduced by HTMT, and to allow gathering of early system statistics.

1.4 Approach

The HTMT design is in the beginning design stage and it is a challenge to collect enough information about this system for both the hardware and the software sides of the project. Thus, a significant portion of the time in the work on this project is understanding and learning about the HTMT system.

The design and development of the HTMT program execution model provides a basis by which the architecture can keep all critical resources in the system usefully busy for petaflop-scale applications, while the latency of the multi-level memory hierarchy is smoothly and effectively tolerated and managed. To prototype this model means to consider the way this execution model works on a not completely defined system architecture, to analyze the dynamics of the HTMT, and to give some statistics on the system that might affect the development of other parts of the system, such as operating system, run-time system, new languages, and new algorithm implementations.

Our approach is to build a working sample program that allows a nearly complete study of the key properties of the HTMT architecture. To do this, we considered different types of existing massive parallel computers and systems, and their features and disadvantages. We studied different execution models for several systems, programming models (such as multithreading, sockets, Linda, etc.), and developed more detailed definitions of the HTMT execution model in terms of the parcel model.

We divided the actual implementation of the prototyping process for the HTMT execution model into small separated, but logically connected, parts, analyzed the dynamics of the HTMT hardware functions with Petri Nets and constructed prototypes using Java programming language constructs. We considered the way of the assembling of the modeled pieces, possibly, but not necessary optimal, and studied an algorithm for matrix multiplication using this model. Several other applications (such as N-body, Fractals, Jacobian factorization) were studied and mapped to the HTMT programming execution model as an extension to this work, and are presented in [72].

The Java object-oriented programming language was chosen as a candidate for creating the HTMT prototype because it supports multithreading which permits emulation of concurrent thread execution on today's symmetric memory processors (SMPs). In addition, Java has support for local area network (LAN) connected parallel applications, and growing suites of libraries for parallel programming. All these language and library features allow us to reflect the system prototype dynamics at a very fine level of details during the execution process simulation.

1.5 Contributions

In this project we analyzed several different massively parallel systems and types of shared memory architectures as to their relevance to the HTMT. Several programming models were studied and described in terms of HTMT. A particular programming prototype for the HTMT execution model was designed, implemented and tested with a matrix multiply algorithm. The results of those tests showed how different parts of the HTMT system can interact. We learned that the software execution models, in particular, directly influence the design of the runtime for the HTMT system, especially hardware support in the PIMs.

Initially, we created a simplified multithreaded model to prototype the system execution model. To express the dynamics of the system, we automated most of the initially hardcoded functions and began to expand this prototype with the parcel model (for communicating between system modules) and some newly developed features for the execution model. We defined the execution model by providing new definitions of the parcels, their types, and parcel functions, identifying the hierarchy of the HTMT software interacting objects and the key runtime functions which will be executed on PIMs. Finally, we used the Petri Net theoretical model to simulate concurrency in HTMT and as a modelling tool to build a software prototype model in Java.

Several papers [72][106][107][108][109] were prepared and published as a part of this effort.

1.6 Related work

There are many execution models that have been proposed and studied for different classes of architectures (shared memory, distributed, and the combinations of both). Examples of such classical execution models include: a centralized shared memory architecture (UMA) with multiple processors that share a single centralized (or virtually centralized) memory (CrayT3D-1[80], CrayT3E[74]); a distributed shared memory architecture with one processor per node [60]; a distributed shared memory architecture among the nodes, possibly with multiple processors at each node; a shared memory on top of hardware for message passing architecture; a distributed shared memory (scalable) with additional level of (non-uniform accesses) memory, shared among a subset of processors, that can be accessed without using the bus-based network (SGI/Cray Origin[81], HP/Convex Exemplar[82]); a cache only memory execution model (DASH[79], Alewife[76]); distributed virtual memory or shared virtual memory models where the software supports the shared memory architecture; and a logical disjointed distributed shared memory model such as multicomputers (clusters of machines, like IBM SP2[77]).

Widely used programming models such as data parallelism (CM-5[22], Split-C[27]), shared memory (or data passing) (SGI/Cray SHMEM[84], University of Oxford BSP[83], Linda[35]), message passing (MPI[73]), multithreading in a shared memory environment (J-machine[70], Alewife[76], Tera[26]), remote procedure calls (Solaris[75]), active messages (J-machine[70], nCube/2[78], CM-5[22], Intel Paragon[28]), and others, provide the basis for design of new classes of execution and programming models on top of new massively parallel computer systems.

1.7 The thesis organization

This thesis is organized as follows. Chapter 2 provides an overview of the HTMT project. Chapter 3 introduces the key HTMT subsystems. In Chapter 4 we explain the HTMT execution model, parcel model, and present the new HTMT programming model. Chapter 5 discusses software models that potentially may fit the HTMT model. Chapter 6 presents the client-server models that describe the early HTMT prototype and outlines the opportunities for concurrent programming that can be expressed in the high-level object-oriented language Java. In Chapter 7 we present our work on designing a HTMT program execution model prototype using Petri Nets. Chapter 8 talks about the HTMT programming prototype implementation and demonstrates how the prototype was tested on the layout of a matrix multiplication problem, and presenting some statistics for the early HTMT prototype. Chapter 9 concludes and shows extension to our work for the future. Finally, the thesis also includes references to the literature.

CHAPTER 2

THE HTMT SYSTEM STRUCTURE

Design and implementation of the HTMT prototype is a part of the HTMT project [6] which attempts to explore and characterize a synthesis of technologies, innovative architectures, and aggressive latency management techniques in a way that could accelerate availability of near petaflops scale computing systems; to develop the architecture, to collect statistics to minimize future design risk, and to determine the feasibility and effectiveness of the HTMT strategy for executing real world computations at revolutionary performance levels. A successful HTMT program will lay the groundwork for prototypes in the 2005 time frame that achieve near petaflop performance with much less than million-way parallelism. It will make the petaflop level performance available sooner, allowing to widen the range of the large time-consuming applications.

The HTMT system is a new architecture with completely new device technologies, component designs, subsystem and system architectures, latency management paradigm and mechanisms, runtime software, compilers, algorithms and applications. In this chapter, we describe the HTMT system architecture, its multi-level memory hierarchy, and provide the main hardware architecture estimates. We conclude with the discussion on the up-to-date state of the work related to HTMT.

2.1 The HTMT system architecture

The HTMT is a distributed Non-Uniform Memory Access (NUMA) architecture. It is an architecture where the physically separate memories can be addressed as one logically shared address space. But, the time needed by a processor to access a given memory location depends upon the distance between the processor and the memory. These latencies are, usually, large and visible to the programmer or operating system. Some memory access latencies in the HTMT can be minimized and, sometimes, avoided because of the use of the new technology, especially, an integration of the memory and the logic on the same die (smart memories), optical switches, superconductive processors, and other devices and system policies.

The HTMT architecture is shown in Figure 2.1. It consists of 4096 multithreaded CPUs, called SPELLs, constructed from low-power RSFQ (Rapid Single Flux Quantum) superconductive devices with special 1M cache like local memories, called Cryo RAM (CRAM), cooled in a cryostat (liquid helium) to 4° K, and capable of clock rates in excess of 100 GHz. These memories are connected to the next level of hierarchy through a RSFQ-implemented self-routing, multistage packet switching cryo-network (CNet) that is able to provide a cross-sectional bandwidth of ~0.7 word per processing node per network clock cycle (30 ps). To interconnect components in the cryostat with external components, a very high bandwidth (100-250 Gbits/fiber) fine-grained packet switching optical network (Data Vortex) with 10 Gbps channels is used to reduce the time-of-flight latency (10-100 ns) between thousands of communication ports.

"Smart" SRAM and DRAM memories are based on PIM technologies on either side of the optical network. The SRAM (liquid nitrogen cooled device with 3 ns of access time) represents an intermediate level of memory hierarchy. It is located in the "warm" part of the cryostat, and the external DRAM is connected to it by the optical network. In PIMs, the memory and the logic are placed together on the same chip to provide the high memory bandwidth minimizing the latencies in hardware for the memory accesses. It reduces the overhead of memory oriented operations, manages the data structures and context coordination and percolation allowing high system performance. One example of processing logic that will be implemented in PIMs is Move engines which allow automatic data transfer between different memories over the networks. Other PIM functions will be discussed later.

The next level of the memory hierarchy consists of *3D* holographic memories (HRAM) for extremely dense and fast on-line backing storage with sustained bandwidth of ~100-1,000 Gbps per storage module. They are arrayed in parallel, and are capable of storing Petabytes or more. Large arrays of disks and tape silos (not shown in the picture) are used as an external storage.

2.2 Memory hierarchy

Each level of the HTMT system hierarchy has a separate address space. During the execution, data in the HTMT become distributed in such a way that the frequently used data are allocated in very fast memory units (SPELLs), and the rarely used data are moved to an auxiliary memory. This data allocation process is called data migration. The HTMT physical memory hierarchy consists of five levels [48]: HRAM, DRAM,

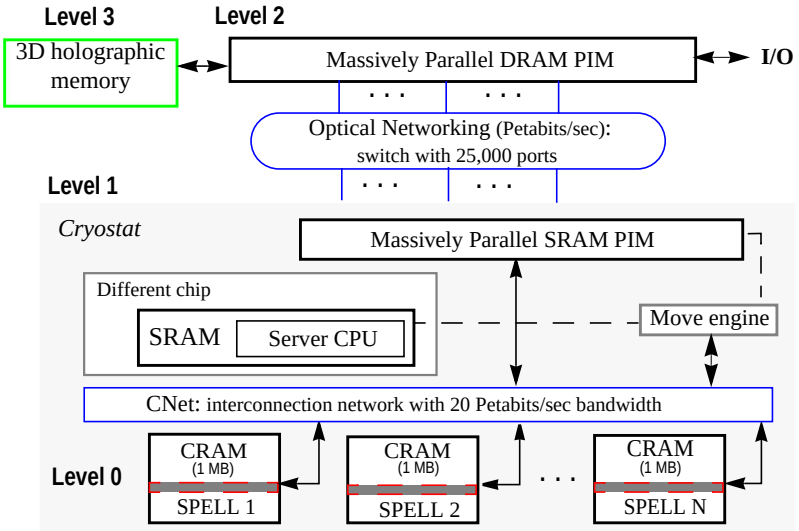


Figure 2.1 - HTMT architecture

SRAM, CRAM, and disk or the tape. The CRAM level is physically partitioned and these partitions are attached to individual processors. All these processors logically share the same address space.

Table 2.1 summarizes addressability of the HTMT memory hierarchy levels of the by different engines. The different numbers in the table stand for different types of memoryoperations. Minus in the table means that there is no memory operations are allowed between these two particular devices. “1” means that processor can directly access memory via “Load and Store” operations. Plus sign shows that this access to the memory requires short packets in data vortex, plus multithreading to overcome latency. “2” indicates that

Table 2.1 - The HTMT memory hierarchy

	CRAM	**SRAM**	**DRAM**	**HRAM**
SPELL	1, 2	1, 2, 3	-	-
SPIM	1, 2, 3	1, 2, 4	1+, 2, 3	-
DPIM	-	1+, 2, 3	1, 2, 4	3

“Load and Store” operations include “Fetch And op” operations such as addition, subtraction, semaphore operations for the operating system, and “parallel prefix” operation. During these operations (can be done, especially, between CRAMs and SRAMs) the “opcode” of data is sent from one memory space to another and results are received back using technique called remote method invocation (RMI) that is described in next chapter. “3” tells us that the processor can initiate a block data transfer from the memory in the column to its own type. “4” shows remote memory interactions by associated PIMs.

Since the HTMT system has multiple memory hierarchy with associated processors attached to different memory levels, it allows to prevent data stalling on memory requests and minimizes the communication overhead at the same processing level, but, at the same time, it produces new kinds of latency in the system: to distribute and to back-up data during the execution. As an example of the kind of latencies that show up in HTMT, if the SPELLs run at 100 GHz, the latency to the CRAMs is on the order of 10 SPELL cycles, to SRAM is 100s of cycles, 10,000 to 100,000 cycles to DRAM, and millions of cycles to 3D memories. Note that having 100 GHz CPUs (vs. 1GHz for CMOS) means that much less parallelism is needed out of applications. This is good, but at the same time memories do not speed up 100 times, leaving more levels of memory hierarchy and huge access latency. The PIM enabled memory hierarchy counteracts this by active pushing data down the hierarchy before SPELLs request it. The HTMT software model and prototype design must mimic this motion.

2.3 Hardware estimates

This section provides the hardware estimates for the HTMT architecture, including specifications of the main HTMT components, latencies in the system hierarchy, and the HTMT execution model prototype parameters.

We characterize the HTMT architecture components and summarize the statistics for our further assumptions and analysis. The overall target specification for HTMT is shown in Table 2.2 [44] and includes the data for the system components and the communication bandwidth in the system.

The approximate access latency in the HTMT from SPELL point of view are shown in Table 2.3, where two-way latency includes access latency from SPELL to memory and back. At this level of design, the CNet latency assumed to be considered as 1.5 ns, and the Vortex latency as 107 ns.

In our work we made several assumptions on the HTMT characteristics that allowed us to minimize complexity and to increase visibility of results during design, modelling and analysis of the HTMT execution model prototype (Table 2.4). The Prototype entries indicate the part count and the memory capacity for

Table 2.2 - HTMT system characteristics

Sub-System Element	Parts Count	Size per chip/ SPELL cluster	Total Size	I/O Bandwidth
SPELLs	4096	-	1 PFLOP/s	1.2-12 PB/s
CRAM	16K chips	4x256KB	4GB	8 PB/s
SRAM	16K chips	4x64MB	1 TB	4 PB/s
Optical Net	372K nodes	-	6250 ports	640 TB/s
DRAM	32K	8x512MB	16 TB	320 TB/s
HRAM	128K	32x8GB	1 PB	320 TB/s
Disk	100K-1M	-	10PB-100PB	1 TB/s - 10 TB/s
Tape:-robots - transports	20-200 250-2500	- -	100PB - 1EB	24 GB/s - 240 GB/s

Table 2.3 - Latency in HTMT from the SPELL view,'+'means even larger latencies

Memory modules	Capacity (Words) 1 W = 8 B	Capacity (W/Flop)	Read bandwidth (W/Flop)	Write bandwidth (W/Flop)	Two-way latency (cycles)
Local CRAM	128 K	0.000005	1	1	70
Local SRAM	32 M	0.000125	0.25	0.25	240+
Remote CRAM	512 M	0.002	0.08	0.08	270
Remote SRAM	128 G	0.5	0.08	0/08	420+
Single DRAM cluster	512 M	0.002	0.04	0.04	16K
HRAM on 1 cluster	32 G	0.125	0.04	0.04	67K
4 DRAM clusters	2 G	0.008	0.16	0.04	16K
HRAM on 4 clusters	128 G	0.5	0.16	0.04	67K

the HTMT execution model prototype. The Simplified Model entries provide the characteristics for one simulated (CRAM-SRAM-DRAM levels) cluster which becomes a building block for a Petri Net model design and analysis, and for a future extensions of the HTMT execution model prototype.

2.4 Related work

Development of the massively parallel HTMT system has opened new questions in computation and organization. Moving memory closer to processors, creating a memory hierarchy, and multithreading give additional levels of complexity. Many systems have similar features to the HTMT architecture or

Table 2.4 - The HTMT model organization

Sub-system element	Prototype count/capacity	Simplified Model count/capacity
SPELLs	4K	1
CRAM/SPELL	2x1MB	1MB
SPIMs	16K	2
SRAM/PIM	4x4MB	2x4MB
DPIMs	32K	2
DRAM/PIM	8x32MB	4x32MB

execution model. The Tera machine [87] has similar underlying architecture and supports multithreading, but does not put memory and processor on the same chip. The J-machine [70] has active messages and multithreading in a distributed environment but does not consider memory hierarchy. The Beowulf [85][86] groups different types of processing nodes into clusters to ease load balancing, has two networks (interconnection and external) that simplify network loading and support multithreading but does not have a memory hierarchy and PIMs. Several special computer systems were designed to suit particular problems. GRAPE6 [88] was specifically designed to execute particle models where the processes can freely migrate over the system. Researchers study the mapping of hierarchical particle models onto special hybrid computer architectures [89].

2.5 Conclusions

In order to achieve and sustain the HTMT performance at petaflops level, we need to have technological advances in hardware to build very fast processors, active memories that will actively react on the execution process, high-bandwidth low-latency optical interconnection networks, new execution models, new software (such as operating system, compilers, runtime systems, libraries), and new algorithms for the multithreaded multi-level system and applications.

CHAPTER 3

THE KEY HTMT SUBSYSTEMS

As stated earlier, the HTMT system has a hierarchical network organization with multiple levels of memory and three types of processors: SRAM PIMs, DRAM PIMs, and SPELLs with attached CRAM memory, where PIM is the Processor-In-Memory element, and SPELL is the RSFQ-based superconductor processing element. This section discusses these key HTMT subsystems. This is necessary for our thesis because of the many unique characteristics we had to model.

First, we introduce the active memories of the HTMT system (PIMs), and later, the holographic memory (HRAM) as the data storage for the HTMT. Next, we describe the main computing engines in the HTMT (SPELLs). Then, we explain the state of research on the communication network (CNet), and discuss the optical interconnection (Data Vortex) organization and functionality. Finally, we conclude our study of the HTMT subsystems.

3.1 PIM subsystems

The Processing-In-Memory (PIMs) in the HTMT system are called active memories, which means that on each access to the memory processing can occur “at the memory”. PIMs expose the high memory bandwidth, reduce the overhead for memory oriented operations, manage the data structure manipulations, perform context coordination and percolation, and help achieve very high performance by pre staging data to minimize the latencies in hardware for memory accesses. We distinguish between SRAM PIMs and DRAM PIMs by their functionality, execution model, and the memories to what they are attached: SRAM and DRAM.

3.1.1 SRAM

SRAMs (static random access memory) are simply integrated circuits which are memory arrays with (usually) a single access port that can provide either a read or a write. SRAMs have a fixed access time (possibly, as short as a nanosecond) to any datum, though the read and write characteristics differ. The value stored in a SRAM cell is kept on a pair of inverting gates, and as long as power is applied the value can be kept indefinitely.

3.1.2 DRAM

A modern DRAM is a high-density dynamic memory in which the value in a cell is stored as a charge in a capacitor that needs to be refreshed (periodically read the value and write it back). A single transistor is used to access this stored charge, either to read value or to overwrite the charge stored there.

Because DRAMs use only a single transistor per bit of storage (SRAM uses 4-6), they are much denser and cheaper per bit. It makes DRAM the choice for dense main memory.

The DRAM access times (with the best chips at 50-60 ns - 100 ns and more at the chip level) are longer by factor of 5-10 than SRAM capabilities. It makes SRAM the choice for caches with faster memory access.

3.1.3 PIM technology

PIM technology occurs when the basic memory address inputs and outputs are capable of interfacing to logic circuitry on the same chip. We call a memory array and such adjoining processing logic a node. Multiple such nodes can be placed together on a single chip, with more specialized inter node and off-chip interfaces added out of yet more logic. One such chip architecture, called Shamrock, was designed and analyzed [72], with fabrication planned in the near future.

3.1.4 PIM definition

PIM, also called Intelligent RAM or IRAM, is an increasingly viable VLSI technology that combines on a single CMOS memory chip both dense logic and dense memory [5][7]. This simple trick has a profound impact on computer architecture: if the logic is used to construct CPU-like devices, these CPUs are much closer electrically to the memory arrays containing instructions and data. Combining the processor and memory on a single chip can yield power and timing benefits as off-chip buffers are eliminated and latency is reduced.

Furthermore, the number of bits available from each access can be literally orders of magnitude greater than what one can transfer in a single cycle from today's conventional memory chip to today's conventional (and separate) CPU chip or cache system. An access to the memory array should be possible with an extremely wide memory processor bus interface.One of the first PIM parts with a very significant amount of memory and logic was the EXECUBE chip, fabricated in 1993. It placed an 8 node parallel processor on a 4 Mbit DRAM chip [60]. The M32R/D is the first commercial PIM chip that based on this concept. It was built in 1996 with 2 MB of memory and a single 32 bit CPU. The DIVA (Data IntensiVe Architecture) [71] project is studying the unique issues about address space mapping and the dynamic address translation capabilities for DRAM PIMs which can execute in memory operations on "irregular data structures" (differentiating, tree-searching, database), and have a compiler support capable of serving to a conventional microprocessor.

It is possible to assemble a PIM chip into a variety of configurations, from a single-chip system, through a multi-chip memory single CPU computing mode, to a 3D stack of PIM chips to make up a node in a highly scalable system. The current estimates show that with an advanced technology today such a chip could contain over a 4MB of SRAM, operate at clock rates in excess of 100 MHz, at power levels below 1 watts a chip, with total operation counts approaching a Bop (billion operations per second). An equivalent DRAM PIM chip would hold 32MB of DRAM or more.

3.1.5 PIM structure

In the HTMT design each PIM chip has one or more multithreaded processing nodes, which contain both memory and processing logic; one or more network interface nodes to communicate with the

rest of the HTMT system, especially the non-PIM parts; and one or more PIM-to-PIM interfaces (PPI), which allow multiple PIMs in close proximity to communicate with each other at high speed.

Parcels and parcel interface. A parcel is the unit of initiation of PIM activity in HTMT. It is a logically complete grouping of information that arrives over an external interface (CNet, Data Vortex, PPI links) and fits within a single interface transmission packet. Existing mechanisms similar to parcels are known as active messages and will be discussed later.

The network interface is the logic on a PIM chip which can accept command parcels into the PIM chip from other HTMT subsystems. It also acts as a port whereby data responses can be returned in response to a parcel.

3.1.6 PIM functions

There are five main functions that will be performed by the PIMs. First, is the “context percolation management” function, where data and code are assembled in advance and placed in memory very close to the SPELLs. Second, “gather/scatter” and “pointer chasing” in the memory structure are operations that are necessary to minimize the waiting time for the SPELLs. Third, “data structure allocation and initialization” and “vector/array” operations [36] are employed whenever the ratio of operations to data references is so low that it is faster to do the operations “in memory” than to ship the data all the way down to the SPELLs and back. Those operations in the memory require new cache (memory) consistency models [8]. Finally, the compression and decompression of data sets function in PIM will save time and space in accessing the holographic cubes and disk farm.

3.1.7 DRAM PIM

The DRAM PIM initializes the data structures, exposes fine grain parallelism intrinsic to vector and irregular data structures, e.g. pointer chasing through linked data structures, block moves, synchronization, data balancing [36]. It can stride through regular data structures and transfer data to PIMs and SPELLs, and is capable of performing data parallel basic operations at the row buffer, “join-like” operations, and manages shared resources.

3.1.8 SRAM PIM

The SRAM PIM is responsible for the execution of the following functions: to initiate gather/ scatter operations to and from DRAM, to recognize when sufficient operands arrive in SRAM context block, to enqueue and dequeue SRAM block addresses, to initiate DMA transfers to and from CRAM context block when the space is available, to signal SPELL for the task initiation, and to make some prefix operations (like float-point summation).

3.2 HRAM: optical holographic storage

HRAM is an integrated holographic memory [57] that allows memory intensive but cost critical applications, very high transfer rates and fast access to the memory, and has characteristics intermediate to DRAM and secondary systems (magnetic disks). In particular, the HRAM latency is smaller that in hard

disks by approximately one order of magnitude [48], while its bandwidth (sustained data transfer rate) is ~100 Gbps per storage module.

The holographic memories offer high transfer rates because of their inherent parallelism: hundreds of thousands of bits are read out in parallel with the access of a single two-dimensional data page. The HRAM uses a time-domain holography method [48], using spectral hole burning, for primary storage. Estimates for the optical holographic memory include TB/cm^2 surface storage densities, peak single channel data transfer rates on the order of hundreds of gigabytes/sec, and microsecond latency times (using non-mechanical access for different spatial locations). Each holographic storage module each HRAM modules can contain ~10GB of storage [58], and it is controlled by a PIM chip which serves as its temporary buffer.

3.3 RSFQ subsystem and SPELLs

Several features of the superconductor integrated circuits make them unique for processing digital information at extremely high speed and with very low power, such as (a) microstrip transmission lines that capable of transferring picosecond waveforms over any interchip distance with speed close to the speed of light, with low attenuation, dispersion, and dense layout; (b) Josephson junctions which can serve as ultra fast (picosecond) switches, have very low power consumption (because of low voltage), produce low heat, packet very closely (hence small delays), can work close to the speed of light; and (c) the "Niobium-trilayer" technology of fabrication with light-speed signal propagation that will allow very fast data exchange in the HTMT system.

The Rapid Single Flux Quantum (RSFQ) technology with minimum Josephson junction size of 0.8 micron, 20 kA/cm^2 of critical current density has 100K gates per chip that consumes 0.05 watts per processor and 100K watts per Petaflops. Current estimates for the RSFQ subsystem with 4K SPELLs and a 4-Gbyte CRAM indicate that it should be sufficient to achieve performance close to 0.5 petaflops for many computationally intensive program kernels.

The SPELL is a RSFQ logic based system that uses a kind of magnetic flux to store and manipulate information in an almost lossless manner, and operates with a 100 GHz clock rate (33 GHz inter-chip speed rate), pipelined cryo-memory (CRAM) with 30 ps cycle and inter-processor network (CNet) with a bandwidth of 30 Gbps per channel.

3.3.1 SPELL architecture

The SPELL design differs from almost all the standard microprocessor design techniques because of the light speed of its circuits (the high clock rate is so fast that light speed of signals is not fast enough to travel far) and the very large latencies in the various speed HTMT system levels visible to a SPELL processor (~30 processor cycles to access local CRAM, ~1,000 cycles to SRAM).

