University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

A Refinement of the HTMT Program Execution Model

Guang Gao
José Nelson Amaral
Andres Marquez
Kevin Theobald

CAPSL Technical Memo 22
July 13, 1998

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

We refine the specification of the program execution model for the HTMT architecture that has been presented in previous documents and discussed in HTMT workshops. The parcel percolation model is presented in greater detail in this memo along with a discussion of the functionality of each one of the sub-units and concurrent data structures in the SRAM-PIM runtime system. We discuss the role of a future high-level compiler and its relation with the runtime system and the application program interface. We propose a first definition for an HTMT-Threaded-C language that is an extension of Threaded-C. The advantage of this extension is that it is simple to implement and allows the application scientists to use their knowledge of Threaded-C to write programs in the new language. In this language we assume an automatic address translation for data stored in the CRAM. Such address translation will be provided by the HTMT runtime system. We conclude with a detailed presentation of a block matrix multiplication algorithm implemented in the new HTMT-Threaded-C language.

Note: We invite application scientists to examine this implementation and provide comments on extensions and improvements to the HTMT-Threaded-C language.

Contents

List of Figures

1 Introduction

We present a refinement of the Program Execution Model (PXM) for the HTMT machine. In this document we concentrate on the relation between the activities in the SRAM-PIM level and the processing in the cryogenic area. Although we are interested in a high level description of the program execution model, we feel that it is necessary to sketch some lower level mechanisms that will help in the understanding of the model. We present the ideas in this document as a proposition for discussion. We expect that other HTMT groups will provide feedback pointing to areas in our model that could be improved to allow for an easier or more efficient implementation or that would result in a better application program interface.

Central ideas in the program execution model proposed in this document are originate from the extensive experience that the Delaware team has acquired with the multithreading program execution model and the multithreaded language developed for EARTH [6, 18]. The initial design of the HTMT-Threaded-C language presented here is a simple extension of the Threaded-C language. Thus the HTMT project will be able to benefit from the joint effort and investment of the McGill/Delaware group in the development of the EARTH architecture and the Threaded-C language. Another benefit of choosing an extension of Threaded-C as a first emulator for the HTMT project is the fact that Threaded-C is currently operational on a number of important parallel platforms.

This document builds on a number of documents, discussions, and research efforts in Delaware and elsewhere. In our July 1997 Technical Memo 09 we presented the concept of a percolation model [8]. The model introduced in that document is presented at a high level to allow any configuration that the yet to be established HTMT architecture would assume. This document maintains that principle, but a clearer idea about the HTMT machine collectively produced during this year allows us to specify the execution model in further detail. We revisit the concepts of *parcels*, *threads*, and *strands*, describe in more detail the percolation model, introduce the skeleton of an HTMT-Threaded-C language and present an example illustrating how an application program can be written in such a language.

During the preparation of this document, the exact amount of memory that the HTMT hardware will be able to implement in the cryogenic area has not yet been established. On the other hand it is clear from discussions in the applications group that application programmers would like to see *some* amount of memory available for code and data there. We assume that it will be possible to store a limited amount of data and code in the cryogenic area. We suggest that part of the SRAM area be used as an extension of the cryogenic memory and that these two pieces are to be referenced by the program as a single entity. This assumption allows us to propose a PXM that can last through the evolution of the machine when advances in technology will allow for a more substantial amount of memory to be implemented in the cryogenic area.

In related studies the Delaware group has explored ways to achieve high levels of parallelism at the instruction level without incurring a great penalty in the real estate required for control flow and synchronization mechanisms in the hardware implementation of the machine [14, 15]. The Superstrand Architecture introduces the notion of a *strand* as a block of instructions grouped together to become a scheduling quantum of execution. The first experiments with this architecture indicate that programs can be efficiently partitioned into strands to be executed under a superstrand execution model.

In a different line of research, we considered the adoption of a data-flow approach for the implementation of HTMT. In that preliminary study the processor architecture regards an instruction as the smallest unit of scheduling and synchronization. If the predictions that all the delays are unpredictable in a super-conducting processor are confirmed, the data-flow approach proposed by Theobald *et al.* should be revisited [19].