The multithreaded approach to hide the latency enables independent instruction streams (threads) to be interleaved at each level with minimal overhead to eliminate performance decrease. This technique also permits a global memory address space and simplifies parallel programming for end-users. Multithreading was introduced in 1960 (CDC 6600 peripheral processors [62]). Later, several high-performance multithreaded computers have been built, including HEP [64], MARS-M [65], and Tera [66], and the concept has been studied in different research projects [67][68].

The instruction set architecture (64-bit RISC COOL-1 ISA[63]) supports *two-level simultaneous multithreading* and *pseudo-vector compilation* (using quad instructions to perform operations on short 4-

word vectors). The first level of multithreading, called threads, is the coarse-grain parallelism that represents parallel function/procedure/process invocations. The second level, called strands, is medium-grain parallelism that are the program entities inside threads. COOL-1 ISA, for pseudo-vector operations, does not rely on vector registers. Instead, it uses adjacent data registers to hold fetched input/output operands of the quad instruction.

3.3.2 SPELL organization

The top structure of the SPELL is presented in the Figure 3.1. To provide enough parallel work for all functional units, each SPELL has 16 multi-stream units (MSU) [69] that are combined into four clusters. Each cluster has a single 8 KB multi-port instruction cache (I-Cache) that is shared by 4 MSUs within the cluster. Such shared instruction caches are beneficial when several threads execute the same code, because

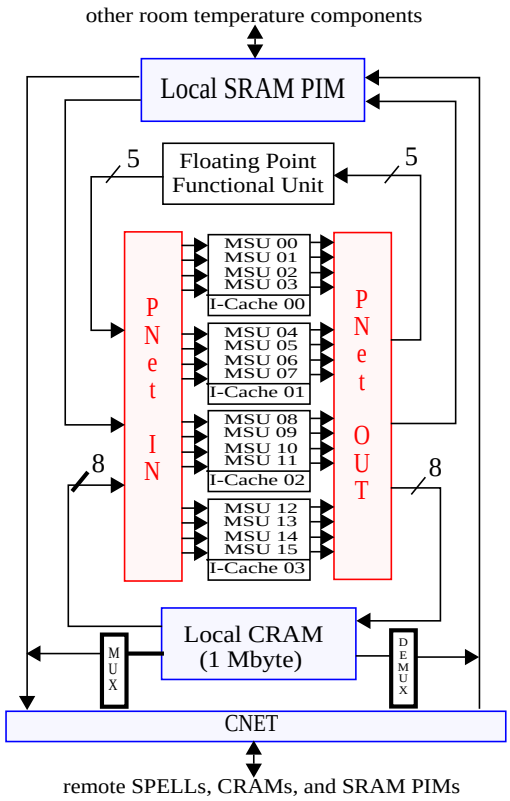


Figure 3.1 - The SPELL structure

only one copy of the code needs to be fetched from CRAM to I-Cache.

Each MSU executes all integer, control, floating-point compare operations within one thread, using a 64-bit integer functional unit and eight 32-bit branch units sharing a 32-bit address adder, all operating within a 15ps cycle. Each SPELL has 5 arithmetic floating-point functional units (FPUs), each operating at 15ps cycle time giving the output performance of 0.3 Tflops per SPELL. The FPUs and CRAM are shared by all instruction streams from the 16 MSUs within one SPELL.

The self-routing intra-processor switching network (PNet) connects MSUs, FPUs and CRAM together via 85-bit-wide communication links through which these units can send packages to each other. The PNet has 32 full-duplex ports: 16 to MSUs, 5 to FPUs, and 11 to the processor-memory interface (PMI),

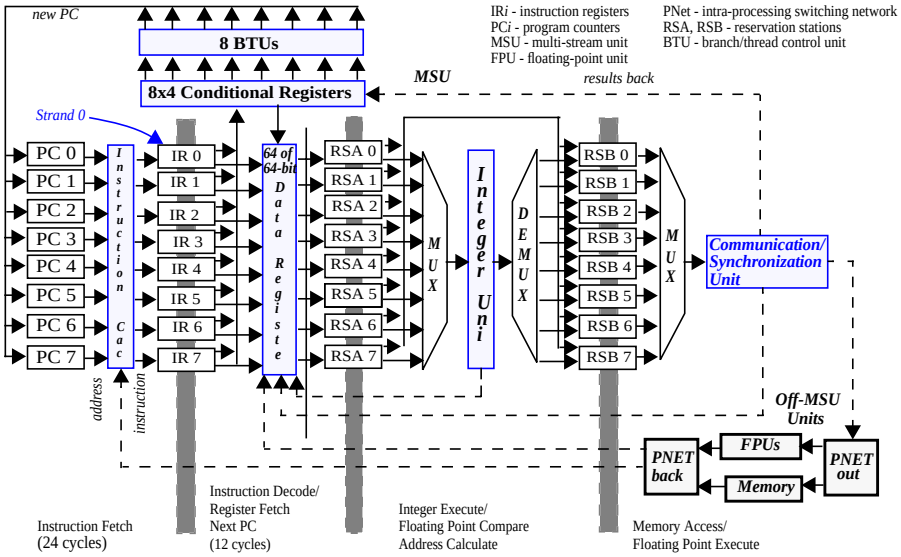


Figure 3.2 - Instruction execution pipeline

each operating at 30 Gbps (limited by hardware pins in the communication node and by the speed of message address decoding and routing). The PMI works at the same speed, providing peak bandwidth of 8 words in each direction to local CRAM, 2 words to local SRAM, via the room-temperature interface (RTI), and 1 word to remote SRAM and computing modules, via CNet, per 32 ps cycle.

3.3.3 SPELL pipelines

The structure of the SPELL pipeline consists of the MSU data path, PNet, CRAM and FPUs (Figure 3.2). In comparison to traditional pipelines, where the stages of the pipeline separated by latches and the signal delay (latency) L per stage equals the clock period T, the SPELL is ultrapipelined. This means that the processor's pipeline consists of five macrostages, each similar in overall function to a single RISC stage.

Each macrostage is composed of several microstages, each usually including one latching gate, non-latching components, and transmission lines. In order to achieve SPELL peak performance the number of independent instructions needed to fill one pipeline macrostage is ~20-40. The load/store and floating-point pipelines require even more instructions.

To provide the processors with enough ready to execute instructions is a challenging task. Such levels of parallelism, when enough data (and control) independent instructions can be made available during the computation at such high speed, can be found only in scientific applications with regularly-structured data objects. The solutions for 100-GHz-scale processors can not be the same as those for *superscalar* (with very complex instruction issue logic in the hardware) or *VLIW* (a very large instruction word with compiler-controlled synchronous computation) *processors*, such as loop unrolling, software pipelining, etc.

The SPELL hardware design provides a way to eliminate the control logic for data dependencies between strands (low-level instruction streams) running simultaneously within each parallel thread. Each integer unit pipeline is shared among multiple strands. The floating-point units with longer pipelines (and requiring more instructions) are shared by all strands of all threads running in parallel within one processor. Each CRAM component will be accessible by all strands in the RSFQ subsystem, where the main data traffic is expected between SPELL and local CRAM.

In the SPELL, there are up to 30 microstages in each macrostage, and up to 8 instruction streams can run simultaneously within each MSU datapath. Except for the Memory Access/Floating-Point Execute macrostage, only instructions from different instruction streams (strands) can be simultaneously processed within each macrostage [62]. Each MSU includes a unified set of 64 data registers. There are also 32 4-bit condition registers used for control transfer, and miscellaneous registers for thread control, synchronization, and exceptions handling.

Instruction streams within each MSU can share the 64 data registers, the instruction cache, and the integer functional unit. Using the program counter, a MSU strand fetches the instruction from the instruction cache and writes it into the instruction register with strand control logic control. Then the instruction is decoded and the operands are read from the register file. Opcode and input operands are placed into reservation station before issuing them to integer/floating-point units and memory.

A branch/thread control unit (BTU) always starts the execution of any thread in an MSU, using the initial instruction address loaded from CRAM. The address is placed into CRAM as part of a thread control block prepared by the SRAM PIM.

Creation and termination of other strands is carried out using special “create/terminate” instructions and requires neither involvement of the runtime system nor allocation of the CRAM. There are no hardware limits (besides the number of the available registers) on the number of outstanding memory references from each strand. The only reason for the individual strand to be suspended by the hardware is the detection of a data/control hazard in the SPELL pipeline.

3.4 CRAM

The CRAM data storage is based on a multi-bank organization, with memory cells organized in 128x64-bit matrices, with one 64-bit word occupying an entire row of this “bank”. Estimates show that the memory matrix clock cycle may be close to 30 ps [62]. Each I/O port operating at 30 Gbps per channel serves a cluster of 128 memory banks. The whole cluster operates in micropipeline mode. Together with PNet and processor-memory interface delays, the local CRAM latency as seen by instruction streams is about 400 ps.

Each CRAM chip has 2 memory clusters. Each SPELL is served by 4 local CRAM chips, i.e. 8 clusters (256 KB per chip, i.e., 1 MB per SPELL). The total volume of CRAM in the 4096 processor system is 4 GB. The communications between each SPELL/CRAM module and “local” SRAM are provided through 2000 wires (including grounds), 8 Gbps per wire.

3.5 CNet: interprocessor network

A self-routing, multistage packet switching cryo-network (CNet) is a high-bandwidth low-latency RSFQ switching network that serves occasional communications between SPELL modules and “remote” blocks of SRAM. The CNet connects 4,096 local memory buffers of SPELLs with remote ones and with 12,288 interfaces to SRAM blocks. It is able to provide a cross-sectional bandwidth of ~0.7 word per processing node per network clock cycle (30 ps).

Previous studies of RSFQ-based switching networks [38] have indicated that the unparalleled speed of these logic devices imposes substantial restrictions on the architecture of networks which can employ this speed effectively [41]. Based on these limitations, several network architectures have been proposed and compared in terms of real estate and timing characteristics [42]: a family of blocking and non-blocking hyper-toroidal networks (pruned meshes [40]), and blocking delta networks (SW banyans [39]). It is too

early to say which is the ideal candidate in the HTMT computation process. However, it was estimated [37] that for a cost of 24,576 switching nodes, two networks can deliver about 1 PByte for second in random access. A prediction of local memory traffic is that about 7 PB/s is sufficient for petaflop computing.

3.6 Data Vortex: optical network

The Data Vortex, a fine grained packet switched optical network [45] is designed to form the communications infrastructure between the SRAM and DRAM levels in the HTMT [59]. The IDA-CCS network [44] was chosen as the basis architecture.

The Data Vortex provides short time-of-flight latency (100 ns) between approximately 20K of communication ports. It employs multiple node levels with no latching and a minimum logic at each to provide the routing. The *Multiple Level Minimum Logic (MLML) network architecture* has the property that all routing decisions for the self-routing data packets are based on a single simple logic operation at each node.

3.6.1 The topology

The topology of the Vortex connecting *N* input buffers and *N* output buffers can be viewed as a set of interconnected concentric cylinders (Figure 3.3). These I/O ports are located on the inner-most cylinder

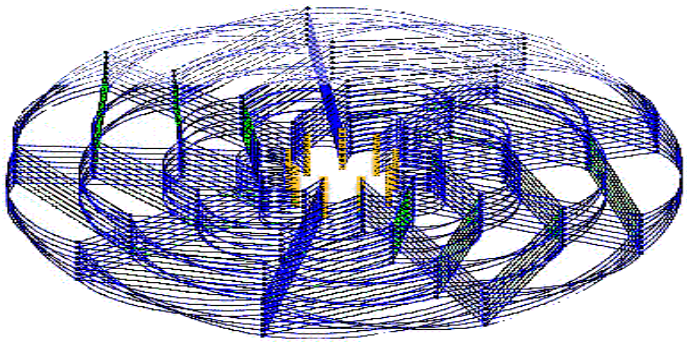


Figure 3.3 - Data Vortex architecture

and are connected to the processors. Each network node (2x2 crossbar switch for data packets handling) has two input and two output ports. The input buffers are connected to nodes on the outermost cylinder, and output buffers are connected to nodes on the inner cylinder [48]. The flow of data on the rings is unidirectional. As the data packets descent from the outer to the inner cylinders, their packet header bits are fixed so that by the time they arrive at the innermost cylinder the complete target buffer address is determined.

3.6.2 Traffic management

Traffic management is accomplished by backward propagation of control bits between the cylinder levels. The control bits arrive at the sending node before the data packet is routed, so that at any one time

instant only one of the two input data ports can be active. Multiple paths between source and destination are possible. If two packets enter the same path, only one is allowed to enter the node in a given clock cycle. The packet deflected by the control stays on its cylinder level, and can be allowed to progress at the next two clock cycles after bouncing through only two nodes (taking another path). The topology is designed so that a data packet propagates in a perfect tour around the cylinder nodes and cannot be blocked twice by the same packet. Thus messages are never lost. There are no electro-optical conversions (hence, no active parts) in the packet passing process.

The interface between the Data Vortex and the PIMs is split into two separate parts: the PIM module and the Input Interface Module [46]. The PIM module is used to direct the admission of packets into the Data Vortex, and to buffer the packets that have been sent, until it is determined that they definitely made it into the Data Vortex. The control signals are used in the control flow scheme. When a packet is not blocked by another packet, it is sent to the Input interface for the electro-optic conversion by the modulators [46].

In the current 2007 HTMT design, the Data Vortex sustained throughput (under fully loaded conditions) is ~5Pbit/sec. Each input (and output) port to the network can deliver packet payloads at 640 Gbit/sec. To achieve this throughput and port bandwidth, and provide a reasonable interface to the PIM DRAM and SRAM the optical technology selected include 64 superimposed wavelength channels each operating at 10 Gbit/sec.

Table 3.1 includes the Data Vortex size calculations that were evaluated for the year 2007. The cross-sectional bandwidth determines the network size from the number of the I/O ports required to carry the sustained (fully loaded) bandwidth. There is an assumption that the Data Vortex to PIM interface can efficiently deliver the data rates in a sustained manner. It was shown that to achieve a high input rate into the

Table 3.1 - Data Vortex network size

Parameter	Units	2001	2004	2007
Overall machine performance goal	**Tflops**	**10**	**100**	**1,000**
Maximum cross-sectional sustained bandwidth	**Tbit/sec**	**40**	**400**	**4,000**
Per-port data rate	**Gbit/sec**	**160**	**320**	**640**
Number of input ports		**250**	**1,250**	**6,250**
Angle nodes number (A)		**5**	**5**	**7 (later 9)**
Network node height (H)		**256**	**1,024**	**4,096**
Number of nodes per cylinder		**1,280**	**5,120**	**28,672**
Number of cylinders (NumC)		**9**	**11**	**13**
Total node number ((A)*(N)*NumC)		**11,520**	**56,320**	**372,736**

network, the number of the input ports should be ~1/5 of the nodes on the outermost cylinder [46]. The angle number (A) means the number of nodes per ring, where the height (N) corresponds to the number of rings in the innermost cylinder.

3.7 Conclusions

In this chapter we introduced the key HTMT subsystems and gave the overview of the future work on their design and development. We showed that the HTMT shared memory NUMA architecture [44] is a very complex system that employs superconducting processors SPELLs and CRAM data buffers with variable access latency (*1 access/cycle*)), cryo-SRAM semiconductor buffers, or SRAM cache memory, semiconductor DRAM main memory (with *40 ns* access time), and HRAM optical holographic storage. It is integrated by the Data Vortex optical interconnects and the CNet packet switched network. The Processors in Memory (PIMs), will increase the memory bandwidth, make memory closer to the SPELLs, hide latency (CRAM: *10s*, SRAM: *100s*, DRAM: *10,000s cycles*), and reduce a power per operation, and provide smart memory support for the multi-level multi-threaded HTMT system.

CHAPTER 4

HTMT EXECUTION MODEL

The new technologies in the HTMT are so different from existing technologies that they require radical changes in system design. The major challenge of the HTMT design is the development of the multi-level multithreaded program execution model to manage the latency in a deep memory hierarchy extending from the superconductor processors to the holographic memory. A first attempt to define the HTMT execution model was made at the University of Delaware [9]. The goal of this thesis is to fill out this design with a more complete set of details.

This chapter discusses the HTMT execution model, gives the main definitions and explains terminology. We describe the multi-level threading model, the HTMT percolation process, and context management. Finally, we introduce parcels as a mechanism to support percolation.

4.1 Terminology

This section explains the main definitions of the HTMT execution model, including percolation, contexts, frames, threads and strands, which allows us to describe the detailed functional flow for the system execution process.

4.1.1 Percolation

The HTMT system will use sophisticated memory management hardware to move data throughout the memory hierarchy and satisfy the latency and bandwidth needs of the SPELLs, and incorporate this into the HTMT threading model. Instead of moving individual elements of data within the memory hierarchy, the HTMT technique will manage complete *contexts*. A context is all the pieces of data needed for a SPELL program to run. In the HTMT multithreaded scheme the contexts propagate to the executing processors automatically through the memory hierarchy so that SPELLs always have access to executable contexts with ready data in buffers in the CRAM without any unexpected latencies. This technique is called *percolation*.

4.1.2 Contexts

An HTMT program is written as a set of threaded procedures. An instantiation of a threaded procedure, called a *thread*, is associated with a separate *context* (a set of *frames* for a code and data for a given set of instructions) of function parameters and local variables. A context packages into a block of memory the data, the program code, and control state needed to run a program. When filled, the context creates a *continuation*. A *closure*, then, indicates mapping a continuation for a given instruction set to the set of machine registers. According to the percolation concept, after starting a program, PIMs perform pre-processing of the program to find ready to execute *threads* and allocate contexts in the CRAM. After a

SPELL finishes the execution of a thread, a SPIM fetches the results from the CRAM into SRAM, and transfers them to DRAM or HRAM if necessary. All the activities, such as finding ready to execute threads, loading input data into the CRAM, execution of threads in SPELLs, and fetching the results can be performed concurrently by different SPIM nodes.

4.1.3 Frames

Because the thread contexts are kept separate, threads can run in parallel, can be created and destroyed at runtime, and can create new threads. From an architectural point of view, a *frame* is a block of memory with a collection of registers for holding intermediate data. From a programmer point of view, a frame is used as a storage (like stack within a thread) of local variables of current procedure. A frame consists of several components, such as a frame pointer (points to the base of the frame allocated for a thread with a strand), a thread pointer (points to the base of the register set allocated to the thread, and shared among all strands in this thread), a register pointer (points to the base of the private strand register subset), and an instruction pointer (points to the next instruction to be executed). Each thread is associated with its own context that can be activated simultaneously. The HTMT frames are dynamic, and can be linked to one another in a non-linear structure.

4.2 Concurrency in the execution model

In general, an execution model serves as an interface between the system architects and the compiler designers. The system architects work on an efficient implementation of the program execution model. The compiler designers automate the process of translation of the high-level programs into the machine language corresponding to the program execution model.

4.2.1 Concurrent techniques

There are many standard techniques that could optimize the execution process at each level of such a system, such as: *register scoreboarding*, to detect dependences at runtime and support limited in-order multi-issue capabilities [11], *out-of-order execution*, to allow multiple instructions to be issued and executed out-of-order from a window into the code [12], *register renaming*, to enable anti-dependent instructions to run concurrently without changing the sequential semantics [12], *branch prediction*, to expose more instructions to parallel execution [13], and *data prediction and speculation*, to allow a speculation based on individual data items more fine-grained than basic blocks (as with branch prediction) [14]. Most of these techniques were considered and discarded from HTMT design because of complexity, when applied over such a large system, or lack of focus on the real design challenge - memory latency.

4.2.2 Multithreading

Multithreading is a competing approach to exploiting parallelism in processor architecture and directly attacks the latency issue. Modern multithreaded architecture models support the potential of simultaneous exploitation of parallelism at all levels (fine, medium, and coarse-grain), and an efficient and smooth integration of interprocess communication and synchronization with computation. Consequently it forms the basis for the SPELL design, where threads run concurrently and can be executed in parallel by separate execution units. Stalling a thread because of memory latency does not stall the entire processor. In

the HTMT, multithreading was chosen as the main approach to manage latencies in accessing memory and in communication between processing nodes.

4.2.3 Stranding

HTMT is a superstrand architecture [17] which exploits the notion of a strand. A *strand* is a block of instructions grouped together by the programmer (or compiler) to become a scheduling quantum of execution at the SPELL level. Strands are not ordered sequentially, but are scheduled according to data and control dependences: they are enabled at runtime when all data and control dependences are resolved. All instructions in a strand must belong to the same function. The activation of a strand does not require the allocation of separate storage, instead, using a set of SPELL registers for intermediate values, it is generated and passed among instructions within a strand. Once a strand is started, it runs to completion without stalls due to latency for branching or data access and synchronization, and instructions within a strand are executed in sequential order.

All strands within a thread share the frame associated with that thread, and use the frame's local variables or registers for passing data among themselves. The compiler partitions a program code into strands in a way that the source and destinations of a long latency data dependency are placed into different strands. This gives the hardware significant instruction level parallelism, reduces the number of potential dependencies which the hardware must check, speculate and resolve at runtime, thus minimizing the hardware complexity.

The HTMT hardware supports 16 threads at each SPELL which share CRAM memory, and each thread can access one context at the time. A thread can support up to 8 strands. When all strands per one thread are "done" executing, the thread writes the resulting data back to the memory and thread of control checks for a new available thread.

4.3 Execution model

There are two main components of the execution model that provide execution of the HTMT application programs: the percolation model and a communication mechanism that supports this model. In the *percolation model* data is transported through the memory hierarchy in such a way that programs located in the SPELL never wait for data and do not leave their memory regions to access it. The communication mechanism initiates the percolation model data transfers and provides support during program execution. Most of the functionality to support this mechanism is provided by the PIMs. We define the HTMT execution model in terms of functions that are performed by programs running in DPIMs, SPIMs, and SPELLs as presented in Figure 4.1. The numbers on the diagram show the order of events that are discussed below.

During execution, the functions performed by programs running in the DRAM PIMs (DPIMs) manage complete data sets in DRAM memory, disks, and 3D holographic memory (often performing simple operations on large data structures, or compression and decompression in the process of file transfers). DPIMs transfer selected subsets of data objects to different context buffers in SRAM (arrow 1), and perform

global synchronization operations. These operations recognize when all intermediate results from a set of processed contexts have arrived back in DRAM (arrow 7).

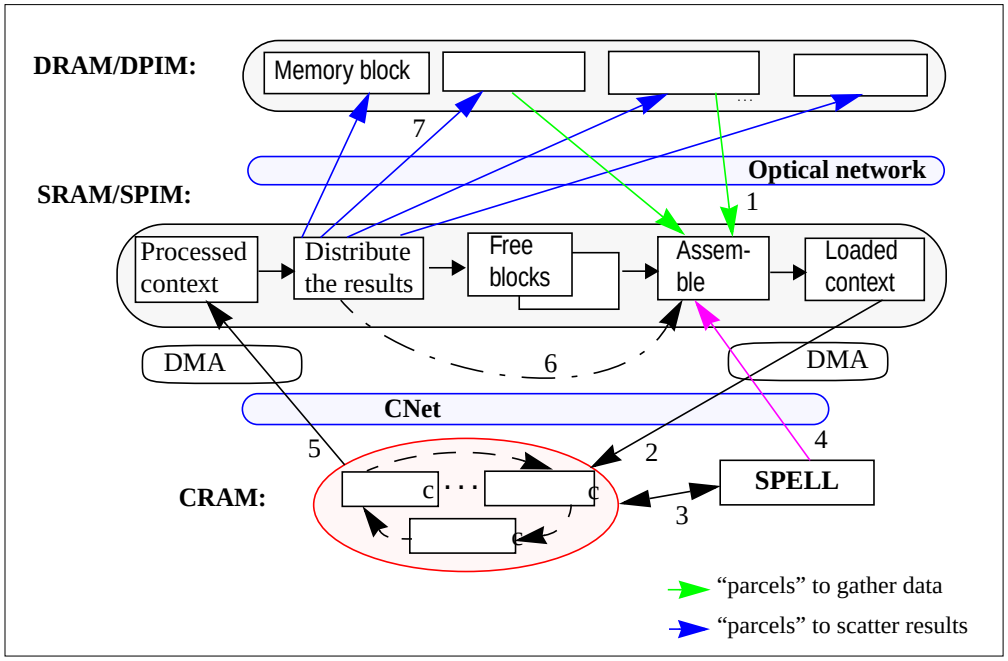


Figure 4.1 - HTMT execution model

The programs running in the SRAM PIMs (SPIMs) must keep track of usage of buffers in both SRAM and CRAM. They also select free buffers to be filled with data from DRAM (and provide the DRAM processors with the addresses of these SRAM buffers), and recognize when all necessary data from the DRAM has arrived (arrow 1). When a free CRAM buffer has been identified, SPIM moves data into ready queue for each SPELL, initiating DMA transfers from SRAM to CRAM buffers (arrow 2), and then signaling the SPELL that the current buffer is full (arrow 3). The computation process in SPELL continues on the new context. An SPIM recognizes when a SPELL has declared a CRAM buffer processed (arrow 4), and starts a DMA transfer of the results from that CRAM to a free SRAM buffer (arrow 5). When a DMA has completed, the SPIM starts scattering the results from a SRAM buffers back into DRAM (arrow 7), or to another SRAM buffer (arrow 6). When all results have been scanned from a SRAM buffer, SPIM marks it as free (arrow 6), so it can be refilled, and the process restarts.