A very interesting challenge in the implementation of modern parallel machine is the definition of a appropriate memory consistency model. While a sequential machine works under the assumption of sequential consistency, it is difficult to extract significant parallelism under such a restrictive model. Gao and Sarkar have proposed a weak memory consistency model that allows for elimination of unnecessary synchronization operations [7]. Under their location consistency model, synchronization only occurs when it is explicitly required by the programmer/compiler. Because of the massive parallelism that needs to be explored for an efficient implementation of the HTMT machine, we anticipate that a weak memory consistency model will be adopted.

The description of a Program Execution Model (PXM) must keep in close relation with both the Application Program Interface and the technology that will be used to build the machine. Whenever possible we provide a scenario that clarifies how the mechanisms included in this PXM could be implemented in HTMT technology and how they could be used in an API.

This document is organized as follows. In Section 2 we present a high level description of the memory model and discuss how the programmer will see the memory segmented in different regions. We discuss the relationship between the HTMT-Threaded-C language introduced in this document, and a future high-level compiler in Section 3. In Section 4 we describe the percolation model implemented by the HTMT runtime system. We present an overview of the operations performed by the HTMT-Threaded-C language in Section 5. We describe in Section 6 the primitives added to the standard Threaded-C language to implement the percolation model of HTMT and in Section 7 the first program implemented in HTMT-Threaded-C. In Sections 8 and 9 we present issues to be addressed in near future work such as integrating runtime statistics in the HTMT primitives.

2 Memory Model

Conventional architectures present to the programmer the appearance of an uniform address space below the registers, with caching and paging hiding the real details of the hierarchy from the programmer. This luxury is not available in the HTMT model. Instead, each level of memory is considered to be a "buffer" of the next level, e.g., super-conducting memory is a buffer of the SRAM, which itself is a buffer of DRAM, and so on. Unlike traditional cache organization, these buffers are directly addressable. Buffer storage allocation and data movement at each level can be fully or partially under the programmer's control. That is, the data unit and size of a memory transaction, the data movement up or down the memory levels, and the allocation of the buffer storage may all be explicitly controlled if necessary [8].

We assume that the whole memory address space is explicitly divided into regions: a CRAM region, a SRAM region (including SRAM-buffer), a DRAM region, and an Optical region. The actual sizes of each region will be initialized at system configuration time. A memory location can thus be addressed by giving the region name and the corresponding offset.

In this section we concentrate on the relation between the processing and storage of data at the PIM associated with the SRAM and the processing at the SPELL (Superconductor Processing ELement) in the CRAM region. It is our understanding that part of the memory associated with the processing of data at the SPELL level must be seen by the Application Program Interface (API) as a “heap memory”, i.e. it could be shared by multiple functions through pointer exchange.

We adopt a programming model in which the programmer does not manipulate memory addresses in the CRAM region. The programmer can request the dispatch of data to the CRAM region and can also reserve space in the CRAM region. However any reference to data stored in CRAM goes through an automatic address translation that maps the SRAM address used in the program to the actual address of the data in CRAM. A strong motivation for this model is to allow flexibility in the implementation of the CRAM. Because CRAM addresses do not appear explicitly in the code, future implementations of HTMT will be able to implement new mappings of the data dispatching and reservation of memory in CRAM.

If the memory capacity in the cryogenic area is very limited, there might be little or no room to allocate even a small heap to contain data shared by multiple threads. In this case the heap will reside in the SRAM memory. Portions of the SRAM will be designated as “SRAM buffers” to be used for the heap allocation. The amount of SRAM space dedicated for these buffers is determined during the configuration of the machine.

The application programmer can view the SRAM buffer as an extension of the CRAM region. Any data movement and address mapping between a location in the SRAM buffer and a location in the CRAM region is handled automatically by the hardware and runtime system. The evolution of technology is bound to allow the future implementation of larger memories at the SPELL level, eventually allowing for this conceptual heap to reside entirely in the CRAM region with the consequent vanishing of the SRAM buffer. Thus it is important that this mapping between SRAM buffer and CRAM be transparent for the application program in the first version of HTMT.