4.4 Percolation model

The HTMT percolation model has two major objectives. First, it reduces memory access latencies by "smart" distribution and programmable prefetching of data. Second, it provides a balanced distribution of data and program instructions to the limited cryo memory at SPELLs by partitioning the application programs into self-contained fragments, which represent threads of the execution.

4.4.1 Principles

The percolation programming model takes advantage of the architectural characteristics and establishes the following principles. First, a program segment must be paired with data required by its

instructions before they are shipped to the vicinity of the fastest processors in the machine. Second, the programmer has a view of a global address space for the entire architecture, but the data movements should be coded explicitly and should use split phase transactions to prevent processors from additional cycles while waiting for long latency operations to complete. Finally, the program is explicitly parallelized by both programmer and compiler in such a way that the computation can be performed in threads, introducing thread switching for long latency operations.

4.4.2 Mapping percolation to the execution

Context percolation from DRAM, through SRAM to CRAM will be controlled by PIMs. Within this percolation model, the SPIM and DPIM processors take on different roles as pictured in Figure 4.2.

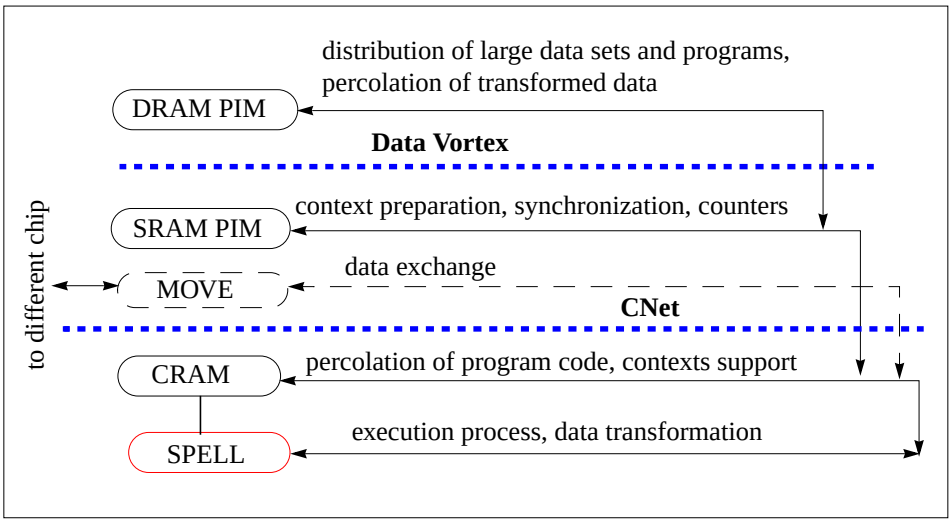


Figure 4.2 - Data accessibility in HTMT

Each context in an SRAM is mapped to a buffer from a buffer pool (Figure 4.1), and then matches to execution of one of the thread in associated with a SPELL CRAM. Both the CRAM and SRAM buffers are managed by a combination of the SRAM's runtime system (PIMOS) and code of the application program. The actual process of gathering data into, or scattering data out of, these contexts is managed by additional threads that run initially in the SPIM. Contexts are sent to the DRAM via parcels, and execute as threads there.

4.4.3 Outward and inward percolation

The HTMT data distribution process starts in the DRAM level. Data and code are selected and prepared in the DRAM region and sent to the SRAM level. Under the percolation model of execution, the entire computation is explicitly threaded. The *closure*, which includes data, program code, and control state needed to run a program, is moved into a *parcel*, a unit that is generated by a parcel threaded function which is written by user and resides at each PIM that is involved in computation. Then, the SPIM and DPIM processors can exchange the parcels, providing the execution through the HTMT system hierarchy, moving parcels towards the large memory storage units (*outward percolation*), or towards the fast processors (*inward percolation*). The percolation process is concurrent since new data transformations can take place in DRAMs while the previously transformed data is used by SPIMs and SPELLs. A pipeline structure can be

coded to allow the SPELLs to process blocks of data while the SPIM requests more data or stores results from previous computation.

4.5 Parcels

In terms of HTMT, a parcel (PARallel Communication ELement) is a unit of information that moves through and activates execution of closures in the memory hierarchy in form of threads. A parcel is associated with an object in memory, and consists of an address space identifier (an address in RAM), a command, and arguments/data (up to several thousand bits). The object contains data area and control area. It can also contain the code to be executed upon the arrival of the parcel. The command can be any of the memory functions at the PIM levels, including the communication with SPELL's level memory, CRAM. The operands include the indices, values, and names of objects (their addresses). Thread created by a parcel can create new parcels.

When a parcel is sent, the target object name is used to route the parcel. At the target object, at a PIM node, the command specifies on the two action types: "do something to some part of data area" and "use some entry in control area".

Second, a PIM can be asked: a) to perform some test and b) if test passed, perform an additional task. This additional code can be used to generate a new parcel, or to run some code that is waiting for some values that are used to perform the synchronization tests or to execute the code if the test in control area has passed. The structure of the parcel is presented in Figure 4.3.

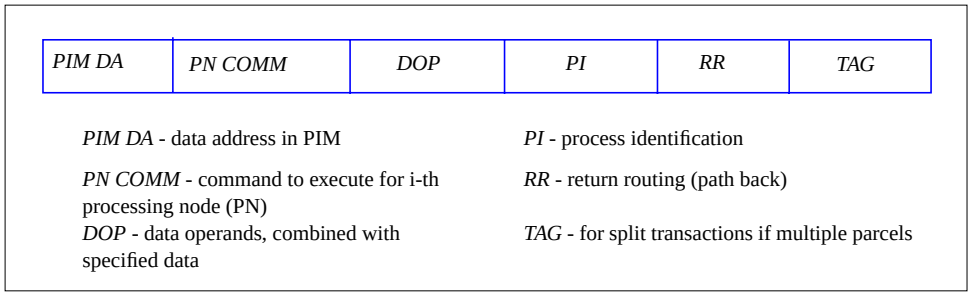


Figure 4.3 - A parcel structure

4.5.1 Parcel functions

The HTMT parcels provide a mechanism for moving data and control information through the memory hierarchy. They also support concurrency and parallelism as well as the ability to overlap communication and computation to combat latency issues. It is important to know when and where parcels are invoked in an application as they are used both for communication and to invoke computations in PIM nodes.

We distinguish between three kinds of parcels by the functions that they perform. First, there are parcel that perform traditional memory operations such as "load" and "store". Second, there are SPELL parcels to implement some special SPELL support functions, for example, to support SPELL suspension and release functions for threads of the execution. Third, there are more sophisticated application level parcels, such as parcels that can be broken into several concurrent parcels, where each parcel will perform a part of the original parcel task.

Primarily, there are four instances when parcels are created. In *control distribution*, parcels are often used to effectively distribute threads of control, and, thus, the execution control, over all the nodes in the system. In the *data distribution* case, a parcel with data is sent across PIMs to provide scattering operations as well as to support multiple loads for multithreading in user-level applications. In a *data movement* case, a parcel is created in response to a request from PIMs, typically, between levels of the memory hierarchy. *Remote method invocation (RMI)* sends a parcel with data to a specific PIM which triggers a specific method and invokes it with the correct arguments.

4.5.2 Parcel percolation algorithm

We outline the basis steps for the HTMT percolation model in Figure 4.4.

The algorithm for the HTMT percolation process.

```
1. Data transformation in DRAM
2. Inward percolation of transformed data to SRAM
3. while SRAM is not full (can accept parcels)
4.          A parcel threaded function in SRAM is enabled
5.          Data in SRAM is split into smaller chunks and transformed (again)
6.          The transformed data percolates inward to CRAM
7.          The code of parcel threaded function percolates inward to CRAM
8.            while CRAM can accept parcels
9.                  A parcel threaded function executes in SPELL
10.                 More data is requested from SRAM
11.                 Results are percolated outward to SRAM
12.         A parcel threaded function exits from CRAM
13.         Results are arranged in larger blocks and transformed in SRAM
14.         Results are percolated outward to DRAM
15. Data transformation in the outward percolated data in DRAM
```

Figure 4.4 - The HTMT percolation model

In this algorithm, the parcel threaded function represents the unit that initiates execution closures with parcels, and provides the multi-level concurrency in the system hierarchy. The percolation algorithm summarizes the execution model flow, presents the main execution steps at each system level, and will be modelled as a HTMT execution model programming prototype.

4.6 Conclusion

In this chapter we discussed the HTMT execution model which is explicitly multithreaded with two levels of concurrency (threads and strands), incorporates global memory address space, and explicitly exposes the HTMT memory hierarchy to the programmers. We described the two basic components of the execution model, the percolation model, which extends dynamic prefetching to allow the management of contexts, and parcels, the communication mechanism. Finally, we presented an algorithm for the HTMT percolation process which provides a skeleton for the HTMT execution model prototype.

CHAPTER 5

PARALLEL MODELS AND PARADIGMS

Many parallel processing models and paradigms have been introduced in the last three decades, virtually all of which resemble some level in the HTMT system.

In this chapter, we show how those existing models and paradigms are represented in HTMT, how we can apply their features when designing the execution model prototype, and how they can affect the development of the future software and runtime system support for the HTMT. We discuss multithreading in both hardware and software, the client-server model, the remote procedure call paradigm, and the main principles that were introduced by the message-passing model. We provide a comparison of the HTMT parcels and active messages. We also discuss Linda, which paradigm can be a candidate for the data manipulation operations at the HTMT memory. Finally, we give a short overview of the Petri Net model, and explain why we can use it for building a programming prototype.

5.1 Multithreading paradigm

Multithreading has been proposed as a promising processor execution model to overcome memory access latencies in parallel machines, and as a mechanism to overlap the computation with interprocess communication.

5.1.1 Hardware threads

The basic unit, a thread, is a unit of control for parallel execution. It has its own registers, may have its own stack, but typically shares address space with other threads. A thread may be created at any point of the program where an independent stream of control is needed and all data required by the thread is available. When one thread is suspended because of a long latency access to memory, another thread may be executed by the hardware at the next machine cycle. During the execution, the threads in a sequential code cooperate to solve a given problem. They are dynamically created, asynchronously scheduled according to data and control dependences, and concurrently executed. Depending on the hardware they need not come from the same application. From the programmer’s view, the threads can run simultaneously, and even on a single processor they can increase efficiency (and thus, throughput) during execution, such as in case of a shared data model when one thread is waiting for some low level event (a long memory access, a wait for a value from a highly pipelined function unit, or a synchronization operation between threads), and another is ready to compute.

5.1.2 Software threads

Multithreading in software indicates the ability to use the system hardware, allows concurrency and parallelization in the system and the program, and improves system and program resource utilization. The software threads are expressed with use of the operating system (OS), user libraries or language support. Multiple threads from a single application share their memory resources. Each thread starts its execution when all data to run this thread are available. Multiple threads may run concurrently with threads executing the same code on different pieces of data with threads that have different functions on different data sets.

On computers with multiple processors, such as a modern SMP node, there is potential for real parallelism, and threads can run concurrently if the underlying system supports it. On a computer with only one processor, threads aren't actually processed in parallel. Rather, at regular intervals or when the thread is waiting for something, the computation switches from one thread to another, giving an appearance of parallelism.

A thread may be created at any point of a program where an independent control is needed. The threads can run simultaneously, and even on single processor they can increase the performance during execution in case of shared data when one is waiting for some event (input/output, task completion by other threads, etc.), and the other is ready to compute. Such threads share their address space, have very fast context switch, and have no protection support besides application/compiler directives. The operations that can be performed on threads may include starting, stopping and release operations, and synchronization and set priority operations.

5.1.3 Multithreading in HTMT

The HTMT properties for simultaneous multithreading are motivated by the project design goals and include: (1) high performance, which implies both high throughput and low latency; (2) resource conservation, where stalled threads must use no processor resources; (3) deadlock-free, when sharing the same instruction scheduling unit; and (4) scalability, which provides that the same primitives be used to synchronize threads on different processors and threads on the same processor, even if the performance differs.

Looking at HTMT, multithreading is explicitly part of the SPELL architectures [9]. In fact, each SPELL has two such levels of hardware multithreading - threads and strands, which have support at the architectural level. It also plays an important role in the PIMs where access to data for multiple threads from SPELLs will need to be concurrent. In the HTMT multithreaded applications, the threads can improve resource utilization, such as memory and I/O, provide fast data decomposition and distribution, and increase the overall performance of user-level functions during computations.

5.2 Client-server model

The idea behind client-server, a subclass of the message-passing model, is to structure the system as a group of cooperating processes, called servers, that provide services to the user processes, called clients. An individual machine may run a single such process, or it may run multiple processes for multiple clients or multiple servers, or a mixture of the two. The client-server model is, actually, a subclass of the message-passing mechanism. In order to provide the communication between clients and servers, connectionless (no established path between clients and servers) or connection-oriented (data transfer through pre-set path) protocols could be used. They are both low-level in their request for service because the programmer needs to specify explicitly the server ID and its location. The most common server naming convention uses ports.

A server “listens” at a well-known port that has been designated in advance for a particular service. The client explicitly specifies a host address and a port number on the host when setting up the connection.

To avoid considerable state maintenance in the server and to support the large bandwidth for messages, the client-server model is usually based on a simple, connectionless request/reply protocol where the client sends a request message to the server, asking for some service. There are two cases possible for the client behavior. In the first case, the client blocks its execution and waits until a reply message from the server arrives. In the second case, the client sends the message and continues its work, which suites well for the requirements on increasing concurrency in the HTMT system with highly distributed data in the multiple memory hierarchy. When the server completes the work, it returns the requested data (a message) or an error code indicating why the work could not be performed to the application on the client machine. The reply message can also serve as an acknowledgment to the request.

In HTMT the PIMs, especially the DPIMs, are so far away from the SPELLs (latency to access data is huge), and will have so much internal processing capability that HTMT programming models may wish to view them as some sort of servers with the programs running in the SPELLs as clients.

5.3 Remote method invocation

Remote Procedure Call (RPC) and Remote Method Invocation (RMI) [10] allow programs to call procedures located on other machines. When a process on machine A calls a procedure on machine B, the calling process on A is suspended, and the execution of the called procedure takes place on B. Information is transported through the stub (files that are created by interface generator) from the caller to the callee in the parameters, and can come back with the procedure results. No message passing or I/O at all is visible to the programmer. A run-time library usually handles data exchange and other communications. An RPC can be built directly on top of unreliable, connectionless (datagram) services, requires explicit specification of the host as in connectionless and connection-oriented protocols for client-server model implementation, but allows a user to request a particular service on the host by name rather than by port number. This latter requires some sort of directory services to translate between programmer specified information (application, service, host) and an explicit port number and user handler at the receiving end.

5.3.1 RMI and threads

Threads combined with the RPC mechanism can increase parallelism for both servers and clients. Multiple threads on both sides can provide multiple remote procedure calls and services which will sufficiently improve the process utilization in the system. Today, some OSs (such as UNIX) provide possibilities for generating multithreaded servers which provide synchronization mechanisms, and dynamic parameters and returned data allocation during remote services. Plus, since more than one server can perform a particular service it would be convenient (in future implementations) if an OS could request the service by name and have the system designate the server. Again directory services are needed, but only using application and method name.

5.3.2 Implementation as an RPC

RMI is a RPC implementation of a distributed object design [73], which provides a way for client and server applications to invoke methods across a distributed network of clients and servers. Such an RPI was implemented for the Java Virtual Machine, where the RMI was distributed, and had an automatic management of objects with the ability to pass objects themselves from machine to machine.

The Common Object Request Broker Architecture (CORBA) is an another implementation of RMI which enables invocations of methods on distributed objects residing anywhere on a network, just as if they were local objects. A CORBA [90] implementation employs Object Request Brokers (ORBs), located on both the client and the server, which allow objects on the client side to make requests of objects on the server side without any prior knowledge of where those objects exist, what language they are in, or what operating system they are running on.

5.3.3 RPC in HTMT

In HTMT, for reasons similar to that for the client-server model, it may be appropriate for RMI-like calls to be made by the SPELLs to be executed in the DPIMs. This is especially relevant for operations which require multiple data accesses, but little processing per access, such as object construction and initialization, pointer-chasing, garbage collection, or even simple matrix operations such as scaling rows, columns, or subarrays. From another side, in the software development, it could be possible for PIMs to make RMI calls on SPELLs, mapping run-time system functions directly on top of underlying hardware functions.

5.4 The message-passing model

A communication network provides a means to send raw bit streams of data between nodes in a distributed memory parallel system. The message-passing model provides two basic primitives to pass messages, *send* and *receive*, to implement this. The *send* primitive has two parameters, a message and its destination, and is executed at the source of the message. The *receive* primitive has two parameters, the source of a message and a buffer for storing the message, and is executed by a destination of a message, usually before the send is executed. There is an explicit call to receive for each call to send message. To guard against lost messages, a receiver can send a special acknowledgement message, and the sender will wait for this special message within a certain time interval or will retransmit the original message. Duplicated messages can be recognized by putting consecutive sequence numbers in each original message. The receiver, then, can ignore a message with the same sequence number as the previous message.

5.4.1 Functionality

The message-passing model provides a highly flexible communication capability, and involves the following functionality: pairing of responses with request messages, data representation mechanism, resolving the addresses of clients and servers, and also taking care of communication and system failures.

When clients send a request messages to a server there are two different cases of client behavior are possible. In the first case, a client blocks and waits until a reply from the server has come. In this case we call the primitives as synchronous. In case of asynchronous primitives, the client sends the message and can continue its work. The communication primitives can be nonblocking or blocking. With nonblocking primitives, sending messages are copied from the sender to a buffer, returning control to the sender after that.

The receiving process can periodically check on message arrival or wait for a signal from receiving device. With blocking primitives, control to the sender program is not returned until the message is sent, the data copied to the receiver buffer, and a reply or an acknowledgment received. In the nonblocking case the programs have maximum flexibility to perform computation and communication in any order, but programming is tricky and difficult; the programs may become time-dependent. The big advantage is that there is the possibility for concurrency. In the blocking case, the programs are predictable, the programming

is easy, but there no flexibility in programming and few opportunities for concurrency between computation and communication.

5.4.2 Message-passing vs. client-server

Any process in message-passing models can be client or server. It means that each process in message-passing model has (software or/and hardware) support for both client and server functions. In the client-server model the software and hardware support for clients and servers differs, which simplifies the communication scheme as well as, partially, resolves the naming problem (in a case of very large number of processes, especially distributed across network), but limits this model in flexibility.

5.4.3 Implementations

One of the first programming notations to use the message passing concept was communicating sequential processes (CSP), which employed synchronous message transfers between pairs of *named* processes that is called a rendezvous [75]. This concept was implemented in the programming language Ada [76].

PVM (Parallel Virtual Machine) and MPI (message-passing interface) [84] are today's message-passing model implementation packages. Those software portable packages permit a heterogeneous collection of computers connected by a network to be used as a single large parallel computer. Thus large computational problems can be solved more cost effectively by using the aggregate power and memory of many computers.

5.4.4 Message-passing in HTMT

In the HTMT model, variations of messages are being designed, including parcels. A parcel, which is based on the active message distribution principle, initiates work on the HTMT units, and can result in creation of new parcels. Thus, a message-passing mechanism can occur in different places: between SPELLs, as a part of a client-server model over the optical network, or inside of PIMs, as message-passing between nodes in the memory to support a complex request from the SPELLs.

The message-passing paradigm could be used in an HTMT programming language which will enable the programmer to specify parallelism directly. These processes will be executed in parallel and will require control for the interactions between processes, communication (the transfer of data from one process to another) and synchronization (the transmission of information concerning the state of a process).

5.5 Linda

Linda is a coordination language that uses distributed data structures for synchronization and communication, where a distributed data structure contains separate groups of data objects that can be manipulated simultaneously by several processes. In general, process communication in coordination languages using Linda involves accessing the data structure contents (read operations), inserting components into the data structure (write operations), or extracting existing components for updating and

then returning them to the structure (read-write operations). Accessing the data involves providing a pattern, and using pattern matching algorithm to implement the operations on the data structure components.

Parallelism is introduced by allowing such actions to be performed by multiple processes on different components of the data structure. If a process tries to access a nonexistent data structure component (nothing matches the pattern), the process is suspended until another process adds the required component; this introduces process synchronization. In contrast with message passing, communication involving distributed data structures is anonymous.

5.5.1 Tuple space

Linda [35] gave an early introduction to distributed data structures for parallel programming, and is well known as an object-based distributed shared memory paradigm. To implement the distributed data structures, Linda uses the concept of a *tuple space*.

The tuple space functions as a kind of database, and its elements are referred to as *tuples*, which are simply ordered sequences of typed values. Tuples do not have addresses and are accessed using associative look-up methods by searching on any combination of tuple field value. The tuples can be of two types: data and patterns. Data tuples consist of multiple fields much like a conventional data base. Pattern tuples consist of conditions to match (like a query in a database system) and the name of the process that created the pattern. The tuple space is global to the entire system, and processes on any machine can insert/remove or search for either type of tuple without regard to how or where they are stored. To the user, the tuple space looks like a big, global shared memory, but the implementation may involve multiple servers. This memory is accessed through a small set of primitive operations that are independent of any base computing language and can be added to existing languages to form a parallel language extension to support and simplify the construction of parallel programming [97].

5.5.2 Implementations

Examples of Linda-based implementations include C-Linda, C++-Linda, Modula-2-Linda, and Fortran-Linda; these implementations also consist of a run-time kernel that implements interprocess communication and process management. Other languages were introduced after Linda that support the distributed data structure concept; these are Orca [52], SDL [96], and Tuple Space Smalltalk [95].

5.5.3 Linda in HTMT

The HTMT is a distributed shared memory model, with significant processing in the memory. Thus, using Linda functions, the search and memory management in the PIM memories could be done in parallel at the memory, and Linda’s operations can form the basis for implementing process communication and synchronization. A prototype of such a PIM system has been constructed [12], and demonstrated significant advantages.

5.6 Active messages

An active message is a message that executes a procedure when it arrives to the memory. The active messages model consists of sets of asynchronous processes running on different processors according to a local scheduling policy for each processor. Processors are connected by fast local area networks, and a process running on a processor may send messages to processes running on different processors by accessing the raw hardware controllers of the network devices with a minimal operating system overhead. Messages are “active” in the sense that the reception of a message on a node triggers the execution of a handler on the receiver processor, and the code of the handler is provided by the application programmer. Low latency and high throughput are the main benefits of this approach to communication (usually is provided by sockets).

5.6.1 The active message structure

In general, an active message consists of the head and a body. At its head, an active message may contain the address of a user-level handler which is executed on message arrival with the message body as argument. The handler gets the message from the network, and puts it into the computation ongoing on the processing node (with uniform code image on all nodes). It can be implemented as a privileged interrupt handler that executes on the message arrival making its execution very fast.

5.6.2 Active messages vs. software paradigms

In comparison with the matching buffered send/receive primitives, the active messages are not buffered, except as required by the network transport, allowing minimal overhead and real concurrency in communication and computation, and do not require the matching request messages. At the receiving side, the storage for arriving data is pre-allocated in the user program, or the message holds a simple request to which the handler can immediately reply. At the sending side, the sender launches the message into the network (blocking until message is injected to) and continues computing. Instead of performing the computation on a data as with general remote procedure calls (RPC), the active messages extract the data from the network and integrate it into the ongoing computation with a small amount of work.

An active message is an asynchronous mechanism that allows to expose the full hardware flexibility and performance of modern interconnection networks, and can be used to implement parallel programming paradigms more simply and efficiently. This mechanism was shown to be an order of magnitude more efficient than message passing primitives [21], and permits its overlap of the communication overhead by integrating the communication with computation.

5.6.3 Implementations

The simplicity of active messages and its closeness to hardware functionality translate into fast execution. Several machines were designed to implement them directly in hardware, such as J-machine [91], nCube/2 [78], Monsoon [92], SP/2 [77], or as the network interface, in SUNMOS [21], or in both (CM-5 [22]). Many different universities and organizations work on the implementation of active messages interfaces to support applications such as client/server programs, file systems, operating systems, and parallel programs. The active messages are currently used by UC Berkeley by the Fast Communication layers (such as sockets, RPC and MPI), the xFS parallel file system, the Split-C [27] and Id compilers, as

well as in other libraries like Scalapack. The Intel Paragon investigated active messages on a broad range of hardware, including a dedicated message processor per node (Intel Paragon and Myrinet) an FDDI interface at the graphics bus of a high end workstation (HP 735 with Medusa), and a conventional interface to the next generation LAN (Sparc 10 with Sahi-1 ATM), evaluated the active message communication interface and tested them on real programs on real machines [28]. An Active Message Driven Computing [103] parallel programming system, based on active messages, Itinerant Actors, and MEMO-all systems, was implemented at Illinois Institute of Technology. The GAMMA project [104] (a communication software for message passing) studied the active message concept [105] using a low-cost distributed platform composed of Pentium workstations connected by 100 Mb/s Fast Ethernet was to implement a network device driver for Linux.

5.6.4 Parcels - active messages in HTMT

The active messages in HTMT are called parcels, and represent messages that “travel” in the system, providing service for all HTMT system components (especially PIMs). In terms of comparisons to more traditional paradigms, parcels are more than a message in a conventional message passing environment because they are addressed to an object, not to a process or a node, and are interpreted more like a continuation than a simple data transfer. Also, unlike an active message, a parcel does not need a program-specified target node, since it is derived from the target address by the HTMT global translation mechanism. An additions, since the parcel arrival in the HTMT memory hierarchy can take several cycles, the synchronization mechanism for parcels will be significally more complicated than that of a simple semaphore as for the active messages.