We assume that for each SPELL, there is an area in the SRAM that is “closer” (in terms of latency) to that SPELL than any other SRAM location. Therefore each pair of SPELL and SRAM-PIM has its own heap. Notice that if the interconnection between the SPELLs and the SRAM-PIMs is such that the cost of transmitting data from anywhere in the SRAM region to anywhere in the CRAM region is roughly the same, the program execution model would be simplified. In such a case the heap allocation would not have to take into consideration locality properties. On the other hand the application performance might suffer because the technology is not taking advantage of naturally existing spatial locality properties.

Although operating system virtual memory support is not an inherent part of the HTMT program execution model, there may be an advantage to providing a virtual mapping of addresses within a region in the lower levels of memory (DRAM and Optical memory). Such support would allow for the implementation of basic multitasking and multiuser functions.

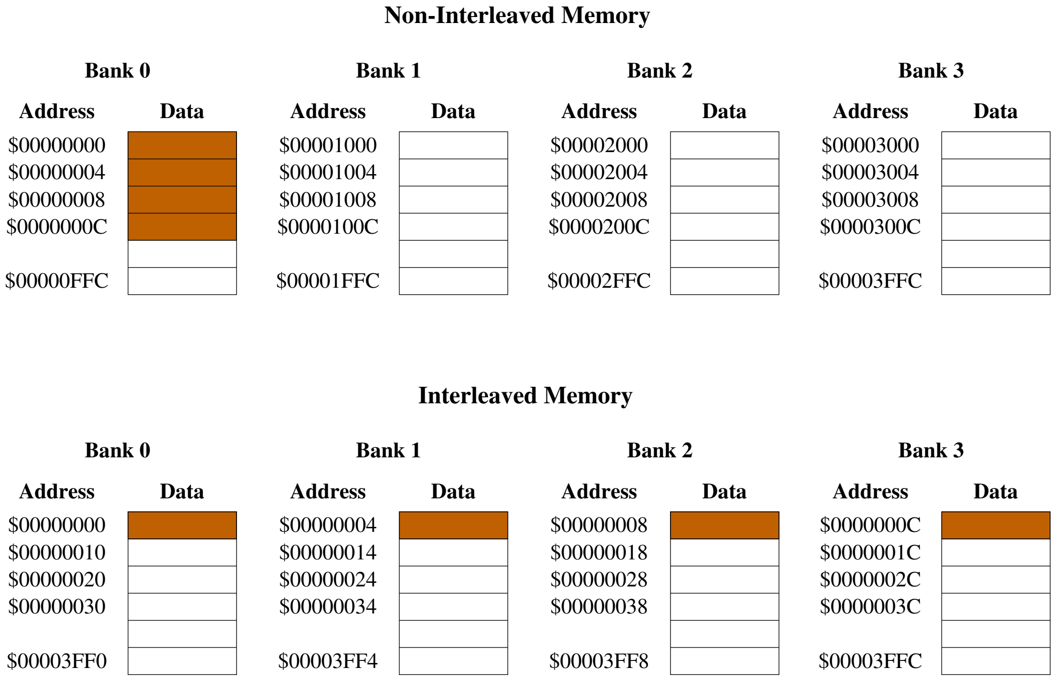

Figure 1: Illustration of Memory Interleaving.

In the HTMT architecture, the need to ensure locality and avoid memory misses means that memory sharing must be less transparent than in conventional systems. The HTMT memory model will provide the programmers with full and explicit control over the consistency of copies of data in the memory hierarchy[1]. It cannot rely on a "hidden" coherence management mechanism – like some popular cache-coherence schemes in conventional shared memory multiprocessor systems – the performance of which is in question.

If a program moves a memory object from region X to region Y explicitly, we should expect that the rest of the program, and any other programs that interact with it, are aware of this movement and correct code is in place to access the moved memory object in region Y.

Memory interleaving is an established technique that optimizes access to streams of data stored in consecutive memory addresses. If a memory module is implemented over a set of memory banks, the distribution of the addresses over these banks affects the access time to

[1] In a future implementation of the HTMT system, the compiler in cooperation with the runtime system might provide memory consistency management. For efficiency some of the higher level language constructs might provide some "hints" to the compiler/runtime system to help reduce the number of synchronization operations performed.

the memory. Consecutive memory accesses require more time if they are served by the same memory bank than if they are directed to different banks.

Figure 1 illustrates the concept of memory interleaving for a memory with four memory banks. If the four first words in the memory are accessed consecutively, in the non-interleaved memory all four accesses will be directed to Bank 0. In the interleaved memory each access will be directed to a different memory bank. Observe that with the simple interleaving mechanism illustrated in the figure, access with an address stride of four words will hit the same memory bank. Advanced interleaving techniques, that are beyond the scope of this document, are available to cause regular stride accesses to be distributed among the memory banks [10].

Each region of the HTMT memory will be divided into an interleaved segment and a non-interleaved segment. A memory allocation primitive in HTMT-Threaded-C allows the programmer to specify that a given memory allocation is to use the interleaved portion of the memory.

3 The Role of the Compiler and of the Runtime System

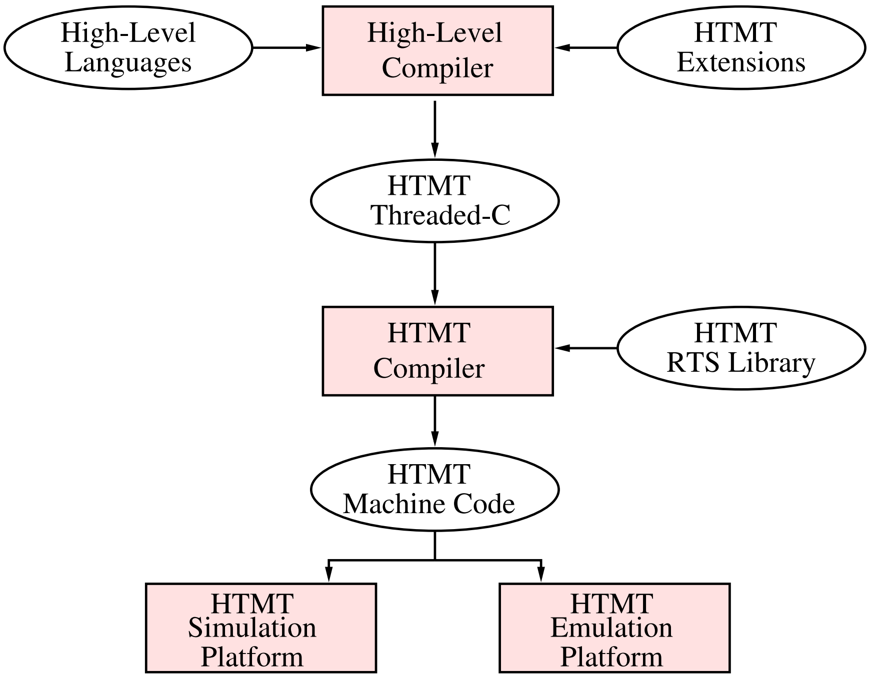

Figure 2: Relation between a Future HTMT high-level compiler and the HTMT-Threaded-C language.

By supporting many threads of control and fast switching among threads, multithreaded architectures can tolerate inherent communication and synchronization latencies by switching to a new ready thread of control whenever a long-latency operation is encountered [4, 5, 13, 16, 17]. However, providing support for multithreading at the architecture level is not enough, we must also be able to effectively program such architectures. One alternative is to make the threads and synchronization between threads explicit in the programming language, and ask the programmer to handle all of the details of thread creation, communication, and synchronization between threads. While this approach gives the full flexibility of multithreading to the user, it tends to introduce many low-level details that are hard for the programmer to handle. Thus, it is unlikely that such a programming model would become generally acceptable. A preferable approach is to give the programmer a more traditional high-level language that does not explicitly support threads, and to use selected language extensions and compilation techniques to automatically translate high-level programs into lower-level threaded programs. Thus, with minimal effort, the programmer should be able to benefit from the power of multithreading.