Table 5.1 - Comparison of communication mechanisms

Feature	RMI	MP	RPC	AM	Parcel
Provide arguments	yes	yes	yes	yes	yes
Require send/receive coupling		yes	yes		
Address of a processor				yes (virt.#)	yes
Arguments *can* include data	yes	yes	yes	yes	yes
Address as a pointer (address to procedure)	possible	poss.		yes	yes
Changes the destination state (memory)	poss.	yes	poss.	yes	yes
Supported in hardware		yes		J-machine	HTMT
Supported in software	yes	yes	yes	CM-5	DIVA
Has built-in possibilities to create new messages upon arrival				yes	yes

Table 5.1 compares parcels with software paradigms, such as remote procedure invocation (RMI), message-passing (MP), remote procedure call (RPC), and active messages (AM). It shows that parcels provide all the possibilities that might be required by the system to make the HTMT interprocess communication flexible, specified, and sufficient to system execution model requirements.

5.7 Petri net model

The Petri Net model [29] is useful for analyzing asynchronous, non-deterministic concurrent systems with complex condition-event systems that include both control and dataflow, and can be used to

formally detect properties in the system [93][94]. The latter makes the Petri net model more powerful than other techniques such as FSM (a finite state automata). But, the Petri Net model has some limitations: it is more complex, and the decision problem results in undecidable and intractable problems concerning liveness and safety of resource allocation in the system. If the Petri Net model is extended with what is called zero testing instruction, it is equivalent to a Turing machine (the most general model of computation).

A *Petri Net* model [93][32] is a set of elements that consists of some resources that are needed to represent data and *places* for data to accumulate, perform transactions, and synchronization operations. *Transitions* represent synchronization points between tokens. A *token* is a basic Petri Net component that signifies that a particular place contains data, or some condition is true. Multiple tokens residing in a place are representative of the existence of multiple resources. A *place* represents a condition. If a place is a member of the input function of a transition, it is a pre-condition for that transition, or event, to occur. If a place is a member of the output function of a transition, it is a post-condition of the event or transition firing. A transition in the Petri Net model represents an event. The event (or firing) may occur when all the input places to the transition contain enough tokens to enable the input arc to the transition. The firing the transition will result in tokens being deposited through each arcs of the transition into the respective output places.

Like any grammatical or well-formed structure, a Petri net structure can be given several different interpretations, depending on the applications to which it is put. If a Petri Net structure is left uninterpreted, then it is useful for abstraction and analysis. The interpreted Petri Net structure is useful for simulation, description, data dependence and correctness questions. For example, a totally interpreted structure is useful to describe a multiprogrammed operating system, and an uninterpreted structure is adequate to study the absence of deadlock. In the case when both analysis and description are needed, we need to balance the levels of interpretation of the Petri net structure. When the Petri net model is constructed, it is a straightforward procedure to translate it into a simulation program. Consequently, the use of Petri nets simplifies the construction of a simulation program considerably.

5.7.1 Implementations

Variations to the basic structure of the Petri Nets can be added to better suit specific applications. Such variations of Petri Nets include Artifex [98], Algebraic [99], Communicating, Macronets, Queuing, Timed [102], Hierarchical [100], Object-Oriented [101], and others. The structure of these different types of nets include temporal extensions, places with tokens, places with token queues, timed transitions, colored tokens, tokens as objects (as related to programming languages), transitions labeled with transformations on state space variables, and many others. All those features can be applied to nets designing and modelling the specific systems.

5.7.2 Petri Nets and HTMT

Because of the HTMT system complexity, the execution model design and modeling as a whole system is a practically impossible task. We use the Petri Net model to represent HTMT execution model as a set of subsets, where each subset is a finite state or sequence of functions. An HTMT state includes all the necessary data and instructions. We can represent it as a Petri Net place. An HTMT transition between states is possible when all conditions are satisfied. At Petri Nets the conditions can be mapped into tokens, and transitions between places will be possible when there is a required number of tokens that has to be accumulated to perform this transition.

An access to each subset in the HTMT system model is possible if it satisfies the Petri Net properties for a given set. This principle allows us to model the HTMT system by dividing the HTMT

execution process into logically connected blocks, analyzing each block, studying the synchronization points during computations and in the communication between blocks, and by mapping those blocks into an HTMT execution model prototype.

5.7.3 Properties of Petri Nets

When building a prototype of the HTMT in Petri Nets we must be sure that this model is correct for further simulations and modelling. Some of the most important properties of the Petri Nets include safeness, liveness, boundedness, and conservativeness.

The property of safeness can be determined for both individual places and for the entire net. A place is said to be *safe*, if, for all possible markings, the number of tokens in that place never exceeds one. The Petri net is declared safe if all the places in the net are safe.

A place is said to be *k-bounded* if, for all possible markings, the number of tokens in that place never exceeds k. A Petri Net is k-bounded if, for any possible markings, the number of tokens in any individual place in the net never exceeds k. Both the safe and bound properties for Petri Nets are important in the field of engineering design. For instance, if all of the places are safe, it would be possible to implement these conditions with a boolean variable. If a place is representative of a buffer, and is 16-bounded, then the designer knows that buffer of size 16 will never overflow.

The *liveness* property encapsulates the concept of a system which will be able to run continuously (without deadlock). A Petri Net is considered live if, for all possible markings, there is always a transition enabled.

The *conservativeness* property is associated with the total number of tokens within a Petri Net. A Petri Net is said to be strictly conservative if, for all possible markings, the total number of tokens in the Petri Net always remains constant.

5.7.4 Petri Net's properties in the HTMT prototype

In terms of our HTMT prototype, we can say that the model has a reachability property if on a given initial marking for a corresponding program we can reach a marking which signifies correct termination. We say that the model is bounded if we have no more than *k* tokens in a place (finite storage). An initial marking is the first number of tokens (parcels in the HTMT) initially transmitted by a program, where a marking is a mapping of tokens to places which defines a state for the model. Our system has to be free of deadlocks, and for all possible markings, there should always be a transition enabled that makes the system live. Our model is said to be safe because for all possible markings there are no places in the model where the number of tokens exceed one, since there is only one parcel can arrive at a place at any stage of the HTMT execution process. Our model should satisfy the conservativeness property, since for all possible markings, the total number of tokens in our model always remains constant *K*.

5.8 Conclusion

In this chapter we included the short description of existing software models and paradigms which features and algorithms can be found at each level of the HTMT system design and implementations. The structure and the idea of execution in active messages, for example, are used in the HTMT parcels to provide communication, code percolation and the data exchange in the system hierarchy, while the Linda tuple

model idea can be used for the HTMT memory algorithms for pattern matching and recognition. Modeling the HTMT architecture execution model with Petri Nets will minimize the complexity of the design and the implementation of the concurrent HTMT programming prototype. Also, building a system generic model using Petri Net formalism is related to formal language theory, and every regular language is a Petri Net language. This allows us, in the future work, to start modelling a new HTMT programming language.

CHAPTER 6

CLIENT-SERVER MODELS IN AN EARLY HTMT PROTOTYPE

In this chapter we discuss the importance of building an early prototype for a new system. Next, we introduce a client-server relationship for modeling DPIMs and SPIMs, where each of them can access the necessary data through the interconnection network. Then, we show such relationship for SPIMs and SPELLs. Finally, we present the HTMT prototype as a client-server model, describe its object-oriented nature, and explain how we used the Java programming language and its libraries to implement it.

6.1 Prototyping

In general, prototyping is very suitable early in the design phase. Its purpose is threefold. First, it identifies requirements for the new system. Second, it helps users to study the system. Third, it allows developers to build a hardware or software, model, combining well known parts of design with undefined ones, to get a global view on the system. Any prototyping process is evolutionary in that it allows the designers to build an interface, connecting separately designed parts, quickly, and provide a first glance on the capability of the system.

A prototyping process includes the following stages: defining system concepts and requirements; prototyping system architecture; simulating the execution models, runtime and compilation tools; analyzing constraint violations and error preventions; and gathering statistics on system behavior.

Prototyping is a necessary step in the HTMT system design. We need to analyze and model new hardware architecture demands, create and study new programming models and execution models, analyze and, eventually, generate the code. Building an early HTMT prototype can prevent costly design errors. Since the HTMT model is being built in its early design stage, many aspects are undefined. Prototyping allows us to analyze the system, to gather its requirements at the beginning stage of design, to estimate its future performance, and to collect and analyze the data for building an adequate system. The final HTMT model prototype must provide the full picture of the system, and allow us to model different tasks on its platform.

6.2 Client-server models

Designing the early HTMT prototype, we distinguished between three models. The first model is the HTMT system model, or a *structural model*, which is based on the architecture, organization, and overall execution flow of the HTMT system. We discussed the details of this model in previous chapters. A *functional model* describes the behavior of HTMT by incorporating the structural and execution models of the HTMT system. The third model is an *implementation model*, which provides details on the functional model, considering the programming language that will be used in the HTMT, the communication

mechanisms, and memory models. This model will allow us to gather statistics on different aspects of the HTMT and will serve as a basis for an early HTMT prototype.

6.2.1 The functional model

In the process of building the early HTMT prototype, we need to define the execution mechanism, identify different communication methods for system components, and analyze the data distribution, communication, and data migration at each HTMT level. All of these are addressed in the functional model.

The functional model is presented in Figure 6.1. In this scheme, we assume that the data of any application are loaded into DRAM distributed data structures. The program instructions are loaded into

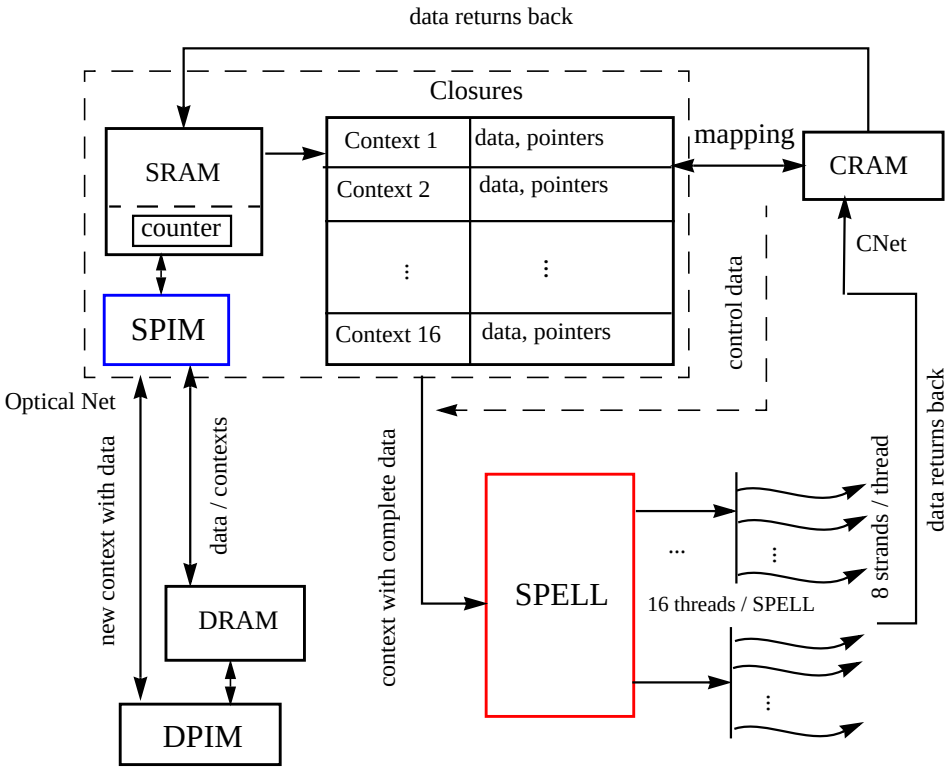


Figure 6.1 - The functional model

DRAMs and distributed by DPIMs to SRAMs as closures for execution in SPIMs and SPELLs. Since the number of unique closures for an application in the prototype is assumed to be defined at compile time, a counter can be used to determine the end of loading by decrementing the “counter” value by the number of already loaded closures to SRAMs (we assume that there are no multiple copies of closures).

When loading of the closures to SRAM buffers is completed, SPIMs can perform additional code and data analysis and distribution for computationally intensive tasks as well as for vector operations or the operations on small data sets. Then SPIMs pre-load CRAMs by mapping SRAM closures to CRAM buffers, and signal to SPELLs that the execution of the application can begin.

The concurrent multiple threads in SPELLs can execute multiple contexts at the same time. When the CRAM buffers are filled with the data from the computation, the data can be mapped back to SRAMs

and new closures can be loaded to CRAM and SRAM. When the task execution is completed, results are sent back to DRAMs.

In this scheme the DRAM/DPIM level performs mostly the data and the application code loading, analysis, and distribution, while the SRAM/SPIM level manages the control of the execution and provides the workload for SPELLs. These levels differ in their functionality, and can be distinguished as *data*, *control*, and *compute* processing levels. Each such processing level communicates with another via parcels through the network or by the direct data mapping. It allows us to view the HTMT system as a client-server model.

6.2.2 The implementation model

An interaction between any two levels in the HTMT system hierarchy (DRAM and SRAM PIMs, or SRAM PIM and SPELLs with CRAMs) can be treated as a client-server relationship where each client signs on a server by sending messages (parcels) or a hardware signal, and the server assigns jobs and returns necessary data. The clients execute their portion of the original task (via processing in the SPELLs) and transfer results to the server for further computations and manipulations. At the same time, the server can execute some tasks itself using its own concurrent threads. Depending on what level of the HTMT memory we choose, two special cases for this client-server model relationship can be considered: shared-memory and distributed.

6.2.3 Distributed memory

In this case, we consider multiple PIMs that are connected through a communication network, and where accesses to different objects at different memory levels may require explicit messages between processors.

We apply this intercommunication scheme to the DPIM-SPIM levels where the multithreaded DPIM servers can explicitly exchange data over the optical network with the SRAM memory at SPIMs using a datagram protocol. The same may be true for SPIMs during the access of DRAM memory, where SPIM clients can request data exchange through DPIM's servers (Figure 6.2(a)).

In the case of simple loads and stores, this model does not require extra communication threads for handling the sending and receiving messages (especially data) across network, but, in the "real" HTMT system, it still will need some mechanisms to distinguish between processes on clients and servers in order to provide the correct data exchange and to control timing for reliable scheme of interaction.

When more complicated memory accesses occur, such as accessing a large object, parcels are needed. In this case, besides the local data, each server sees the data that were sent from the client, and executes only the task which was specified in the parcel.

In our model, the server is implemented as a multithreaded system with multiple threads to handle sending and receiving messages for each client during the data exchange process, and also to control the execution process.

6.2.4 Shared-memory

In the HTMT system, SPELLs and SPIMs communicate through a communication network. This can be seen as message-passing. But, the most of the memory interactions between SRAMs and CRAMs are

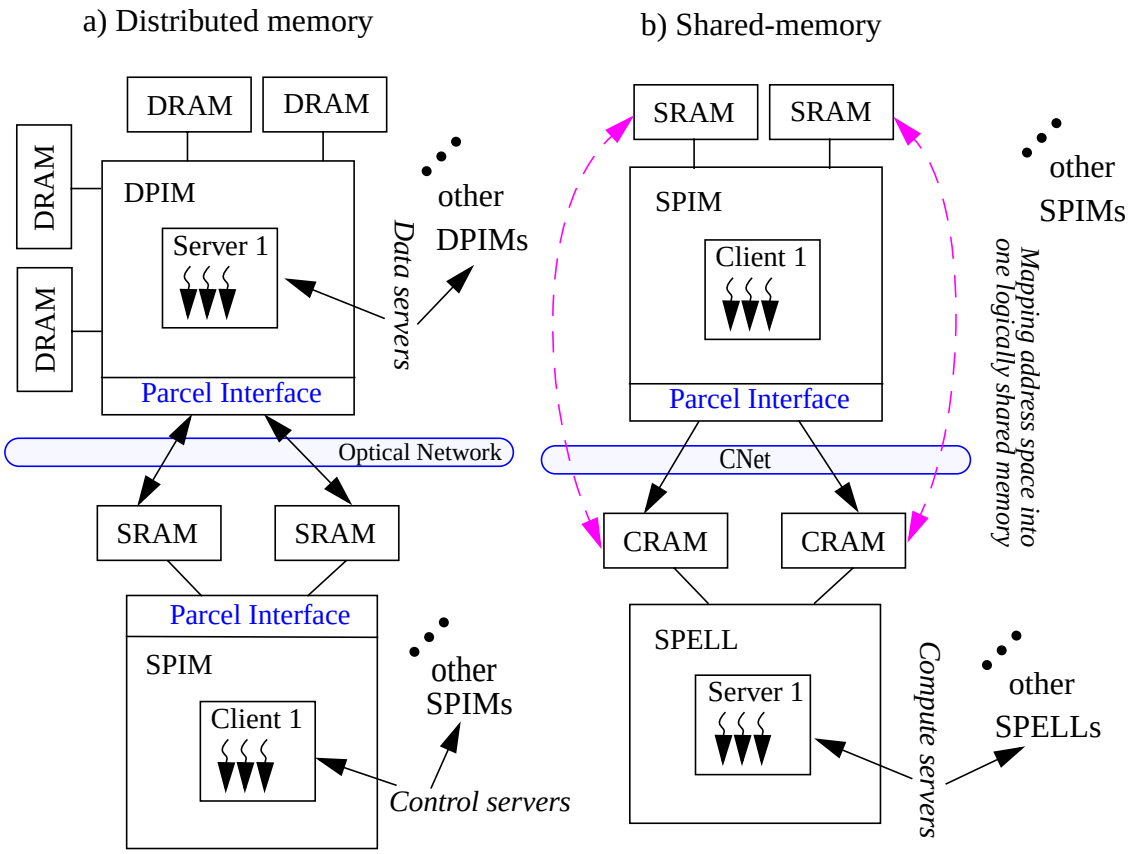


Figure 6.2 - HTMT as a client-server model

performed through memory mapping operations. That is why we view this relationship as a shared-memory, where all SPELLs "see" the same memory through simple CRAM loads and stores from SRAMs.

In this case multiple SPIMs "share" their memory with memory at the SPELL level, and even though SRAMs and CRAMs are physically distributed, the SPIM and SPELL processors treat them as one memory storage. We distinguish the CRAM from SRAM by allocating a separate range of addresses in the SRAM address space.

We define SPIMs as clients which request services from SPELLs via "pre-loaded" information in CRAM closures. When an SPIM level parcel arrives to the CRAMs, the processing in the SPELL compute servers occurs, and the results from the SPELLs are written to CRAM buffers. Then SPIMs can leave the results in local memories or map it to SRAMs (Figure 6.2 (b)).

6.3 The early prototype structure

The early HTMT prototype structure is presented in Figure 6.3, where DPIMs serve SPIM multithreaded application clients, providing them with the data, while SPELLs execute SPIM clients contexts as compute servers. Both DPIM and SPELL servers are multithreaded with specified thread management functions and provide service to multiple multithreaded clients in SPIMs via parcels. The parcels used in the DPIM/SPIM interactions are called *distributed* parcels (*D*-parcels), while those used between SPIM and SPELL levels are called *shared-memory* parcels (*SM* parcels) as shown in Figure 6.3 (a).

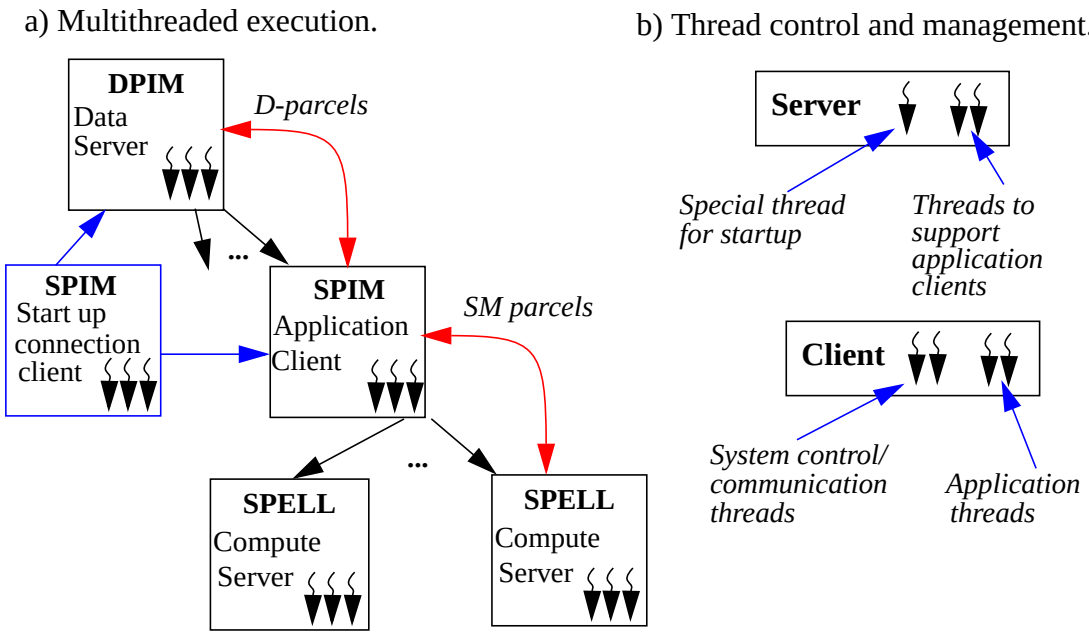


Figure 6.3 - A prototype as a client-server model

When the data and code are loaded from the front-end computer which initializes the work, the start-up connection client in SPIM activates a special thread that begins the application execution from DPIM level through D-parcel exchange. The application clients at SPIM which manage the execution process and the data transformation inside of applications, can distinguish between D-parcels and SM parcels. The latter can be sent to SPELLs as a special request to access the data or to execute the sequence of instructions from the application.

From the application point of view, we distinguish between communication *control* threads that handle the data and, possibly, program distribution functions and *computation* threads that execute contexts with ready data, as shown in the Figure 6.3 (b).

6.4 Object-oriented design

In comparison to the functional design approach, where a project is described in terms of functions applied to data, an object-oriented design approach uses objects to encapsulate the data and defines methods that manipulate the data in the objects, and describes the execution process as an interaction between the objects.

We refer to our work on the early HTMT prototype as an object-oriented design. First, we put the data and the execution code together which creates an object. A set of such objects compose the contents of storage for a PIM, and the definition of these objects is a description of the PIM's structure. Second, we provide an object communication feature when we move the data and execution code together through the system and memory hierarchy from memory storages closer to the processing in SPELLs. Third, we specify all data accesses by the object address instead of by a conventional memory location/address. This scheme is used in the HTMT execution model. For example, when a parcel arrives at a PIM, we trigger the data structure and invoke the method specified in the parcel (the name of the object that we have to access). Another example is large data structures in DRAMs where the data are accessed by providing the object name that finds the meta-data for the object, which in turn controls the actual data distribution across the

memory hierarchy. Fourth, different objects are related to different specified classes according to the class properties. For example, there are objects in the SPIM class. These objects are related by the property that their methods will be executed in SPIMs. Fifth, the execution process is described by the object interactions, where methods of an object can access the data or invoke methods of other objects. Such interactions are possible during data transfers through the memory hierarchy or during parcel exchange in the system hierarchy.

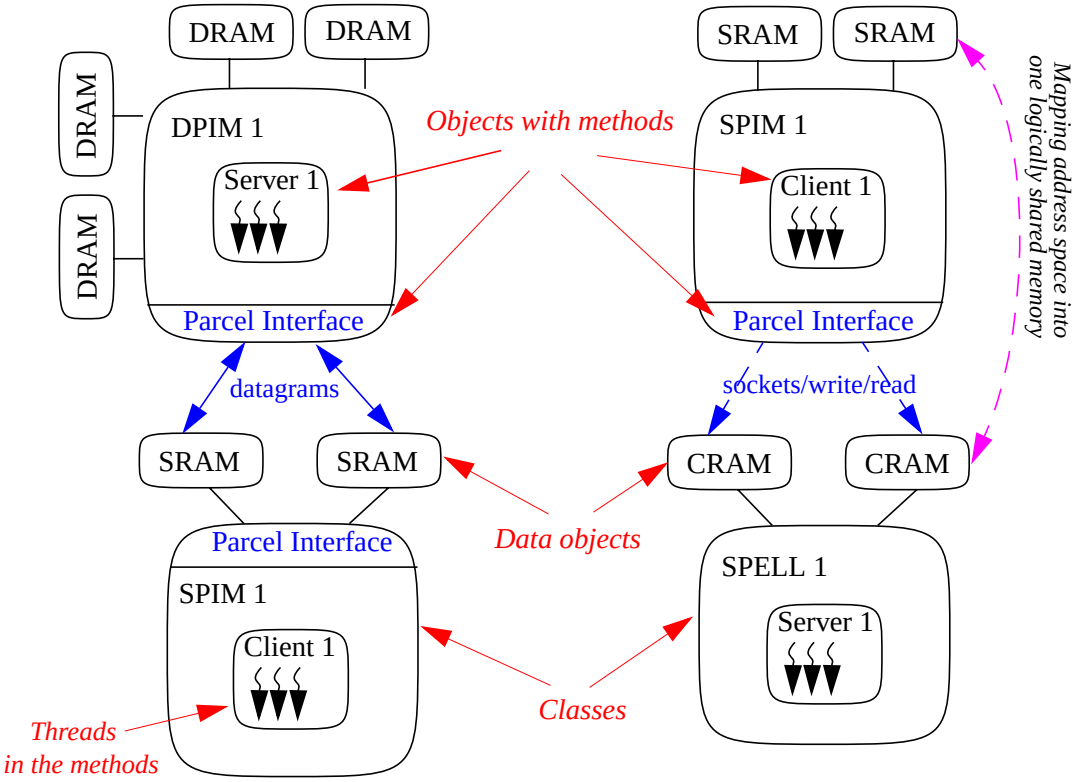


Figure 6.4 - The early HTMT prototype

We view all clients and servers in the HTMT prototype as objects with data and methods that work on the data (Figure 6.4). The objects communicate by requesting services from other objects through the parcel interface which is itself an object with methods.

For the distributed memory model in our prototype, we use datagrams as a communication mechanism to simulate the parcel percolation and inter-node communication. In case of shared memory, we used sockets or direct write/read functions to mimic the HTMT write and read operations.

6.5 Java

We chose Java as a programming framework for exploring HTMT models, especially those derived from the PIM-enhanced memory hierarchy, because of its unique features to support different kinds of concurrency and its growing use in the distributed and parallel programming environment. Java is a powerful object-oriented language that eliminates dependency on any one particular platform, while its dynamic nature provides maximum flexibility, permitting the analysis of the concurrency in the HTMT and allowing the object presentation of the design. Java provides the necessary interface to implement this fairly simply, giving difficulties only in distribution of data between clients and in the synchronization of execution processes.

In this section we provide a short overview of Java threads, datagram, and socket packages, and discuss the RMI and MPI wrapper packages as a part of the future work on the HTMT prototype.

6.5.1 Threads

On computers with multiple processors, there is potential for real parallelism of Java threads if the underlying system supports it. All modern operating systems allow multitasking and multithreading. The support for multi-tasking is built into the Java language. Thus, applications written in Java are MP-hot, which means they will benefit if they are executed on a multi-processor machine.

The Thread Class is a datatype for a thread object (a variable of that type) that acts as a control for an executing thread. This class includes methods that define, initialize, manage threads, and execute threads. When multiple threads are accessing data or invoking methods that affect the state of an object, it is necessary to protect the data from simultaneous access by multiple threads to preserve a consistent state. By declaring a method as *synchronized*, Java will automatically block other threads from invoking the synchronized method if it is already being executed by one thread. When the synchronized method terminates, one of the threads that are waiting can proceed to execute it. A monitor object associated with each such object acts as the gatekeeper, letting in only one synchronized method at the time.

Threads in Java can also have priorities, which means that threads with higher priority will run over threads with lower priority. Priorities are useful in certain cases when certain threads need to execute whenever they aren't waiting on anything. They should be viewed as hints to the system as to which threads should be granted CPU time first.

Inter-Thread Communication in Java includes the following methods: "wait" where the executing thread gives up the object monitor and waits for a notify, "notify" which wakes up the first thread that called wait on the same object, and "notifyAll" that wakes up all the threads that called wait on the same object.

6.5.2 Datagrams and Sockets

The *Uniform* (User) *Datagram Protocol* (UDP) provides a mode of network communication whereby applications send packets of data, called datagrams, to each other. The DatagramPacket and DatagramSocket classes in the java.net package implement system-independent datagram communication using UDP. Sockets for Clients can be used to connect Java's I/O system to other programs that may reside either on the local machine or any other machine on the Internet. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes: *Socket* and *ServerSocket* that implement the client side and the server side of the connection, respectively. A ServerSocket will wait for a client to connect to it, whereas a Socket will treat the unavailability of a ServerSocket to connect to as an error condition. Sockets for Servers will wait at known addresses and published ports listening for local or remote client programs to connect to them.

Datagrams are high performance "fire-and-forget" bundles of information sent over the network from a server to a client. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from one machine to another may be routed differently and may arrive in any order. DatagramSocket and DatagramPacket are classes that include Constructors and several access methods to support datagrams in Java.

6.5.3 Java wrapper for MPI and RMI

The Java wrapper for MPI package can be used for future simulations on the HTMT prototype when several physically distributed nodes can be added to the design scheme. This package consists of a small set of classes with a lightweight functional interface to a native MPI implementation. The classes are based upon the fundamental MPI object types (e.g. communicator, group, etc.). There is a one-to-one mapping between MPI functions and their Java wrapper bindings, and the Java wrapper for MPI functions are method functions of MPI classes.

Remote Method Invocation in Java enables the programmer to create distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. RMI uses a native-transport protocol, and can only communicate with other Java RMI objects. A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in a bootstrap naming service provided by RMI or by receiving the reference as an argument or a return value. A client can call a remote object on a server, and that server can also be a client of other remote objects.

RMI is the simplest and fastest way to implement a distributed object architecture due to its easy-to-use native-Java model. Therefore, it is a good choice for the small-sized applications that are implemented completely in Java.

6.6 Conclusion

In this chapter we showed that early prototyping is a necessary step in designing large systems such as HTMT, discussed the early HTMT prototype models, and introduced client-server models for modeling DPIMs, SPIMs, and SPELLs. We compared the object-oriented design with our HTMT prototype, and gave a short overview of the features of Java language and its packages which we used to implement the client-server models as a part of the object-oriented design for the HTMT prototype.

CHAPTER 7

THE EARLY HTMT PROTOTYPE ANALYSIS

This chapter includes analysis of our early HTMT prototype. We study the system prototype using Petri nets and discuss the details of different prototype parts which allow further simulations. Finally, we explain the transition from Petri net model to the programming simulations.

7.1 Alternative approaches

We need to find a model that can represent the HTMT prototype features discussed earlier and help to check correctness of the design. Most typical solutions are not acceptable for prototyping the HTMT. For example, a tracing tool for study different kinds of architectures, such as Shade, can provide an accurate information on how a program runs on a given architecture, and the SimpleScalar toolset allows detailed simulations for different architecture specifications, but they are useless for radically new architectures exhibiting multiple concurrent execution and massive parallelism. Plus, the HTMT architecture is not yet well enough defined to permit the detailed simulations.

At another level, many graphical and mathematical tools can be used to model and analyze large parallel systems. For example, a Communicating Sequential Processes (CSP) model is based on the idea of several “regular” sequential processes that are running in parallel. In this context, it is not important whether these processes are really running on distinct processors or if they are scheduled on one processor by a multitasking environment. Two major problems that affect such model techniques include: how to communicate (transport data between the processes) and how to synchronize certain actions. The finite State Machine (FSM) approach is a mechanism that represents a mathematical engine to describe the system behavior. The initial state and input for this machine are known, and the set of final states is also defined. Using FSM transition table we can only model the transitions between states.

In contrast, the classical Petri net model allows modeling of states, events, conditions, synchronization, parallelism, choice, and iteration. Extensions of the basic Petri net model allow us to model data, time, hierarchy, and structure of large models. For example, with *Stochastic Petri nets* we can model asynchronous distributed systems and simulate real cases of distributed systems to study their performance [32].

7.2 Questions to answer

The main questions that we answer by creating the HTMT prototype include, but are not limited to, the following. How does the HTMT system start up when the power has just been turned on? What starts the initialization process, where, and how? What are the pieces of the system programs that are preloaded and how are they preloaded? What pieces of code might be executed where, and how is that code loaded? What functions need to be provided natively in the PIM nodes? What functionality might be needed at run time, especially in the SRAM layers? What is the data flow in the system, and do we need some kind of garbage-

collector to manage the memory relocation during the execution and after the execution of the application? What kind of application code is expected at this level of the HTMT design, and how accurately should it be implemented? How can the results from this prototype be interpreted for the future use (OS “parcel-functions”, runtime “parcel-functions”, compiler expectations, etc.)? The answers to these questions are the skeleton of the new HTMT prototype.

7.3 Application view

To demonstrate the prototype we consider matrix multiplication, where $C_{ij} = A_{ik} \times B_{kj}$. The data flow for a simple matrix multiplication in the HTMT prototype model is shown in Figure 7.1, where each context in SRAM and CRAM contains the following information: 1) the program code which multiplies each element of a row by a corresponding element of a column, 2) the data which consist of the row and the column, plus a pointer to where the result should go, 3) control information including context id, time stamp, etc.

We assume that in the beginning the machine is cold, and the OS of the front-end computer (FEC) initializes the machine before starting an application. The matrices A and B arrive from an external source in well defined structure, size, destination path, and source information. All three matrices in our example can be of arbitrary size. They can require multiple node (DRAMs or SRAMs) for allocation, and a mechanism to assign the data to potentially 10’s of thousands of PIMs (tiling information). The communication is provided by the parcel functions, and only parcels can be used in the system to exchange information between different system nodes. At different levels, PIMs have “built-in” methods that can recognize the messages that arrive at the node, and what action, based on the parcel context, should be taken on those messages.

When given control in execution, each SPELL thread spawns multiple strands. In our example, each strand multiplies one element in the row by one element in the column. In the case of the large matrices, the matrices can be distributed between DRAMs in blocks or small submatrices that fit in registers, then they are distributed across SRAMs in blocks and rows (or columns), and the multithreaded SPELLs will manage not the individual elements but subrows or subcolumns. The last strand puts the sum of the products in the output queue for CRAM with a completion code (“done”, “intermediate”, etc.). These data can be used for further calculations or can be stored in SRAM/DRAM. The SRAM / DRAM servers create the contexts and percolate them up and down the simulated memory hierarchy. The computed matrix C goes back as the result of the task.

7.4 Analyzing HTMT using Petri nets

A Petri net model (Figure 7.2) allows us to view the HTMT system as a set of independent interacting modules which can be considered, designed, and tested separately. Each of these modules can consist of functions for scheduling, loading, management, and synchronization on the arriving data. Hence, this modularity provides an opportunity to debug each module separately, and to get detailed statistics and traces of execution flow.

In our model, each place in the Petri net represents a state of the HTMT memory after the transition was fired. The arcs indicate that the transitions between given places are possible, and the data can be moved between these two places. In our model, tokens represent messages in the system (signals and parcels) that can be moved through the arcs into places, and initiate new transitions if all conditions for this action are satisfied (Figure 7.3). Tokens must reside on all input places of a transition to enable it, and the associated condition written next to the transition must be satisfied to fire the transition, which means execution of a parcel with potential generation of new parcels.

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \qquad B = \begin{bmatrix} e & f \\ g & h \end{bmatrix} \qquad C_{ij} = A_{ik} \times B_{kj}$$

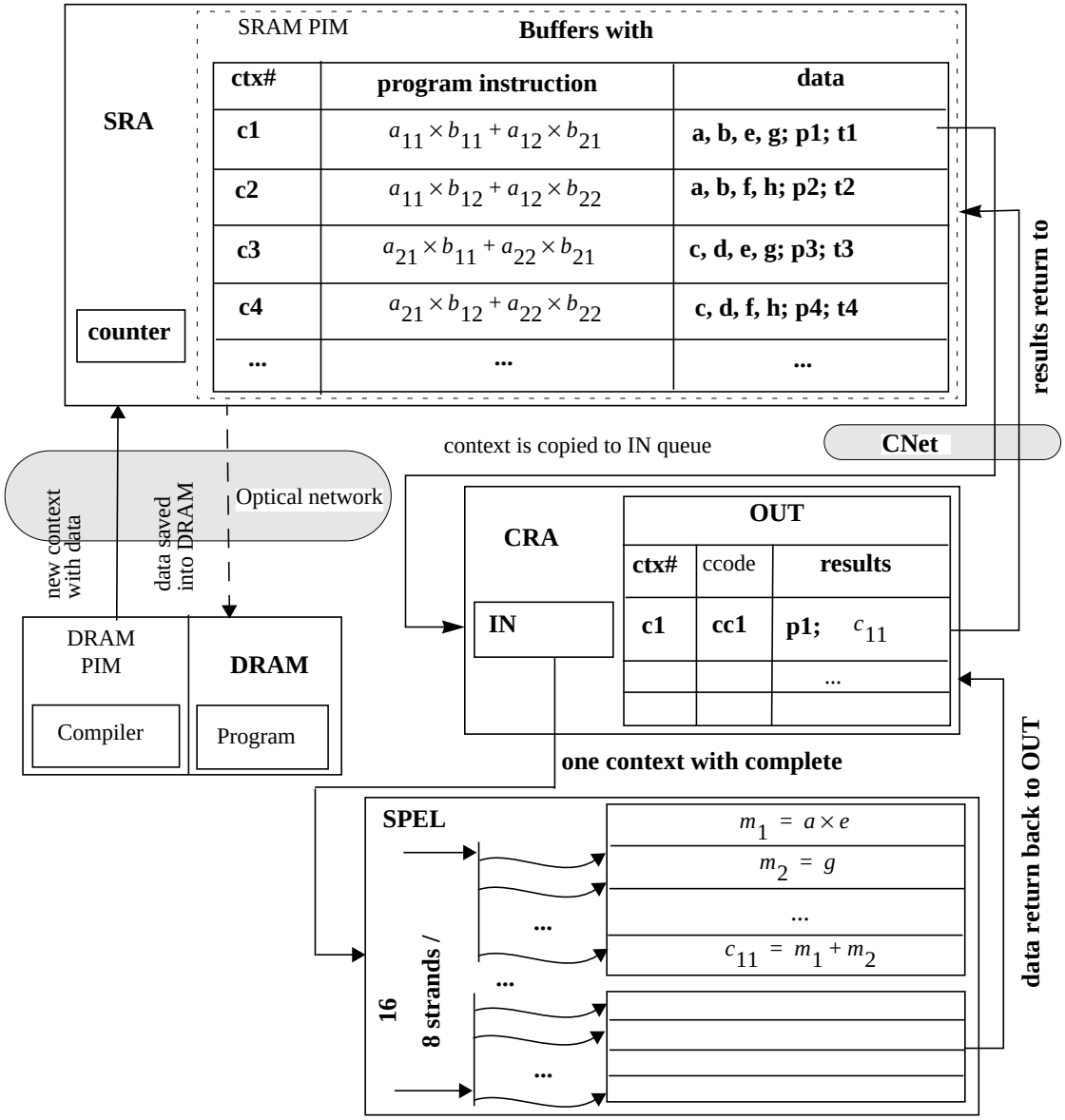


Figure 7.1 - Matrix multiplication operation with one SPELL

Such representation of the HTMT system allows us to accumulate statistics at different levels in execution process, such as when a SPELL is idle, what part of the system is busy, and when. Expanding this model to a large number of nodes increases the complexity of the model, but at the same time provides us with the tool that allows to build a programming prototype that simulates the system execution flow, and to describe the system in a mathematical language.

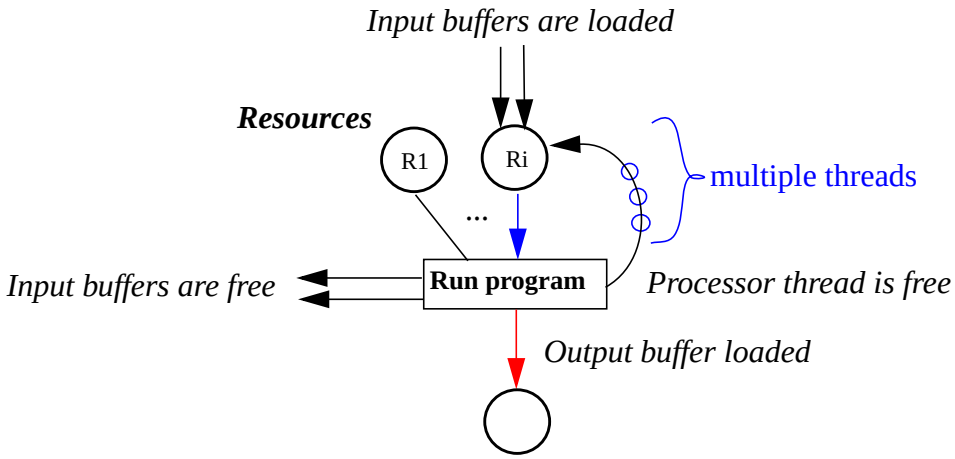


Figure 7.2 - HTMT prototype transaction in Petri net model

7.4.1 SRAM PIM level

DPIMs load closures from local DRAMs into SRAMs through input queues, 16 closures per SRAM which can be modelled via tokens, where 16 tokens represent parcels that arrive to the HTMT state or to a place in the Petri net model. When a transition is fired, a token can be moved across to the level below to indicate that the parcel was accepted and its data can be accessed. The multiple closures that reside in the SRAM input queue are ready to be executed. An SPIM sends to its assigned SPELL a parcel with data and name of the function that must be performed. Once a context is copied to CRAM, a new context from DRAM can be loaded to SRAM buffer through a parcel request. When a SPELL thread completes the execution of one closure, it writes the results into CRAM buffer, and signals (via parcel) to SPIM that computation is performed. SPIM, then, moves these results to its own output buffers in the output queue or initiates the transfer to the output buffers of some other SPIM and sends a request to DPIM to update the meta-data table.

In case of multiple SRAM nodes, an SPIM signals to a SPELL that its buffers are full with context data and it is ready to perform some action. The same SPIM copies data from its SRAM buffer into CRAM (assigned to the SPELL), and sends a parcel to start instruction execution on a given data set. When required action is performed, the SPELL sends the results to the SRAM PIM node. If the buffer in the SRAM is busy/ full, then a buffer in another SRAM can be used to place intermediate or final results from the SPELL.

The problem with this model design is how to present the parallel dispatch of messages to the SPELLs. This model needs some kind of mutual exclusion where only one SRAM PIM at a time can access the CRAM memory (at SPELLs). Moreover, the SPELL should be able to distinguish between PIM nodes by address in the context. In Figure 7.4 we show the Petri net representation for the mutual exclusion solution for any two SPIMs that access a common CRAM. If the SRAM buffers are available at SPIM1 and a parcel arrives to SPIM1, the loading of CRAM buffers is provided only by SPIM1, and results are sent back to SPIM1 if condition on free available buffer is satisfied. Otherwise, if this conditions is not satisfied for SPIM1 but it is true for SPIM2, then the results from SPELL are sent to SPIM2. When parcels arrive to both, SPIM1 and SPIM2, at the same time, and they both have available buffers to load data, only one of them (first) can access the CRAM buffers.

Constructing a working Petri net model forced us to study several approaches on how to build a model that will allow us to represent multiple PIM nodes at SRAM level of the memory hierarchy, and to demonstrate the concurrency in the shared memory system that can be, for example, supported by mutual

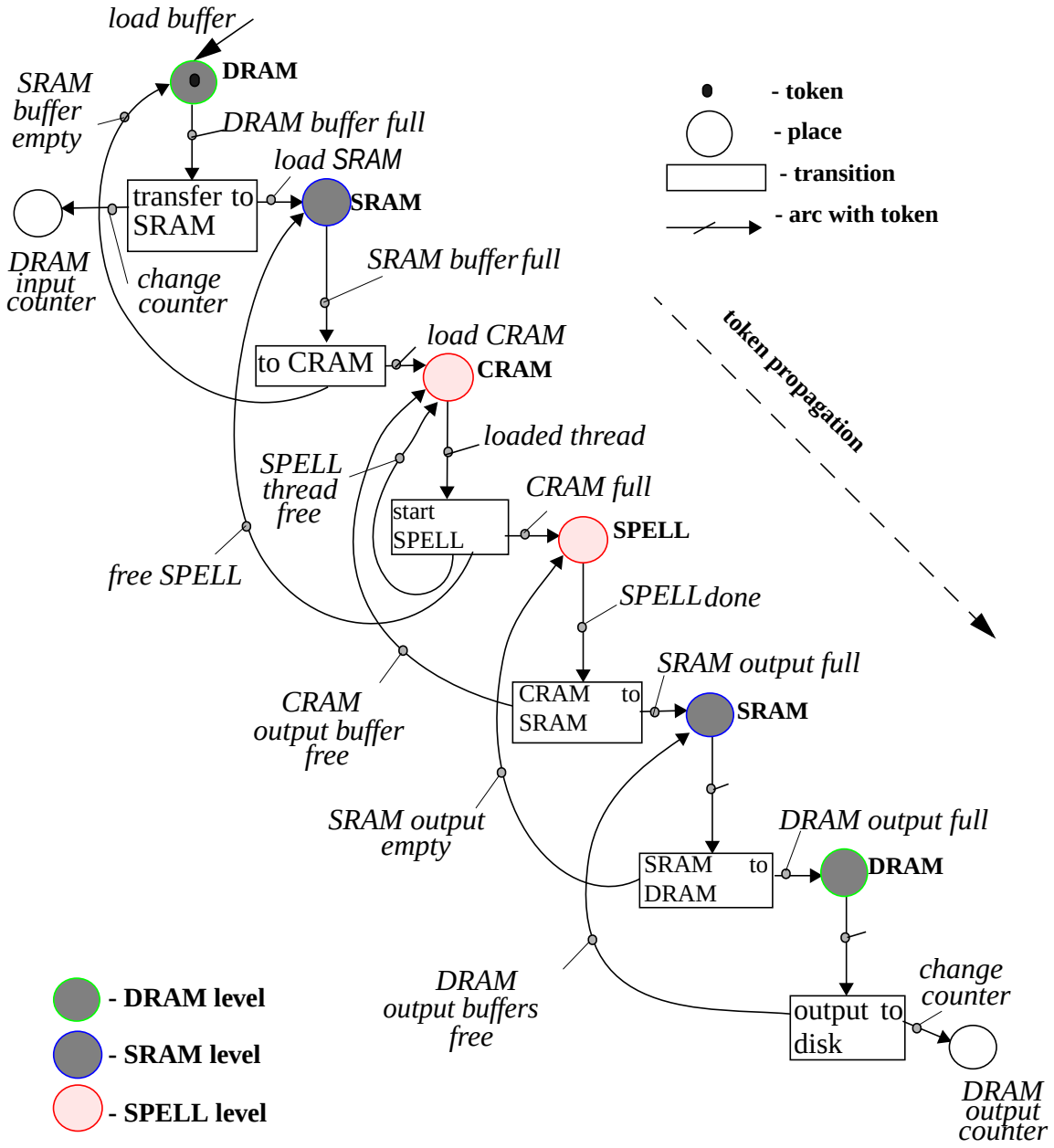


Figure 7.3 - Buffer relationship and data flow for single node HTMT

exclusion protection mechanism. In the first approach, we considered a central queue for each SPELL. In this case each thread checks the queue when it is free from execution. The problem with this approach is the "hot-spot" at a queue that will slow down the context availability for running threads and decrease the overall performance of the SPELL. A second approach requires a number of queues (one per thread) where the SRAM PIMs look for an empty queue to write the next context. The problem with this approach is the additional engine needed to perform the search for an empty queue. The third approach considers one task queue per SRAM. The running SPELL thread, searching for a new context, scans the queues of tasks. This approach requires an additional mechanism that will help threads to find ready to execute contexts (such as the algorithm for SPELL threads of finding ready to execute context at the closest SRAM position) or will assign threads to task queues preventing them from being idle. In our prototype we have chosen the last approach, but did not take in consideration finding the closest SRAM to the SPELL.

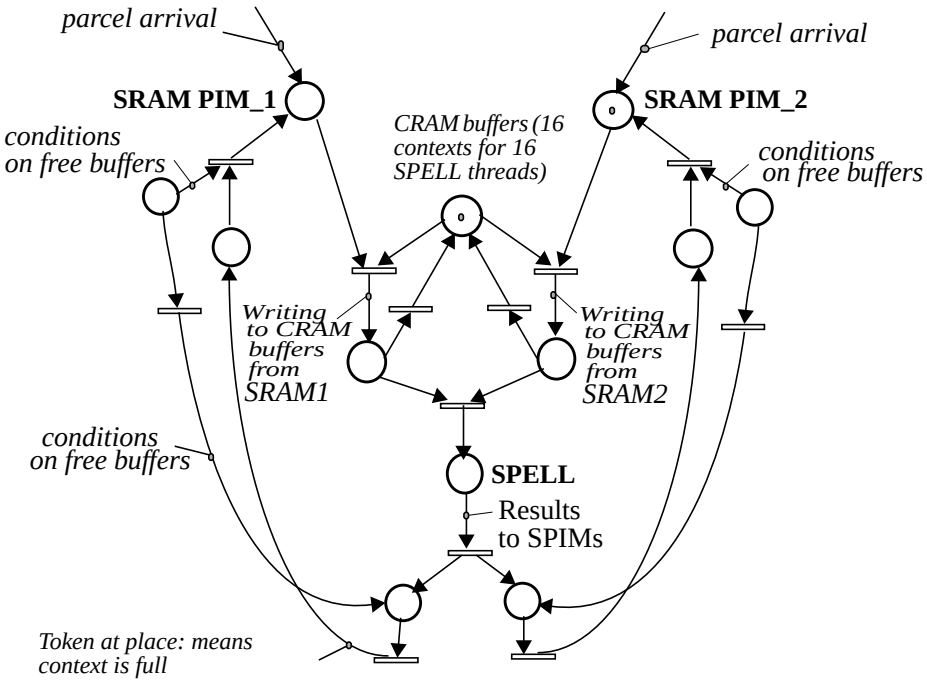


Figure 7.4 - The mutual exclusion solution for SPIMs

7.4.2 SPELL level

In a SPELL (Figure 7.5), a token representing a start signal from an SRAM PIM, arrives into a place that fires a transition which activates multiple threads in a SPELL. Up to 16 threads can be spawned simultaneously if SRAM buffers contain enough ready contexts. Each thread activates multiple strands that represent a second level of concurrency in the model which corresponds to the real system model. A token is associated with each thread and with each strand to show that these processes are always active. The current state of the system is defined at each transition, and at least one state will be reached on each transition.