Figure 2 illustrates the relationship between an HTMT high-level compiler and the HTMT-Threaded-C Language. In a future development of HTMT, application programmers will be able to write programs in high level parallel languages, such as parallel C and parallel Fortran, with HTMT extensions and have a compiler translating the program into an explicitly multithreaded language such as HTMT-Threaded-C. However the development of such a high level compiler is expensive and requires a considerable amount of time to be completed. Among the desired features of such a high level compiler and its associated libraries we highlight:

- a library of functions to perform data transformations in the SRAM region and in the DRAM region to prepare the data for percolation;
- automatic generation of multithreaded programs from high level parallel constructs specified by the user.
- support for the runtime partition of threads into strands;
- automatic generation of the data movement primitives (such as DATA_DISPATCH and DATA_RETURN) required to implement the percolation model.

However in this first exploratory phase of the project, neither the resources nor the time to implement such a compiler are available. Therefore the pioneer programmers of HTMT will have to code their problems in a lower level explicitly multithreaded language that requires more effort to implement the correct synchronization mechanism than what would be desired by an application programmer.

The HTMT-Threaded-C language is explicitly multithreaded and requires a set of primitives to implement the percolation model. The functionality described by the semantics of this primitives is implemented by the HTMT Runtime System (RTS). For instance when a DISPATCH_PARCEL primitive is executed, a request to transfer the code to CRAM must be issued, and once the code is transferred, it must be enabled for execution. The activation of the hardware necessary to perform such tasks is provided by the runtime system. Among the responsibilities of the runtime system are:

- Implement the thread synchronization mechanism through synchronization slots;

- Implement the percolation model providing primitives for the percolation of data and code;
- Provide the automatic translation of addresses from the SRAM region to the CRAM region;

Even in the earliest versions of an HTMT emulator, an application programmer must always have the illusion that a complete and functional runtime system is available, i.e., the semantics of the HTMT-Threaded-C language must be fully implemented before the first program can run.

Throughout our discussions in this document we will describe program constructs that must be provided by the "programmer", such as runtime system primitive calls, data assembling functions, synchronization slot initializations, invocation of synchronization primitives, etc. In the first emulation of HTMT these actions are actually provided by a programmer. In a future phase of the project, a higher level language will be defined, and a compiler will be provided to perform most of the actions described in this document as the "programmer" responsibility.