7.4.3 DRAM PIM level

When a parcel arrives at a DRAM level from SRAM level to read or to write data from/to DRAM, it causes creation of new thread in the DPIM. This thread generates a new parcel, which accesses the meta-data DRAM to find the address of the specific DRAM where the requested data is located (objects in HTMT can literally reside on thousands of chips, and the mapping of sub-datum to chip may change dynamically). Then, depending on the parcel context, the accessed data is sent back to the original destination (to SRAM level) or it is written to some DRAM causing changes in the meta-data tables. This finishes the thread's action. If parcels request data that is not present at the meta-data DRAM at this cluster of DRAMs (set of DRAMs that are grouped in hardware and, logically, in software), a new parcel (and thread) will be created by the same thread as an additional request parcel.

Multiple threads represent multiple parcels that arrive at DPIMs to access data. Concurrent threads are created at the DRAM level on each parcel arrival. We associate one thread per parcel to provide the parallel search and data access to different DRAMs for independent active streams. This models concurrency at this level of memory hierarchy.

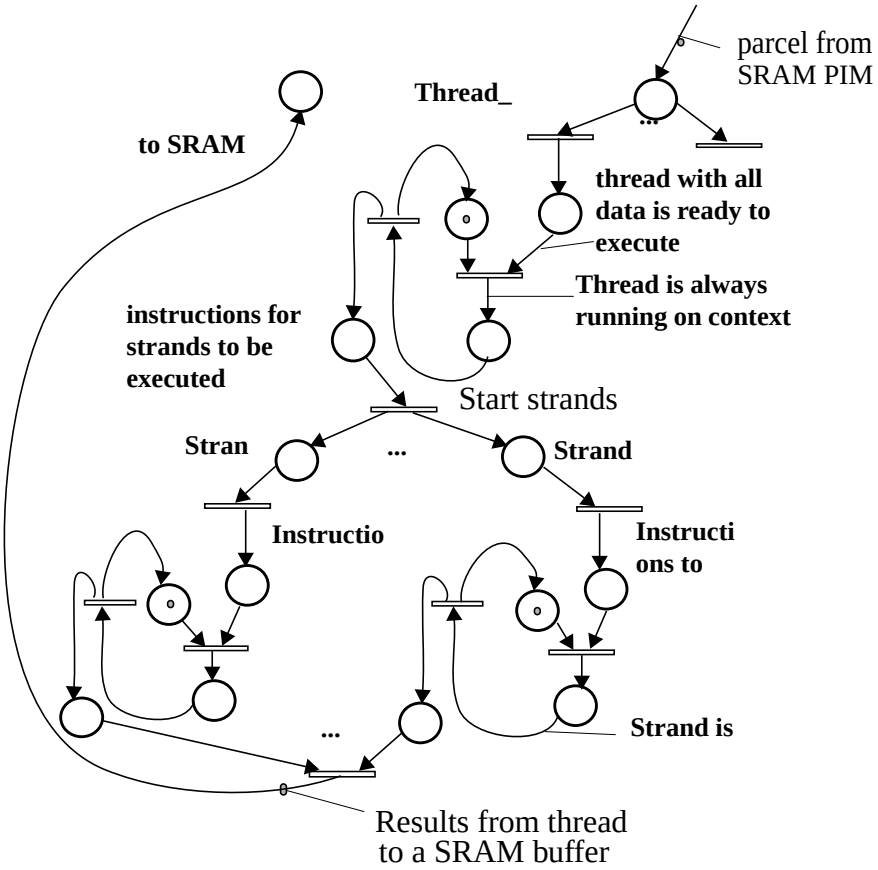


Figure 7.5 - SPELL threads and strands in the HTMT

7.5 Transformations

We created the HTMT prototype using a matrix multiplication example for two NxN matrices. Using our Petri Net model we simulated the DRAM PIM, SRAM PIM, and CRAM/SPELL functions.

We made several transformations from the created Petri net model to be able to program our prototype. Tokens were replaced by parcels, leaving the meaning of data migration but changing the token into a message. Transitions were substituted by function/method calls, and places by the HTMT memory states where data resides initially or after a parcel execution. A simplified parcel interface is designed to simulate the dynamics in the system data flow where action on parcel arrivals are implemented as library calls, and at different HTMT levels parcels are represented by Sockets or DatagramSockets. Arranging separate modules of the HTMT together, we demonstrated the complex structure of the HTMT execution model. The analysis of the Petri net model on non-determinism and block-free process at any state is beyond the scope of this work.

7.6 Conclusion

In this chapter we explained the model that was created to study the functional flow and the program execution model using Petri nets. It allowed us to provide the details for the HTMT prototype. We explained how to interpret the Petri net model and how to transform it in terms of the HTMT prototype implementation. The algorithm that implements the HTMT execution process in our prototype was built during this work and is presented in the next chapter.

CHAPTER 8

IMPLEMENTATION

8.1 Introduction

To program the HTMT prototype and its execution flow, we used a Petri net model to analyze the early HTMT prototype. As a result of this analysis, we present the model of HTMT prototype as a program where each block of the HTMT is represented by a separate Java object with its own input and output data and methods. This choice of the Java language was made to make the design simple and the implementation easy. It allowed us to integrate Java objects as abstractions of components of our system and to use multithreading and distributed execution which are available by default.

In this chapter we discuss details of the HTMT programming prototype implementation, and explain a parcel-driven design that was created using matrix multiplication as a test application for our prototype. Finally, we discuss results and gathered statistics from the studied example.

8.2 A program organization

In our prototype (Figure 8.1), the DRAM, SRAM, and CRAM data objects consist of the data structures that represent the HTMT memory hierarchy. The DPIM, SPIM, and SPELL functional objects provide the methods that modify the states of each object. The CNet and Opt_Net objects represent the intercommunication process between SPIM/SPELL and DPIM/SPIM levels, respectively. There are other objects that represent more detailed functionality, such as InputQueue and OutputQueue, which represent the memory context switch, and the Parcel object which is used for different kinds of messages in the system.

8.3 DRAM level

The DRAM level design is presented on Figure 8.2. The application data is distributed across DRAMs, where the set of DRAMs in one DPIM is called a *DRAM cluster*. The data can be stored in more than one DRAM cluster. To find a piece of data, one needs to specify the cluster number, the DRAM number, and the address within the DRAM. Because data is addressed by parcel tags, to find the data in DRAMs it is necessary to translate these parcel tags into the physical addresses. This translation table is defined per application and is called a *meta-data table*. Each DPIM has its own table which contains translation for tags of local closures, and the address of the DPIM which has the *complete meta-data table*. If the tag's data can not be put in one DRAM entirely, we need a flag in the meta-data table to indicate that the rest of the data for this tag is located somewhere else. This flag is called "Complete_read" and is set if all data for this tag is local.

When a parcel arrives at a DRAM level from SRAM level to read or to write data from or to DRAM, it creates a new thread in the DRAM PIM space (“Accept parcel” method). This thread generates a new parcel and, using meta-data in specified DRAM, accesses requested data in DRAMs. Then, depending on the parcel context, these data are sent back to the original destination (to SRAM level) or written to some DRAM, causing some changes in the meta-data tables. This finishes the thread’s action. If a parcel requests data that is not present at the meta-data DRAM at this cluster of DRAMs, a new parcel will be created (“New_parcel on miss”). A new thread will be created for the generated parcel, and will access meta-data in different DRAM clusters. The results will be packed in the parcel form by DPIM data server and will be send back to the client (“Send_parcel”). The pseudo-code for DRAM level functionality is presented below (Figure 8.3).

In our implementation Vector data containers are used to represent physically distributed data structures (one Vector per DRAM). Distributed shared-memory is represented by accessing different Vector structures in the same DPIM_level object, where Vector’s name represents a DRAM address, a name of component in the Vector structure gives an address in DRAM, and the index of named component simulates data address.

Methods of the DPIM_level class represent the activities on DRAM data structures, such as initialization of DRAM, acceptance of a parcel, finding the data in DRAMs, creation of new parcels (if necessary) during data search and data movement at the DRAM level, sending parcels with results back to the clients at SRAM level, etc. All parcels at DRAM level go through the meta-data DRAM or a set of the meta-data DRAMs (in general) which contain the information about data in all DRAMs at this level.

The meta-data table in DPIM contains the information about the array names, indices that point to the named array row or columns, and the DRAM destinations, where the data resides physically. The

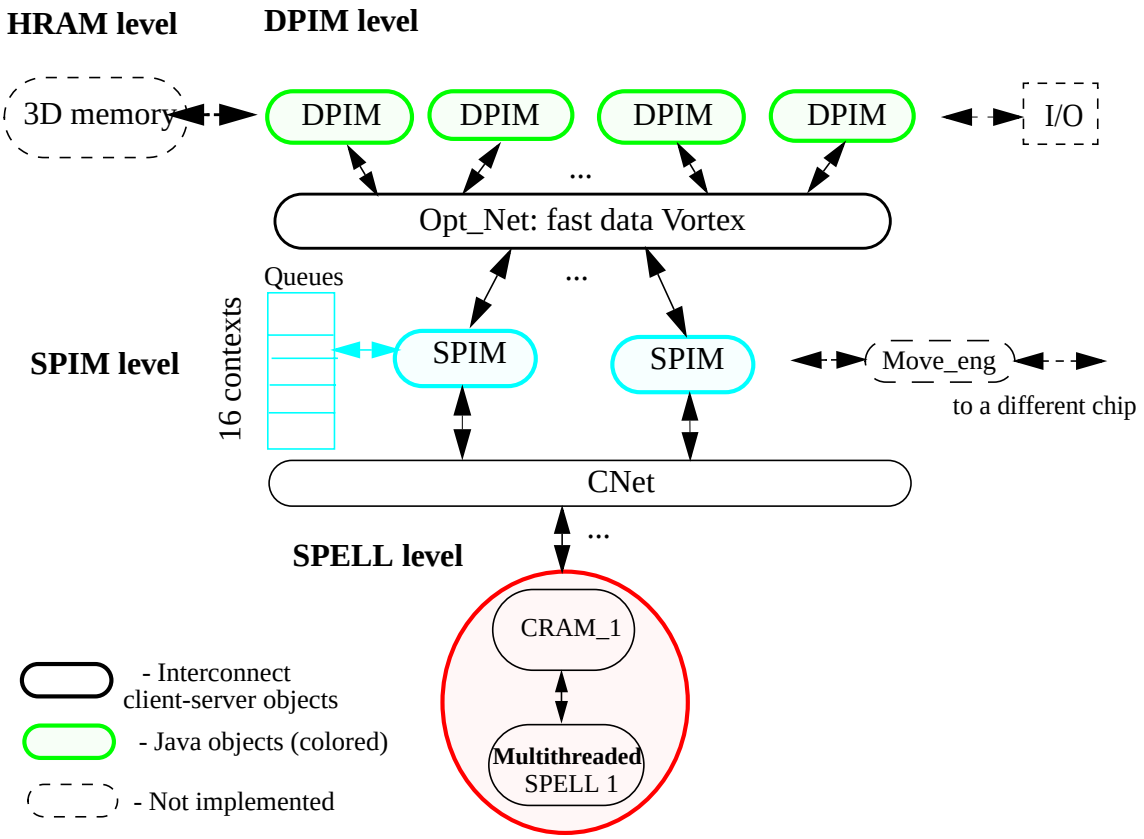


Figure 8.1 - Java objects in the HTMT prototype

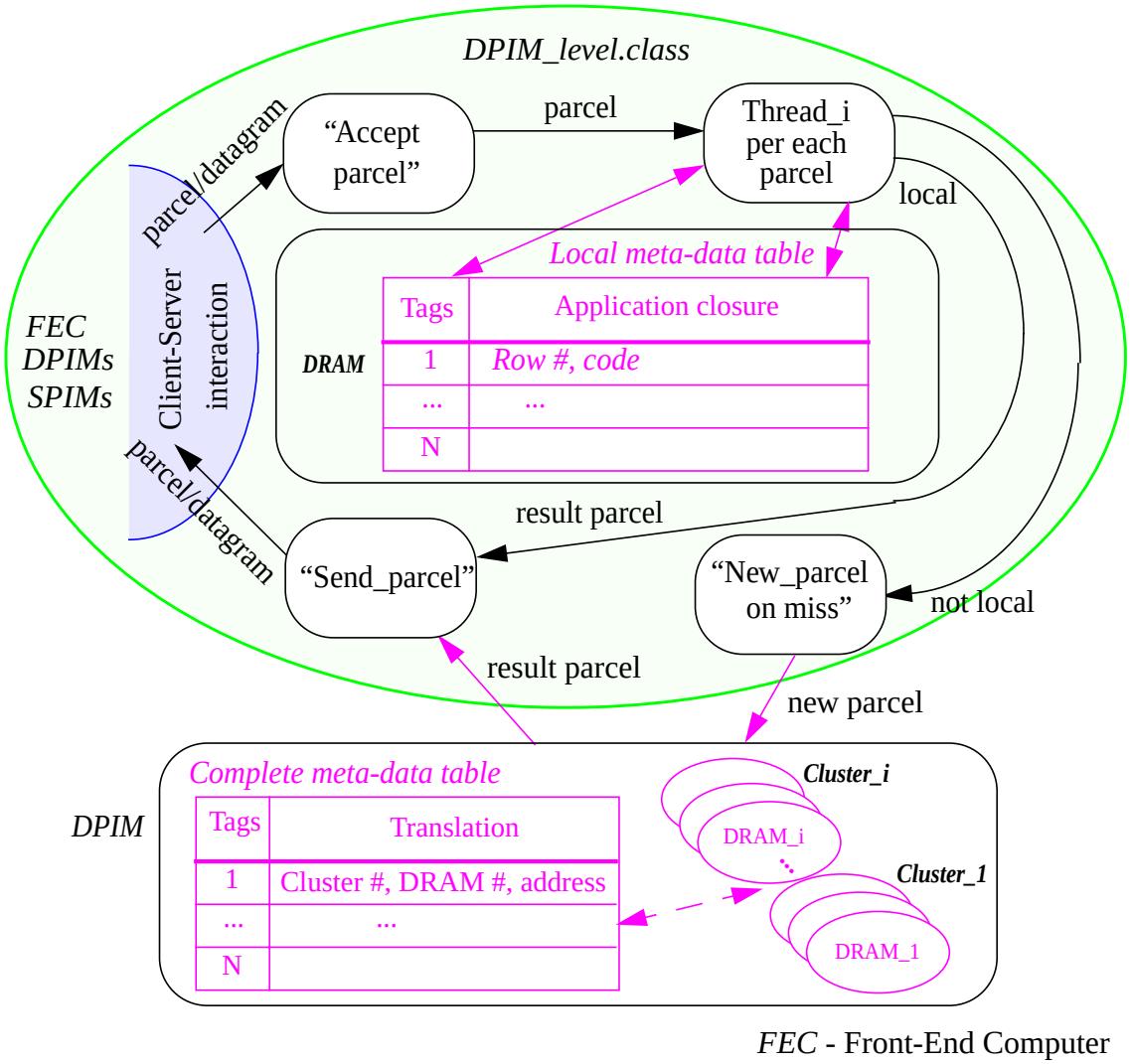
"regular" DRAMs include the following fields: the array name, row/column index, and the data for this row/column.

The DPIM level threads functions may include threads of control, threads that are involved in the inter-process communication, and threads that partition the data in DRAMs. The number of the last ones depends on data partitioning itself.

8.4 SRAM level

We assume that all transfers are done via parcels where parcel have a header with a sender name, receiver name, message type and size information, and when computation is performed, a parcel will be send back as the result of provided service.

When SPIMs are initialized (ready to accept parcels), several parcels with methods are loaded into their memories, including "Scatter" and "Gather" methods. These application level methods partition ("Scatter") the closures data and attached code from incoming parcels and place them into the SPIM's ready to execute input queue. The size of the data and code chunks depends on the number of SPELLs, the SRAM/



FEC - Front-End Computer

Figure 8.2 - DRAM level process implementation

```
Parcel_counter = 0;   // Parcel counter is a new parcel identifier.

Object_"DPIM"_level()   {

   Thread_"Listen_Connection"(String)   {
      Accept the datagram as a string with parameters;
      Recognize fields and prepare parameters to start a thread;
      Starting_a_thread(parameters);
}

   Statistics_"Parcel_arrival"(Parcel_number, time)   {
      Allocate a record on each parcel in the file;
      Set the parcel arrival real time;
}

   Starting_a_thread(parameters)   {
      Create and start a thread on each parcel arrival;
      Parcel_counter++;
      Statistics_"Parcel_arrival"(Parcel_number, time);
      run Thread_"On_Parcel"();
      Statistics_"Parcel_completes"(Parcel_number, time);
}

   Thread_"On_Parcel"()   {
      Find DRAM index in the Meta-data table;
      Find data in indexed DRAM;
      Get requested data from DRAM;
      Check if data more than in one DRAM; //distributed data
      if (data not in one DRAM)
         Check the Complete_read status label; //Complete_read
         if (Complete_read)
            Data_Incomplete(parameters);
      else
         Write results into parcel form;
         run "Write_Connection"();
   }

   Data_Incomplete(parameters)   {
      Fill fields and prepare parameters for new parcel;
      Create and start a new thread on each "data incomplete" in DRAM;
      Starting_a_thread(parameters);
}

   Statistics_"Parcel_completes"(Parcel_number, time)   {
      Search a record on parcel number in the statistics file;
      Set the parcel real completion time;
}

   Thread_"Writer_Connection"(Parameters)   {
      Create a new parcel with results;
      Create a datagram as a string to send results back to the sender/
client;
     Statistics_"Parcel_completes"(Parcel_number, time);
}
}
```

Figure 8.3 - Pseudo-code that implements DRAM level

CRAM sizes, the number of threads that we want to use/activate, and the level of concurrency in the application (the number of strands per a thread and the number of operations and data elements per thread and strand). Newly formed closures from SPIM's input queues will be loaded to CRAM each time a SPELL's thread completes the computation, frees the buffers in CRAM, and "signals" that it is done to SPIM via a "control" parcel. The SPIM multiple concurrent threads are associated with parcels that arrive to SPIM, one per parcel. Additional threads can be created during the context exchange if a parcel is sent to DPIM level. The diagram for the SPIM level is shown in Figure 8.4.

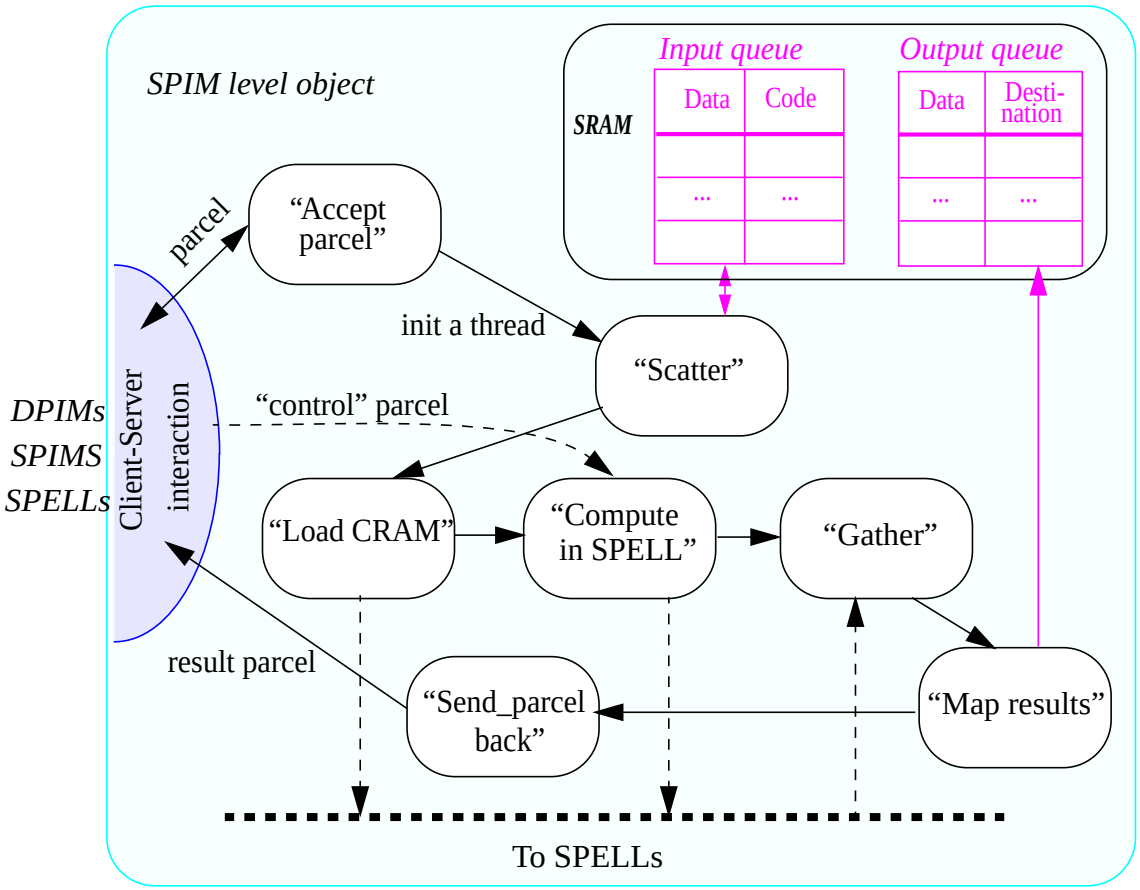


Figure 8.4 - SRAM level process implementation

SPIM is implemented as an object with methods which will accept the parcel ("Accept parcel"), scatter the information from them ("Scatter"), initiate the data and code transfers to CRAM memories ("Load CRAM"), initiate the work of SPELLs ("Compute in SPELL"), manage the transfers of the results from CRAMs to the SRAM's output queue, collect the data and perform additional operation, if needed ("Gather"), and send results to the DRAM memories ("Map results") using the destination address that was provided in the originally sent parcel. When a task computation is finished, SPIM sends a "completion" parcel to DPIM.

The input and output queues are implemented as FIFO queues and Vector containers ("Queues"). Each element of this Vector contains the data and the starting address of the pre-loaded by SPIM (by a parcel) code that must be performed on these data.

8.5 SPELL level

The context allocation and execution provides the dynamics in SPELLs. The context arrival at this level is always associated with thread. The SPELL level can contain at most 128 concurrent streams at a time which include 16 threads that can spawn at most 8 strands each. All SPELL threads share a frame of contexts, but each thread is associated with one context (simulating the hardware thread resource hold on register context once the execution is started), and contains its local variables for passing data between its own group of strands. Threads also contain the intermediate data during execution. The strands are the deepest level of the HTMT threading, and represented as the sequences of independent instructions that will be performed entirely once strand is enabled. Strands can share the context data with other strands within one thread, but can not invoke any other threads. The block diagram of SPELL level prototype is presented in Figure 8.5.

In our prototype we implemented the CRAM memory as an array. The “Accept parcel” method distinguishes between parcels that arrive from SPIM, allocates space for data and code in memory, and creates new threads. The “Compute thread” method creates up to 8 strands which are controlled by the application code that arrives in the parcel, and sends a “control” parcel to SPIM when computation is finished and new data is written to CRAM.

8.6 Inter-process communication

In our prototype, inter-process communication between clients and servers is implemented via Opt_Net and CNet objects that represent the optical network and communication network, respectively. They include methods that manage other objects (“Server” and “Client”).

The pseudo-code for “Client” object is presented in Figure 8.6 where several threads are created to listen to “Server” for a connection and to request service through a known port.

Object “Server” creates the thread that will listen on the port for client’s requests to perform some task. We include detailed pseudo-code in Figure 8.7 which explains the multi-threaded Server functions. The execution of this code starts when client signs for the service and listener-thread accepts the client’s request (a parcel). Several parcels can be sent to this port where they will be processed by the “Server” object in the first-come first-serve order. In the next section we discuss different kinds of parcels used in the prototype.

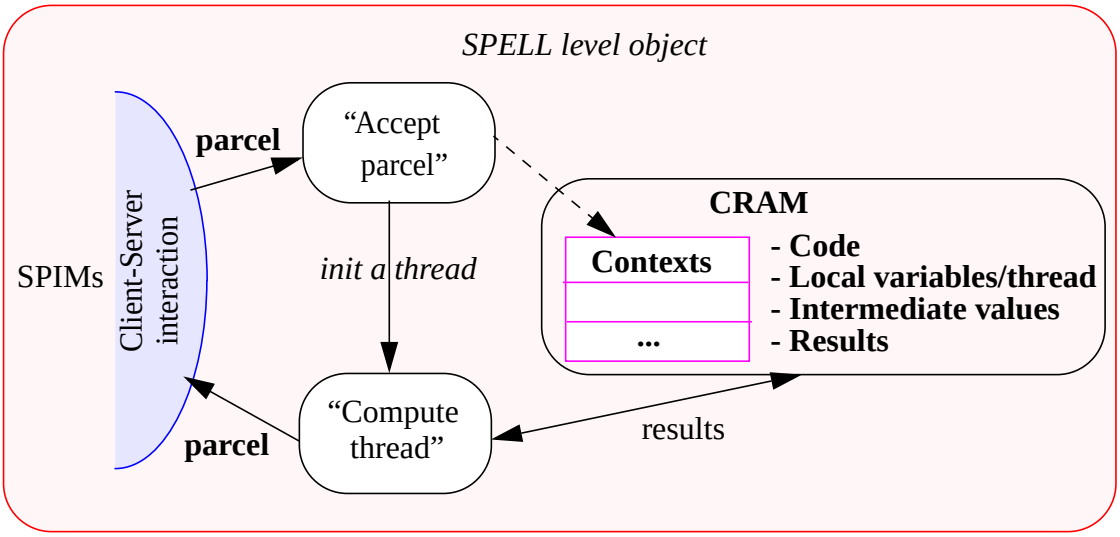


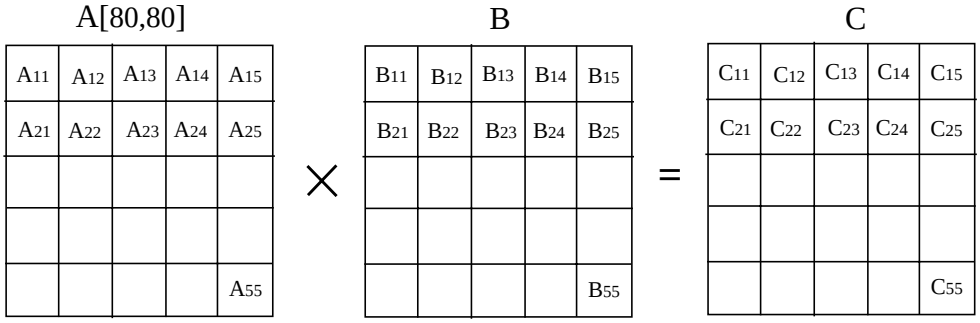
Figure 8.5 - SPELL level process implementation

8.7 Parcel types

Parcels are messages in the system that initiate work and allow the computation process. We distinguish between different types of parcels by their contexts. We assume that each parcel contains control information. The parcels that contain only control data are *control parcels* and are used as signals to different processors to perform some action or to synchronize the computation (such as an update signal, or request to check the point in the computation). If those parcels also contain code, they are *method parcels*, and are used to distribute code across the system. If parcels contain control information and data elements, they are *data parcels* used to distribute data. The parcels are called *context parcels* if they contain control information, data and code that must be performed on these data. The parcels will be described in more detail when we discuss matrix multiplication example.