4 The HTMT Runtime Percolation Model

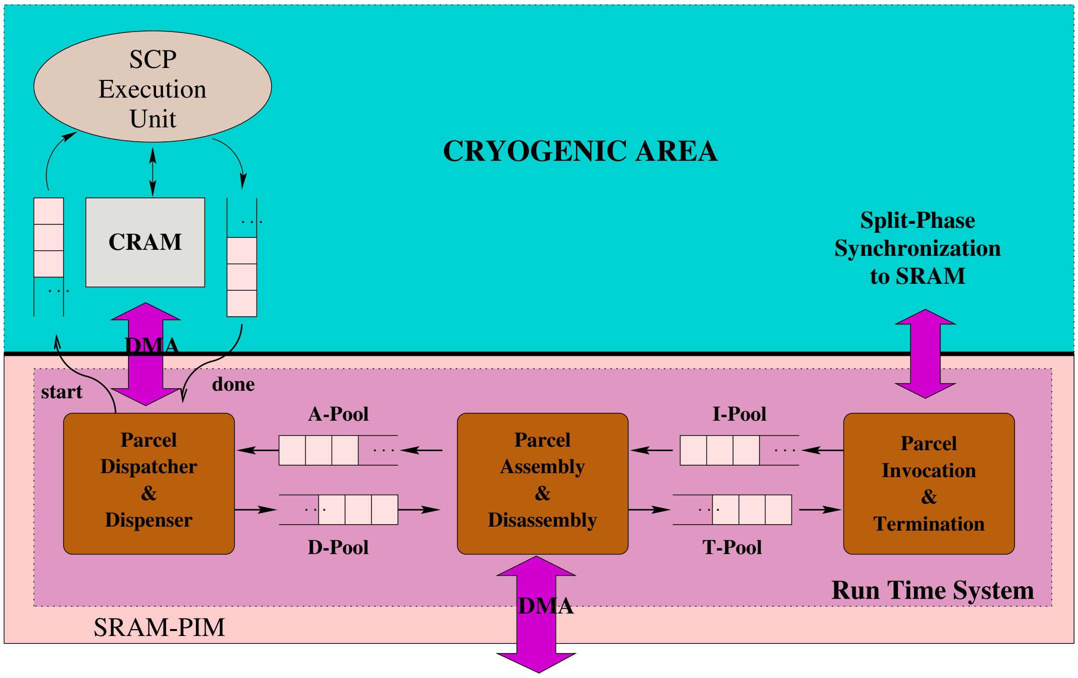

Figure 3: HTMT Runtime Percolation Model.

This section describes the HTMT percolation model implemented by the HTMT runtime system running in SRAM-PIM. The runtime system consists of three major components: the

Parcel Invocation and Termination Module (PIT), the Parcel Assembly and Disassembly Module (PAD), and the Parcel Dispatcher and Dispenser Module (PDD). Various concurrent data structures are used to connect these major components; I-pool, T-pool, A-pool, and D-pool. The relation among this modules and data structures is shown in Figure 3.

The main role of the PIT invocation manager is to detect if a parcel function has become enabled, i.e. whether all its dependences have been satisfied. The invocation manager will invoke enabled parcel functions and put them in the I-pool. From here on we will refes parcel trheaded functions simply as parcels.

The PAD module will take enabled parcels from the I-pool and prepare them for further percolation. The role of the PAD assembly manager is to move the required code and data into local SRAM locations. At the same time, the data should be organized into desired structure (via gather/scatter, generalized "corner turn", permutation, pointer swizzling, etc.). This usually involves data movement from DRAM into SRAM through special PIM operations via the DMA (Direct Memory Access) channel between DRAM and SRAM as shown in Figure 3. Once the assembly process is completed, a parcel is passed on to the next stage by entering it into the A-pool.

The PDD module selects parcels from the A-pool and moves them further up to the cryogenic region. The role of the PDD manager is to first reserve space in the CRAM region and then move the data and code associated with the parcel into the reserved region. This movement uses the DMA channel between SRAM and CRAM. After this is completed, the parcel has completed its percolation process and can start execution in the cryogenic region once the superconducting processing resource becomes available.

After a parcel finishes its execution in the cryogenic region, it needs to be retired from it. This is begun by the PDD dispenser manager. A completed parcel has its return data (if any) structured and stored in the C-buffer for post-processing. The dispenser copies the C-buffer to SRAM and immediately deallocates the CRAM resource reserved by the parcel. It then enters the parcel into the D-pool.

The PAD disassembly manager processes the parcels from the D-pool, disassembling and distributing output data into its proper places. It then releases the SRAM space occupied by the C-buffer copy. When the disassembly process is finished, the parcel is entered into the T-pool for the final termination service.

The PIT module will take parcels from the T-pool for termination processing. The role of the PIT termination manager is to inform the dependent successor parcels that the parcel under consideration has completed it execution. This may cause its successors to become enabled, the beginning of another percolation process.

4.1 Extensions to the Base Model

First, there is a short-cut path between the D-pool and the A-pool within the PAD module. This is provided for the case in which the parcel from the D-pool will be immediately re-enabled, so that its required data and code do not need to go through the disassembly and reassembly processes again.

Another extension is the connection between the cryogenic area and the PIM-resident PIT module. This is provided for the case when the code running in the superconducting processors may need to initiate the execution of a PIM based operation, say a reduction or scan operation, but it may not be efficient to retire the whole parcel from the cryogenic region. A split-phase synchronization operation can be performed by a superconducting processor for this purpose so that its execution does not have to be blocked. When the PIM finishes the operation, a synchronization signal is sent back to the cryogenic region so the code waiting for the result can continue.