8.8 Matrix multiplication example

Let us consider the multiplication of two matrices A and B of sizes 80x80 each. Initially, the matrices reside in the DPIM memory where they are partitioned into blocks. A_{ik} and B_{kj} represent blocks of the size 16x16 (Figure 8.8). DPIM sends these blocks as new context parcels to SPIM's input queue.



$C_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} + A_{14}B_{41} + A_{15}B_{51}$

Figure 8.8 - Partitioning data in DPIMs

```
Object_"Client"_Proc(host, port_num) {
   Create thread "Reader";
   Create thread "Writer";
   Set "Reader" higher priority then "Writer";
   Start two threads.
}

Object_"Reader"() {
   Read data from "Server";
   Do some calculations;
}

Object_"Writer"(data_to_send) {
   Create "In", "Out" streams;
   Send data to the "Server".
}
```

Figure 8.6 - Pseudo-code for Client

```
Object_"Server"_Proc(port_num)   {
   Create server thread with name "Server";
   Create a Server Socket to listen for connections on: "listen-socket";
   Create the ThreadGroup for connections;
   Initialize a vector to store connections in;
   Create additional "Vulture" thread to wait for other threads to die;
   Start to listen for connections.
}

Thread_"Server"()   {
   Listen for connections from clients:
   Socket client.socket=listen_socket.accept() returns with a normal
   socket;
   Initialize the streams and start Thread_"Connection";
   Prevent simultaneous access (will synchronize the connections adding
   them to the list of connections).
}

Thread_"Connection"(Socket client_socket, ThreadGroup, priority,
   Vulture)   {
   Give to "connection"_thread a name, a ThreadGroup and priority;
   Create "In" and "Out" streams for connection;
   Start thread to provide the "Service";
   Notify the "Vulture"_thread when the connection is dropped.
}

Object_"Service"_Proc()   {
   while(all data is sent) {
      if (task is done) {
         Connection is closed;
         Record is removed from the list;
         "Vulture"_thread is "Notified".
   }
       Send data to the client and wait for getting data back;
       Work with received data.
   }
}

"Vulture"_thread() {
   while("connection"_thread is alive) {
      Wait for notification of exiting (dying) thread, checking every 5
      sec.;
      Prevent simultaneous access to the server from adding a new
      connection while removing old one;
   }
   Remove exiting thread from the vector of connections (and list).
}

Object_"Notifying_Vulture"() {
   Synchronize the "Vulture": lock it;
   Notify "Vulture" object with call "notify".
}
```

Figure 8.7 - Pseudo-code for Server

Each block of the matrix C is computed in SPIM as shown in Figure 8.9. In this code the intermediate matrix P is a product of two blocks. Each element of matrix P is a product of a row of the block of matrix A and a column of the block of matrix B, and is computed in SPELLs.

```
C[1,1] = 0;
for (i=1; i<=n; i++) {
        P = A[1,i] * B[i,1];
     C[1,1] = C[1,1]+P;
}
```

Figure 8.9 - Computing a block of matrix C in SPIM

The SPIM data partitioning is shown for two SPELLs in Figure 8.10, where upper half of the block of matrix A is sent to SPELL 1, and the other half to SPELL 2. The block of matrix B is sent to both

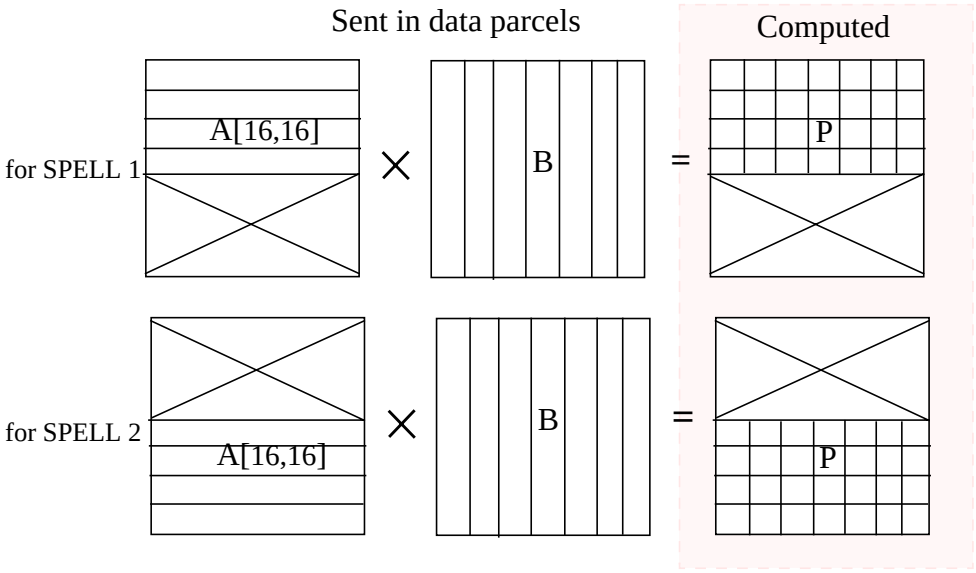


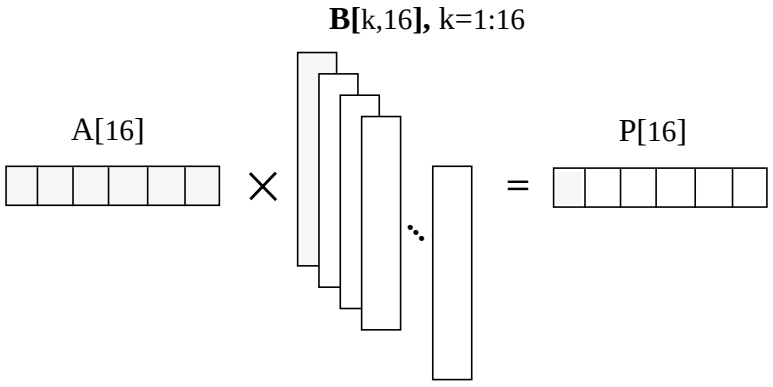
Figure 8.10 - SPIM data partitioning for SPELLs

SPELLs. SPIM sends code parcels to both SPELLs, and then distributes the data via data or context parcels across CRAMs in both SPELLs.

When SPIM loads contexts to each SPELL memory, the code in SPELL assigns one context per thread. Each context consists of a row of the block of matrix A and a column of the block of matrix B. Each thread creates 8 strands each of which performs two multiplications. When a thread finishes computation on the data from one context, it writes the computed element of P to allocated space in CRAM, and starts computation on the next context (Figure 8.11).

The resulting block P is produced by combining results from SPELL1 and SPELL2. It is stored in CRAM until all rows of block of matrix A and all columns of block of matrix B are multiplied. When it is ready, the control parcel is sent to SPIM, and SPIM can load those values into SRAMs.

SPIM accumulates each computed P to produce the block of matrix C_{ij}. When C_{ij} is computed, it is sent to DPIM where individual blocks of matrix C are combined to produce the final result. Finally, the DPIM updates meta-data tables.



16 threads x 8 strands x 2 multiplies x 2 elements
= minimum of 512 CRAM data reads in SPELL.

Figure 8.11 - Data partitioning for 16 SPELL's threads

In this scheme, once SPIM completes its transfer it does not need to wait for others to complete. Data is distributed without any dependencies. The matrix B is the same for all SPELLs, and rows of the blocks of matrix A do not correlate in transfers. So once one SPIM completes transfers, it fills the context with new rows and columns of the blocks, and starts threads. It will help with thread management, decrease the length of communication links, and simplify the element indexing.

8.9 The parcel-driven program flow

Previous section provided details on the data flow in the prototype model for the matrix multiplication example, but did not emphasize the functional model. It is discussed in this section and is presented in Figure 8.12.

The execution process starts at the front-end computer (FEC), which initiates PIMs, allocates memory for new data objects, and starts DPIMs by sending them *control parcels* (arc 0). DPIMs load the application code (1) and data (2) to their memories. The code includes methods to transfer data to PIMs and SPELLs, to control the PIM and SPELL levels, and to perform the computation. The data consists of matrices A and B.

When loading (3) is complete, DPIMs send control parcels to SPIMs to initialize and start their work (4), and load *code parcels* to SRAMs (5) which include methods to perform block matrix multiplications. Next, SPIMs request matrix blocks for A and B matrices from DPIMs, sending control parcels. When *data parcels* from DPIMs arrive (6), SPIMs partition the data (7) and send *code* (9) and *data* (10) *parcels* to SPELLs (8). These parcels contain rows and columns of the blocks of matrices A and B, respectively. When data is loaded, SPIMs send *context parcels* to SPELLs (11) which contain references to data and methods that must be performed on these data. When SPELLs perform computation (12), they send control parcels to SPIM (13, 14). SPIM reads ready block to its memory, accumulating the data. When a block of the matrix C is computed, SPIM sends it in a data parcel to DPIM (15). When DPIMs collect all the blocks of matrix C, the task is complete (16). Results, then, are sent to FEC (17), and new task can be loaded (18).

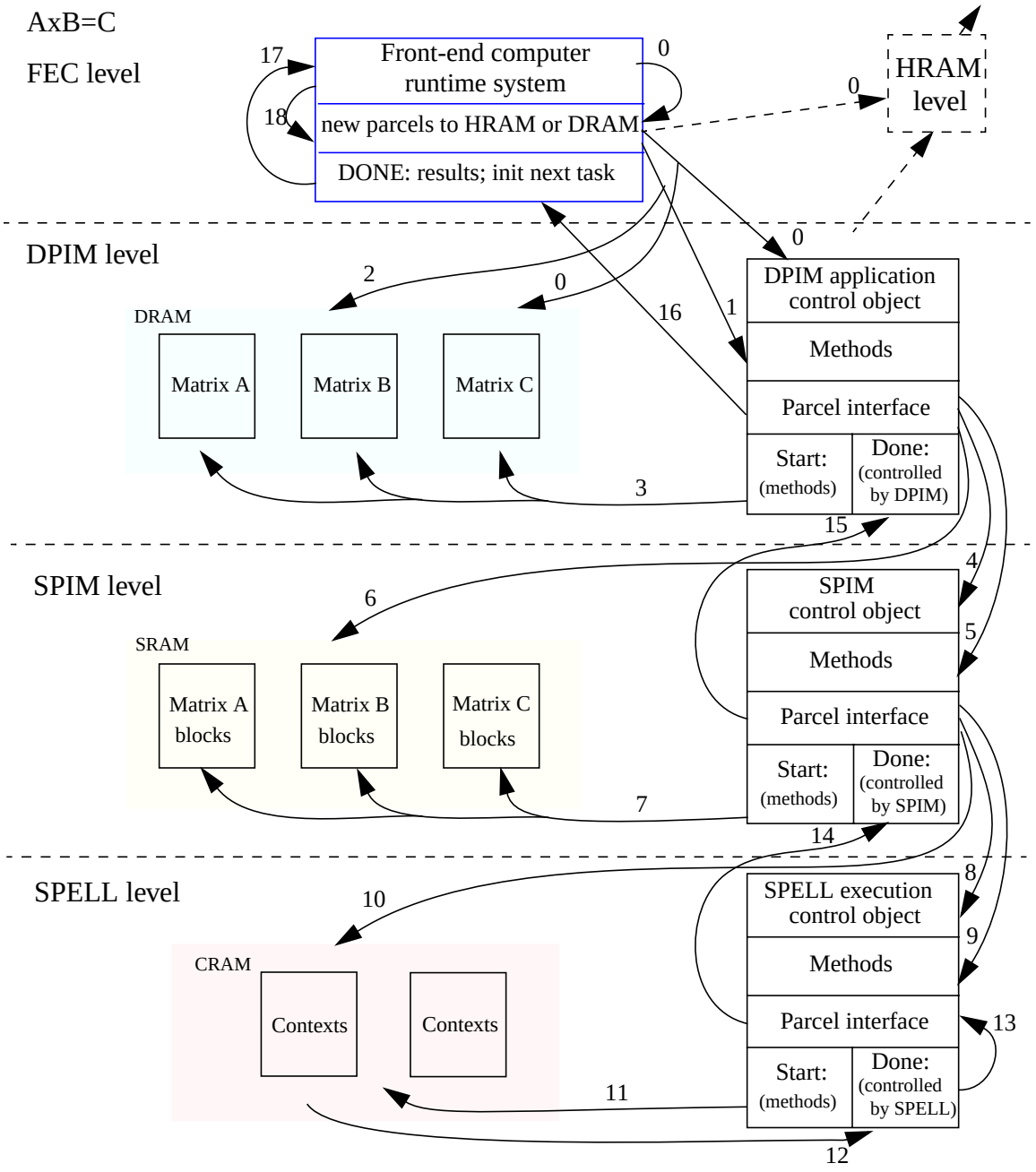


Figure 8.12 - The HTMT parcel-driven flow

8.9.1 Parcels in the prototype

The parcel library includes the description of all possible messages in the functional flow of our HTMT prototype. It is placed in the public class parcel_lib and includes definitions for all parcels. The parcels in the library are organized by their type and include the information about their structures, functionality, and the relation to different levels of the memory hierarchy in the HTMT prototype.

Control parcels are the following: *initiate()* - initializes PIMs and SPELLs; *allocate_data()* - allocates a place for data of specified size and type; *allocate_code()* - allocates memory for method of specified size and name; *read()* - requests data from PIMs; *perform_computation()* - starts the execution of the specified method in the specified object; *compilation_complete()* - informs that the task is computed.

Data parcels in our example is represented by only one parcel: *write()*, which writes the data to the specified by name data object.

Code parcels can be of two kinds: *load_method()* writes the method code to the location, specified by the method name; *load_methods()* does the same for more than one method at a time.

Context parcels in our example is represented by a *load_context()* parcel that contains references on the method and its arguments. These references are written to the specified context.

8.9.2 Methods

In the HTMT prototype model the application program is separated into multiple application pieces (objects) that contain methods to be invoked on the different processing levels with different data sets.

The DPIM methods are generated from the application and loaded and triggered by parcels from either FEC, SPIM, or DPIM application control object. These methods are assigned to partition the data for SPIMs and to gather the results from SPIMs.

The SPIM methods contain code that run in SPIMs to perform data partitioning and data scattering across CRAMs, to manage and control SPELL execution, and to gather the results of the SPELL computations. These methods can be triggered by DPIM parcels, or by control parcels from SPELLs. The SPIM methods are shown in Figure 8.13 where MUL_matrix() calculates the matrix C, MUL_block() compute the product P, and ADD_block() accumulates the block of C. Row() and Column() methods perform data partitioning for SPELLs. MUL_row() is a SPELL method which multiplies a row of the block by a column.

The SPELL methods contain code that run by SPELL from its local CRAM to execute one or more threads of application computation which take data out of a context in CRAM, process it, and return results, usually back to CRAM.

8.10 Statistics

The parcel traffic for the early HTMT model prototype was studied on matrix multiplication of two matrices of size 80x80. 143 MHz SPARC stations with 256 MB of operating memory were used to simulate the HTMT prototype model using the matrix multiplication example. Each parcel in our prototype contains a header and a body of variable length. The header size is 32 bytes, and it includes name of destination, return address, and information about a content (parcel type). The body can consist of data, methods, control information, and references to data or code objects. We collected statistics on parcel traffic, including parcel types, sizes of parcels, and the number of parcels between different levels of HTMT.

For DPIM-SPIM level interactions (Table 8.1), the body size of the data parcels is the size of data blocks that are sent from DPIM to SPIM (16 by 16 elements 8 bytes each). The total number of parcels from DPIM level to SPIMs is equal to the number of data blocks of matrices A and B which must be sent from DPIM. The body size of the code parcels is the size of the methods which are performed at SPIM and SPELLs (Table 8.2). The body of control parcels contains name of one object (32B), and there are 64 such

parcels (50 to allocate the space for 50 data blocks of A and B, to allocate the space for codes in each SPELL (12), and 2 to start computation).

```
static Matrix MUL_matrix(Matrix A, Matrix B) {
  Matrix C = new Matrix();
  for(int i = 0; i < 5; ++i)
      for(int j = 0; j < 5; ++j) {
         double c[][] = new double[16][16];
         for(int k = 0; k < 5; ++k) {
            double a[][] = A.block(i,k);
            double b[][] = B.block(k,j);
            double p[][] = MUL_block(a,b);
            c = ADD_block(c,p);
         }
         C.set_block(i,j,c);
       }
   return C;}

static double[][] MUL_block(double a[][], double b[][]) {
  double p[][] = new double[16][16];
  for(int i = 0; i < 16; ++i)
      for(int j = 0; j < 16; ++j) {
        double ra[] = row(i,a);
        double cb[] = column(j,b);
        p[i][j] = MUL_row(ra,cb);
     }
  return p;}

static double[][] ADD_block(double a[][], double b[][]) {
  double p[][] = new double[16][16];
  for(int i = 0; i < 16; ++i)
      for(int j = 0; j < 16; ++j)
         p[i][j] = a[i][j] + b[i][j];
  return p;}

static double[] row(int i, double[][] a) {
  double r[] = new double[16];
  for(int j = 0; j < 16; ++j)
      r[j] = a[i][j];
  return r;}

static double[] column(int j, double[][] b) {
  double c[] = new double[16];
  for(int i = 0; i < 16; ++i)
      c[i] = b[i][j];
  return c;}

static double MUL_row(double r[], double c[]) {
  double s = 0;
  for(int i = 0; i < 16; ++i) s += r[i] * c[i];
  return s;}
```

Figure 8.13 - SPIM and SPELL methods

Table 8.1 - Parcel traffic from DPIM to SPIM

Parcel types	Header size, (bytes)	Body size (bytes)	Number of parcels	Total (bytes)
Data parcels	32	2048	50	104000
Code parcels	32	322(total)	6	514
Control parcels	32	32	64	4096

Table 8.2 - Methods sizes

Method name	Size (bytes)	Level
MUL_matrix	108	SPIM
MUL_block	68	SPIM
ADD_block	59	SPIM
Row	28	SPIM
Column	28	SPIM
MUL_row	31	SPELL

To request data from DPIMs, SPIMs send 50 control parcels to get blocks of matrices A and B, and send 25 data parcels to DPIM when all blocks for matrix C are computed (Table 8.3). Each such data parcel contains a matrix block of size 16x16 (8B each element).

Table 8.3 - Parcel traffic from SPIM to DPIM

Parcel types	Header size, (bytes)	Body size (bytes)	Number of parcels	Total (bytes)
Data parcels	32	2048	25	52000
Control parcels	32	32	50	3200

The parcel traffic from SPIM level to SPELLs consists of parcels of all four types (Table 8.4). In our case, all columns of blocks of matrix B are sent to both SPELLs while only half of rows of blocks of matrix A are sent to each SPELL. To calculate each block of matrix C, it is necessary to send 5 blocks of matrices A and B. The total number of blocks is 25. To perform this computation, 16 columns of blocks of matrix B are sent to each SPELL and rows of blocks of matrix A are divided by the number of SPELLs which makes the total number of data parcels equals 2000+2000*N$_{SPELLs}$ (in our case N=2). The context parcels are sent to multiply all transferred to SPELLs elements of matrices A and B (25*5*16*16), and specify the name of objects that must be involved in computation by referencing them by their names (method name, 2 operands, result). Control parcels are used here to allocate space for data from each data parcel (6000), to allocate the code (1 for each SPELL), and to signal to start the computation (2).

Control parcels from SPELLs to SPIMs (Table 8.5) are sent to inform SPIMs that computation of P, the products of A and B, is done (125 *Pi*-s). The partial results P*i* are read from SPELL to SRAM through DMA trasfers (250K bytes).

Table 8.4 - Parcel traffic from SPIMs to SPELLs

Parcel types	Header size, (Bytes)	Body size	Number of parcels	Total (bytes)
Data parcels	32	128	6000	960000
Code parcels	32	10	1	42
Context parcels	32	128	32000	5120000
Control parcels	32	128 (max)	6004	960640

Table 8.5 - Parcel traffic from SPELLs to SPIMs

Parcel types	Header size, (bytes)	Body size (bytes)	Number of parcels	Total (bytes)
Control parcels	32	0	125	4000

Based on these data, we calculated the total size of messages for each type to show the parcel traffic between the HTMT levels, and plotted it as a diagram (Figure 8.14). The DMA trasfers are not included in this figure. The darker area shows the total size of data parcels, and the lighter area represents context parcels. As expected, traffic between SPIMs and SPELLs is much higher than between DPIMs and SPIMs. This is analogous to conventional superscalar processor, where register traffic is much higher than memory traffic.

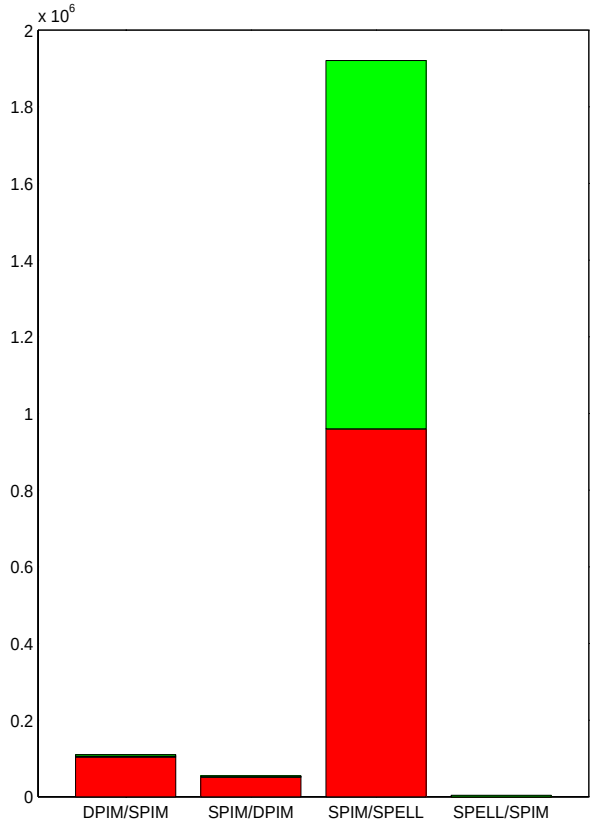


Figure 8.14 - Traffic between the HTMT levels (in bytes)

Next, we consider a more general case, when matrix sizes are very large, and estimate the number of operations that can be assigned to SPELL strands. The matrices are divide into blocks and partitioned so they can fit 1MB of CRAM memory.

In case of one SPELL, let us assume that in CRAM 1KB will be allocated for code and 2KB will be allocated for context parcels (16 contexts with 128 bytes of data) of total 3KB. To allocate 3 blocks of matrices A, B, and P in CRAM of $8n^2$ bytes each (8 bytes per element) requires $24n^2$ of CRAM memory. Then, the maximum size of the block of each matrix that can be allocated in CRAM is $n = \lfloor \sqrt{(1MB - 3KB)/(24B)} \rfloor$. It means that 208 is the maximum size of blocks of matrices A and B that can be loaded to CRAM at a time, and 208 multiplications will be performed by each thread, or 26 multiplications per strand (plus additions).

In case of N SPELLs, 3K of CRAM will be allocated for code and context parcel data, the whole block of matrix B, and $1/N$ block portions (the whole rows) of both B and P, where N is the number of SPELLs. Then, the maximum size of blocks that can be loaded to CRAM will be $n = \lfloor \sqrt{(1MB - 3KB)/((1 + 2/N) \times 8)} \rfloor$. Because (in our case) at least one complete row of block A must be sent to a SPELL to compute a row of block P, the number of SPELLs can not be more then the number of rows in a block (n). This means that the following equation must be satisfied: $n^2 = 130688/(1 + 2/n)$, and the maximum size of matrix blocks that can be loaded into CRAM is 630. Therefore, up to 630 SPELLs can be used for such work partitioning.

To estimate the optimal number of SPELLs for matrix multiplication of very large matrices, we assume that the maximum performance will be achieved when communication requires approximately the same time as computation: $\frac{Traffic}{bandwidth} \approx \frac{N_{mul} + N_{add}}{NP}$, where N_{mul} is the total number of multiplications, N_{add} the total number of additions, and P is performance of 1 SPELL in FLOPs.

To estimate the traffic from SPIMs to SPELLs, let assume that the size of the final matrix C is $M \times M$ bytes, and it is divided into $N_b = \frac{M}{n}$ blocks of size $n \times n$. To compute each block of matrix C requires N_b block multiplications, and the whole C requires N_b^3 block multiplications. We load $B_{kj} + A_{ik}/N$ elements from SPIM to each SPELL, and send back P_{ij}/N computed elements. For the block size n^2, the total traffic to and from SPELL will be $N_b^3 \times \left(n^2 + \frac{2n^2}{N}\right) N = \left(\frac{M}{n}\right)^3 (N + 2)n^2$, where $n = \sqrt{130688/(1 + 2/N)}$.

Using the HTMT characteristics in Chapter 2, we established that the optimum number of SPELLs for our task partitioning is estimated to be 18 which is shown in Figure 8.15.

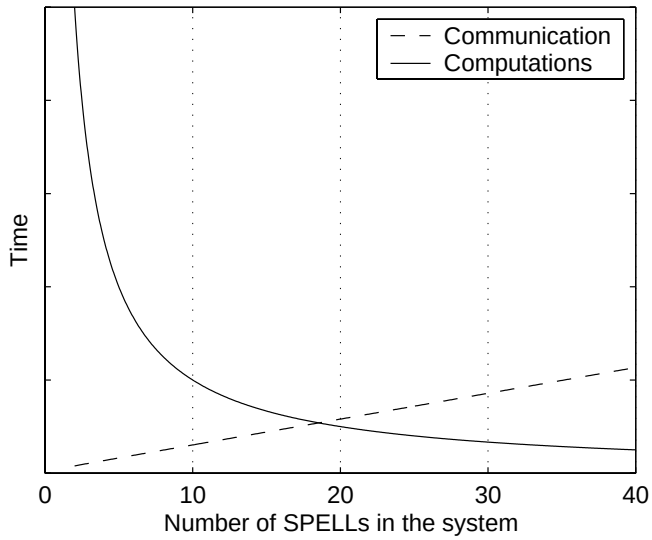


Figure 8.15 - The time spent on communication vs. the time spent on computation

8.11 Conclusion

In this chapter we discussed simulation of the early HTMT prototype which included design and implementation of prototypes for each HTMT subsystem and their main components. We also explained the inter-process communication, the parcel-driven interface, and methods that implement the computations. Finally, we discussed the results and provided statistics on the early HTMT prototype using matrix multiplication as an example for our simulations.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

In this thesis we discussed the HTMT computer architecture and developed a combination of existing parallel paradigms that neatly match this new design. We also identified the ways in which the object-oriented language Java can be used to prototype those paradigms. Then, we proposed new software models for the HTMT model, described their functions and sources of parallelism, mapped them to the HTMT architecture model, designed an early HTMT model prototype, and tested this prototype using a matrix multiplication example.

9.1 Challenges

The major difficulties and challenges in this project were to specify details of the HTMT execution model. Several possible concurrent programming models and paradigms that match the HTMT features were considered. A programming prototype for the HTMT execution model was suggested, designed, and implemented. The Petri net modeling technique was used to provide details on the HTMT start up operations, execution model, run-time functions, and on specifications for the HTMT parcel library. This prototype demonstrated the complex structure of the HTMT, proved that early HTMT prototype model can work on a matrix multiplication example, and showed that we can build distributed model which reflects real concurrency.

The prototype simulation using matrix multiplication allowed to collect first HTMT statistics which was considered in this work. Some additional information from this work can be used for early software development, including new programming languages and new compiler.

9.2 Conclusion

We created the HTMT execution model prototype for the new hybrid technology multithreaded architecture computer that introduces multiple memory hierarchies which are supported by a new kind of memory processing systems, PIMs. This highly concurrent HTMT architecture combines PIMs with new multi-threaded RSFQ processor technology that allows petaflop system performance. In this work we described the development of a Petri net model which allows demonstration of concurrency, multi-level memory hierarchy, deep data propagation in the system (from highly dense DRAM to highly fast processors), and data partitioning during execution process. Also, we used properties of the Petri Net model in design of the HTMT architecture to mirror the system functions. We showed that Petri net is a powerful tool that simplifies the problem of design decisions for very complex systems, and makes it possible to build a scalable system by “from simple problem example to the complex system structure” approach. Starting from the classic matrix multiplication problem, the architecture organization, and the execution process descriptions, we were able to design an early HTMT prototype and to understand the underlying principles of the resulting system.

9.3 Future work

Our future work in developing the HTMT model prototype will involve the consideration of new, more elaborate, models that will be found in the design process of the HTMT computing architecture model. We will add new algorithms to our set that will also fit to the suggested models, and will develop timing instrumentations for those algorithms to allow studies of performance statistics for our models.

The Petri Net model can be used later on to model the dependencies for increased number of SPELLs, and concurrency in the system. We plan on converting the synchronization transitions we saw necessary in the Petri Net models into hardware assisted functions in PIMs. More detailed and complete Petri Net simulations will provide guidance to expected performance characteristics. Also, the Petri net model notation can allow simple specification of complex concurrent operations that go on in the system. Using Petri net theory, the HTMT programming environment and new language grammar can be defined. The programming notation will allow the study new algorithms for HTMT system.

In the prototype, we will increase the number DPIM, SPIM and SPELL processors and add details on their functionality to model concurrency during parcel transfers between levels and to gather more accurate statistics.

The next step will be development of the multi-SPELL/PIM simulator on the cluster of workstations, to define and implement instrumentations. The goal of this work will be to define the runtime system and to break its functions away from the applications. Implementing the runtime system will allow us to built the prototype compiler tools to produce data movement code for the PIMs from applications. We also need to define the HTMT virtual memory and to specify details in the PIM addressing scheme. All this is the work of the next stage of the HTMT development, and must be completed by the year 2004.

BIBLIOGRAPHY

[1] "ASCI: Accelerated Strategic Computing Initiative," http://www.llnl.gov/asci asci_home_text.html, Dec. 1996.

[2] "ASCI Blue RFP Attachment5," http://www.llnl.gov/asci_rfp/cover.html, http://www.llnl.gov/asci_rfp/late.html, Apr. 1996.

[3] "Computing, Information, and Communications: Networked Computing for the 21st Century", Report by the Committee on Computing, Information, and Communications National Science and Technology Council, Supplement to the President's FY 1999 Budget, 1999.

[4] "Subcommittee on Basic Research of the U.S. House of Representatives Committee on Science: Testimony of K. Kennedy," http://www.house.gov/science, March 1999.

[5] P. Kogge, S. Bass, J. Brockman, D. Chen, E. Sha, "Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies," Frontiers '96, EEE Computer Society Press, CA,1996.

[6] G. Gao, K. Likharev, P. Messina, T. Sterling, "Hybrid Technology Multithreaded Architecture," 6th Sump. on Frontiers of Massively Parallel Computation, MD, pp. 98-105, Oct. 1996.

[7] P. Kogge, J. Brockman, T. Sterling, G. Gao, "Processing In Memory: Chip to Petaflops," IRAM Workshop, Int. Symp. on Computer Arch., CO, June 1997.

[8] K. Richard, V. Freeh, "The Modify-on-Access File System," ND CSE TR 98-12, March 1998.

[9] G. Gao, K. Theobald, A. Marquez, T. Strerling, "The HTMT Program Execution Model," CAPSL Technical Memo 09, July 1997.

[10] Birrel, A., B.J. Nelson, "Implementing Remote Procedure Calls," ACM Transactions on Computer Systems, vol. 2, no. 1, pp. 39-59, Feb. 1984.

[11] George S. Almasi, Allan Gottlieb, "Highly Parallel Computing," 2nd ed., pp. 253-255, 1994.

[12] P. Kogge, T. Giambra, H. Sasnowitz, "RTAIS: An Embedded Parallel Processor for Real-time Decision Aiding," 1995 NAECON, OH, March, 1995.

[13] J. E. Thornton, "Design of a Computer: The Control Data 6600," Scott, Foresman, and Co., 1970.

[14] "R10000 Microprocessor User's Manual - Version 2.0," MIPS Technologies Inc., v.2.0, Dec. 1996.

[15] G. Kane and HP, "PA - RISC, 2.0 Architecture book," HP, v.2.0, Feb. 1996.

[16] A. Moshovos, S. Breach, T. Vijaykumar, G. Sohi, “Dynamic speculation and synchronization of data dependences,” ACM SIGARCH and IEEE Computer Society, Computer Architecture News, 25(2), May 1997.

[17] A. Marquez, K. Theobald, X. Tang, G. Gao, “A Superstrand Architecture,” CAPSL Technical Memo 14, Dec. 1997.

[18] T. Sterling, “Proceeding of the 1996 Petaflops Architecture Workshop,” The Petaflops Systems Workshop, Caltech/JPL, Apr. 1996.

[19] J. Dongarra, J. Bunch, C. Moler, and G.W. Stewart, LINPACK User’s Guide, SIAM Publications, Philadelphia, 1979.

[20] H. Wijshoff, “Implementing Sparse BLAS Primitives on Concurrent/Vector Processors: a Case Study,” a book “Lectures on parallel computation,” edited by Alan Gibbons, Paul Spirakis, Cambridge Intl. Series on Parallel Computation: 4, Cambridge university press, 1993.

[21] R. Riesen, A. Maccabe, S. Wheat, “Active messages versus explicit message passing under SUNMOS,” in Proceedings of the Intel Supercomputer Users' Group, 1994 Annual North America Users' Conference, pp. 297-303, June 1994.

[22] Lewis W. Tucker, Alan Mainwaring, “CMMD: Active messages on the CM-5,” Parallel Computing, 20(4), pp. 481-496, April 1994.

[23] “Information Technology Research: Investing in Our Future”, President’s Information Technology Advisory Committee, Report to the President, Feb. 1999.

[24] Pieter Hintjens, Pascal Antonnaux. A portable multithreaded Web server. Sockets. Dr. Dobb’s Journal. Software careers #279, pages 39-43, Spring, 1997.

[25] James Shin Young, “NUMA for IRAM,” Berkeley, CA, 1996.

[26] “Major System Characteristics of the Tera Computer,” Tera company, Nov. 1995.

[27] D. Culler, A. Dusseau, S. Goldstein, et., “Parallel Programming in Split-C,” Computer Science Division, Berkeley, CA, 1993.

[28] L. Liu, D. Culler, “Evaluation of the Intel Paragon on Active Message Communication”, Computer Science Division, in Proceedings of Intel Supercomputer Users Group Conference, Berkeley, CA, June 1995.

[29] E. V. Krishnamurthy, *Parallel processing. Principle and practice*, Intl. computer science series, Addison-Westley publishing company, 1989.

[30] V. Reis, I. Scherson, “A Virtual Model for Parallel Supercomputers,” in Proceedings of IPPS’96, IEEE Computer Society, Apr. 1996.

[31] M. Wolfe, “More Iteration Space Tiling,” in Proceedings of Supercomputing’89, IEEE and ACM, Nov. 1989.

[32] T. Murata, “Petri Nets: Properties, Analysis and Applications,” in “Proceedings of the IEEE”, vol. 77, NO. 4, pp. 541-580, Apr. 1989.

[33] L. Kale, M. Bhandarkar, N. Jagathesan, S. Krishnan, J. Yelon, “Converse: An Interoperable Framework for Parallel Programming,” in Proceedings of IPPS’96, IEEE Computer Society, Apr. 1996.

[34] S. K. Reinhardt, J. R. Larus, D. A. Wood, “Tempest and Typhoon: User-level shared memory,” in Proc. of the 21st Annual ISCA, pp. 325-336, ILL, Apr. 1994.

[35] Cadan, Sherman, “Linda unites network systems,” IEEE Spectrum, Dec. 1993.

[36] J. Brockman, P. Kogge, V. Fressh, S. Kuntz, T. Sterling, “Microservers: A New Memory Semantics for Massively Parallel Computing,” ACM Intl. Conference on Supercomputing, Greece, June 1999.

[37] D. Zinoviev, G. Sazaklis, L. Wittie, K. Likharev, S. Yorozu, “CNET: RSFQ Switching Network for Petaflops Computing,” TR 08, HTMT RSFQ Group Release, SUNY, NY, Oct. 1998.

[38] D. Ziniviev, “Design Issues in Ultra-fast Ultra-Low-Power Superconductor Batcher-Banyan Switching Fabric on RSFQ Logic/Memory Family,” *Applied Superconductivity*, vol. 5, pp. 235-239, Aug. 1998.

[39] H. Ahmadi, W. Denzel, “A Survey of Modern High-Performance Switching techniques,” IEEE J. Selected Areas in Commun., v. 7, pp. 1091-1103, Sep. 1989.

[40] L. Wittie, G. Sazaklis, Y. Zhou, D. Zinoviev, “High Throughput Networks for Petaflops Computing,” Workshop on Parallel and Distributed Systems, Oct. 1998.

[41] S. Yorozu, D. Zinoviev, “Design and Implementation of an RSFQ Switching Node for Petaflops Networks,” IEEE Trans. on Appl. Supercond., 1999.

[42] SUNY RSFQ Group, “Document *netdoc1.ps*”, June 1997. Available via anonymous ftp from *ftp://rsfq1.physics.sunysb.edu/pub/netdoc1.ps*.

[43] D. Ziniviev, M. Maezawa, “Application of credit-based flow control to RSFQ micropipelines,” IEEE Trans. on Appl. Supercond. 1999.

[44] T. Sterling, L. Bergman, “A Design Analysis of a Hybrid Technology Multithreaded Architecture for Petaflops Scale Computation,” International Conference on Supercomputing (ICS’99), Greece, June 1999.

[45] K. Bergman, C. Reed, “Hybrid Technology Multithreaded Architecture Program Design and Development of the Data Vortex Network,” TN #17, Princeton TN2.0, May 1998.

[46] M. Arend, C. Reed, K. Bergman, “Physical Design and Specifications for the Data Vortex Network,” HTMT TN 033, Princeton, 1998.

[47] M. Arend, K. Bergman, C. Reed, “Data Vortex Packaging,” HTMT TN #23, Princeton, July 1998.

[48] T. Sterling, “A Hybrid Technology Multithreaded Computer Architecture for Petaflops Computing,” MS 159-79, JPL, CA, Jan. 1997.

[49] G. Gao, K. Likharev, P. Messina, T. Sterling, “Hybrid Technology Multi-Threaded Architecture: Project Summary. Project Description,” HTMT Reports, 1998.

[50] H. J. Hum, O.Maquelin, K. Theobald, X. Tian, G. Gao, L. Hendren, “A Study of the EARTH-MANNA Multithreaded System,” Intl. Journal of Parallel Programming, 24(4), pp.319-347, Aug. 1996.

[51] H. Hum, O. Maquelin, K. Theobald, X. Tian, X. Tang, G. Gao, et., “A Design Study of the EARTH Multiprocessor,” in Proc. of the IFIP WG 10.3, PACT’95, pp. 59-68, ACM Press, June 1995.

[52] Andrew S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, Inc., Englewood Cliffs, NJ 07632, 1995.

[53] H. Cai, “Dynamic load balancing on the EARTH-SP system,” MT, McGill U., Montreal, Que., May 1997.

[54] L. Hendren, X. Tang, Y. Zhu, G. Gao, X. Xue, H. Cai, P. Ouellet, “Compiling C for the EARTH multithreaded architecture,” in Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques, pp. 12-23, Mass., Oct. 1996.

[55] H. Hum, O. Maquelin, K. Theobald, X. Tian, G. Gao, L. Hendren, “A study of the EARTH-MANNA multithreaded system,” Intl. Journal of “Parallel Programming”, 24(4), pp. 319-347, Aug. 1996.

[56] G. Gao, X.Tang, P. Thulasiraman, K. Theobald, “An Overview of the Threaded-C Language,” CAPSL TN #1, Delaware, July 1997.

[57] D. Psaltis and F. Mok, “Holographic Meories,” Scientific American, Vol. 273, No.5, pp.70-76, Nov. 1995.

[58] I. Redmond, R. Linke, E. Chuang, and Psaltis, “Holographic Data Storage in a DX-Center Material,” Optics Letters 22: (15), pp. 1189-1191, Aug. 1997.

[59] K. Bergman, “Ultra-High Speed Optical LANs,” Conference on Optical Fiber Communications (OFC’98), Workshop on LANs and WANs, CA, Feb. 1998.

[60] P. Kogge, “The EXECUBE Approach to Massively Parallel Processing,” Intl. Conference on Parallel Processing, ILL, Aug. 1994.

[61] Paraskevas Evripidou, Jean-Luc Gaudiot, *Advanced Topics in Data-Flow Computing*, the Ch. “The USC Decoupled Multilevel Data-Flow Execution Model,” editors J. Gaudiot, L. Bic, Prentice Hall, NJ 1991.

[62] M. Dorojevets, P. Bunyk, D. Zinoviev, K. Likarev, “COOL-0: A Preliminary Design of an RSFQ Subsystem for Petaflops Computing”, IEEE Transactions on Applied Superconductivity, 1999.

[63] M. Dorojevets, “The COOL-1 ISA Handbook: v.1.00”, TR 11, SUNY, NY, Jan. 1999.

[64] B. Smith, “Architecture and applications of the HEP multiprocessor computer system,” in SPIE Real Time Signal Processing IV, NY: SPIE, pp. 241-248, 1981.

[65] M. Dorojevets, P. Wolcott, “The Elbrus-3 and MARS-M: Recent advances in Russian high-performance computing”, Journal on Supercomputing, v. 6, pp. 5-48, 1992.

[66] “A Revolutionary approach to the parallel programming: The Tera MTA”, WA: Tera Computer Company, http://www.tera.com.mta.html, 1998.

[67] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, T. Nishizava, “An elementary processor architecture with simultaneous instructions issuing from multiple threads,” in Proc. ISCA-15, IEEE Comput. Soc. Press, CA, pp. 443-451, 1988.

[68] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, D. Tullsen, “Simultaneous multithreading: a platform for next-generation processors”, IEEE Micro. Journal., vol. 17, pp. 12-19, Oct. 1997.

[69] P. Bunyk, M. Dorojevets, K. Likharev, P. Liskevich, S. Polonsky, G. Sazaklis, et., “RSFQ Subsystem for Petaflops-Scale Computing:’COOL-0’,” in Proc. 3rd Petaflop Workshop, pp. 3-9, MD, Feb. 1999.

[70] M. Noakes, D. Wallach, W. Dally, “The J-Machine Multicomputer: An Architectural Evaluation,” in Proc. of the 20th Intl. Symp. on Computer Architecture, May 1993.

[71] P. Kogge, J. Brockman, V. Freeh, “Processing-In-Memory Based Systems: Performance Evaluation Considerations,” PAID’98, Int. Symp. on Computer Arch., Spain, June 1998.

[72] L. Yerosheva, S. Kuntz, P. Kogge, J. Brockman, “A Microserver View on HTMT: New Benchmarks and Applications,” TR# 00-12, CSE, NN, Sep. 2000.

[73] M. Dwyer, V. Wallentine, “Object-oriented coordination abstractions for parallel software,” in Proceedings the Intl. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’97), June 1997.

[74] E. Anderson, J. Brooks, C. Gassi, S. Scott, “Performance analysis of the T3E multiprocessor,” in Proc. on ACM High Performance Networking and Computing, ACM Press and IEEE Computer Society Press, November 1997.

[75] “Solaris 2. Online Documents: Design and Implementation of Transport-Independent RPC,” http://www.sun.com/smcc/solaris-migration/docs/whitepapers.html#TI-RPC, 1991.

[76] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, et., “The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor,” in Scalable Shared Memory Multiprocessors, Kluwer Academic Publishers, 1991.

[77] C.-C. Chang, Grzegorz Czajkowski, Thorsten von Eicken, “Design and Performance of Active Messages on the SP-2,” Cornell CS Tech. Report 96-1572, February 1996.

[78] T.von Eicken, T., D. E. Culler, S. C. Goldstein, and K. E. Schauser, “Active Messages: a Mechanism for Integrated Communication and Computation,” Proc. of the 19th Int. Symp. on Computer Architecture, Australia, May 1992.

[79] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, J. Hennessy, “The DASH Prototype: Implementation and Performance,” Proc. of the 19th Intl. Symposium on Computer Architecture, pp. 92-103, Australia, May 1992.

[80] R. Arpaci, D. Culler, A. Krishnamurthy, S. Steinberg, K Yelick, “Empirical evaluation of the CRAY-T3D: A compiler perspective,” in Proceedings of the International Symposium on Computer Architecture, pp. 320--331, June 1995.

[81] “Silicon Graphics CRAY Origin2000,” http://www.ncsa.uiuc.edu/SCD/Hardware/Origin2000, 1997.

[82] “HP-Convex SPP-2000 User Modules: SPP-2000 Architecture,” http://www.ncsa.uiuc.edu/SCD/Hardware/SPP2000, 2000.

[83] “The Oxford BSP: The Bulk Synchronous Parallel Model,” http://oldwww.comlab.ox.ac.uk/oucl/oxpara/bsp/bspmodel.htm#bspcomp, 1996.

[84] G. Luecke, B. Raffin and J. Coyle, “Comparing the Communication Performance and Scalability of a SGI Origin 2000, a Cluster of Origin 2000’s and a Cray T3E-1200 using SHMEM and MPI Routines,” The Journal of PEMCS, IA, Oct. 1999.

[85] T. Sterling, D. Becker, D. Savarese, M. Berry, C. Reschke, “Achieving a Balanced Low-Cost Architecture for Mass Storage Management through Multiple Fast Ethernet Channels on the Beowulf,” Parallel Workstation Proceedings, 1996.

[86] ~http://www.beowulf.org/

[87] L. Carter, J. Feo, A. Snavely, “Performance and Programming Experience on the Tera MTA,” SIAM Conference on Parallel Processing, March 1999.

[88] P. Hut, J. Arnold, J Makino, S. McMillan, T. Sterling, “GRAPE-6: A Petaflops Prototype,” Invited paper, Petaflop Algorithm workshop, Virginia, 1997.

[89] ~http://www.science.uva.nl/research/scs/Research.html

[90] Object Management Group: COBRA (1990), ~http://www.omg.org/

[91] Dally, et al “The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms,” IEEE Micro, 4/92, pp.23-39.

[92] G.M. Papadopoulos and D. E. Culler, “Monsoon: an Explicit Token-Store Arch.,” In Proc. of the 17th Annual Int. Symp. on Comp. Arch., Seattle, May 1990.

[93] J.L.Peterson, “Petri Nets,” ACM Computing Surveys, Vol.9, No.3, Sep. 1977.

[94] J. L. Peterson, Petri Net Theory and the Modeling of Systems, Prentice-Hall, NJ, 1981.

[95] D. H. H. Ingalls, “Design Principles Behind Smalltalk,” BYTE Magazine, August 1981.

[96] G.Lehrenfeld, W. Müller, C. Tahedl, “Transforming SDL Diagrams Into a Complete Visual Representation,” Proc. 11th Int. IEEE Symp. on Visual Languages, Cadlab Institute, Germany, 1996.

[97] N. Carriero, D. Gelernter, “How to write parallel programs: A Guide to the Perplexed,” ACM Computing Surveys, Vol.21, No.3, September 1989.

[98] ~http://www.aut.utt.ro/%7Emappy/petri/nets/ARTIF.html

[99] E. Kindler, W. Reisig, Freksa C., Jantzen M., Valk R., “Verification of distributed algorithms with algebraic Petri nets.”, in “Lecture Notes in CS: Foundations of Computer Science: Potential, Theory, Cognition”, Springer-Verlag, v. 1337, pp. 261--270,1997.

[100] J. Kurt, “Coloured Petri Nets: A High Level Language for System Design and Analysis,” Lecture Notes in Computer Science, v. 483, Advances in Petri Nets, Springer-Verlag, 1991.

[101] Buchs D., Guelfi N., “System Specification Using CO-OPN,” Lab. de Recherche en Informatique, France LRI Rapport de Recherche, May 1991.

[102] B. Bonchev, “Hierarchical Modeling of Parallel Systems with Extended Timed Nets,” Petri Net Newsletter # 40, pp. 8-16. Bonn, Germany, Dec. 1991.

[103] T. Christopher, “Experience With Message Driven Computing and the Language LLMDC/C,” Proc. of the ISMM Int. Conf. Paral. and Distr. Computing and Systems, New York, Oct. 1990.

[104] G. Chiola, G. Ciaccio, “Implementing a Low Cost, Low Latency Parallel Platform,” in Parallel Computing 22, pp. 1703-1717, 1997.

[105] G. Chiola,G. Ciaccio, “GAMMA: a Low-cost Network of Workstations Based on Active Messages,” March 1995.

[106] Lilia Yerosheva, Peter Kogge, “Prototyping execution models for HTMT Petaflop machine in Java”, in the book “Network-Based Parallel Computing, Comm. Arch., and Applications” by A. Sivasubramaniam, M. Laura (Eds.), “Lecture Notes in Computer Science. 1602", Springer, pp. 32-47, April 1999.

[107] Lilia Yerosheva, Peter Kogge, “Prototyping HTMT execution model using Petri Nets”, Proc. of the SCI’99, FL, July 1999.

[108] Lilia Yerosheva, Shannon Kuntz, Peter M. Kogge, J. Brockman, “Microserver View of HTMT,” will appear in Proc. for IPDPS 2001, CA, April 2001.

[109] Lilia Yerosheva, Shannon Kuntz, P. M. Kogge, “Memory partitioning in the HTMT high-performance parallel system,” subm. to PACT 2001.