4.2 Managing Concurrent Data Structures

Note that the runtime system as described is highly concurrent in itself. All the three modules can process different parcels in a pipelined fashion.

We anticipate that, depending on the application and the resource requirement, a family of policies may be provided at runtime for the management of the concurrent data structures. For example, if a collection of parcels is sharing some data or code (e.g. from the same loop), a good policy might try to schedule them closely together in time so that code and data reuse can be achieved.

5 An Overview of HTMT-Threaded-C

This first version of HTMT-Threaded-C considers that the percolation process in the HTMT runtime system can be described as a set of phases. The program example in Section 7 will make it clear that these phases can overlap to implement computation in a pipeline fashion with data sharing among iterations of an algorithm.

DATA ASSEMBLY Performs data transformations required before the data can be transfered to the CRAM region. In a future implementation of HTMT-Threaded-C a collection of functions for data transformation will be provided. For the time being the programmer will construct the routines to perform such data transformations.

DATA DISPATCH The transformed data is moved to the CRAM region. Previously reserved space in CRAM is used to store the dispatched data. If no space has been previously reserved, the data dispatch phase allocates the space necessary for the data storage.

PARCEL DISPATCH Assembles a parcel with the code of the parcel threaded function to be executed in the SPELL. Parcel dispatch is part of the functionality of the `DISPATCH_PARCEL()` primitive.

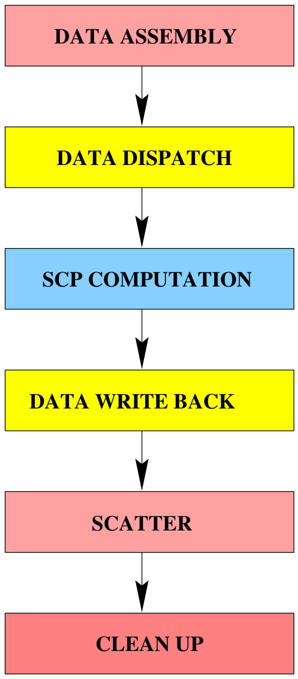

Figure 4: Phases of the Percolation for HTMT-Threaded-C.

SPELL COMPUTATION Performs the specified computation in the cryogenic processor. The data used during this phase is referenced by its address in the SRAM region. In HTMT-Threaded-C this computation is specified by a PARCEL THREADED function. The parcel threaded function is invoked by the `DISPATCH_PARCEL()` primitive. This invocation causes a parcel to be assembled and dispatched to the cryogenic region for execution.

DATA WRITE BACK Writes back to the SRAM region the results produced by the computation. The data to be returned is refered by its SRAM address. The runtime system will provide the address to locate the data in the CRAM region.

SCATTER Apply transformation to the results that have been transferred back from the CRAM region and store them in appropriate locations in SRAM. In a future implementation of HTMT standard functions for the scattering transformations will also be provided, but for the time being the programmer must supply the scattering functions.

CLEAN UP Release memory that has been reserved in the CRAM region to allow the start of the next percolation.

The phases of computation are presented in Figure 4. Although this Figures indicate a sequential execution of the phases, the actual program execution model allows for the overlaping of the percolation phases. For instance, after the data for one iteration of the computation has been assembled and dispatched, the assembly of the data needed in the next iteration can start while the SPELL computation of the first iteration is performed. To allow the overlaping of computation phases the runtime system implements a synchronization mechanism based on synchronization slots that allows for the specification of multithreaded programs in the SRAM-PIM level. Such a synchronization mechanism was previously implemented in the EARTH system [18].

6 A Skeleton for an HTMT-Threaded-C Language

The HTMT-Threaded-C language uses a synchronization mechanism based on synchronization slots and threads that is identical to the one implemented in the Earth architecture [11]. Therefore all the primitives described in the Threaded-C documentation are available in the HTMT-Threaded-C language [18]. In this section we introduce a set of new primitives that are necessary to implement the functionality of HTMT and that are not available in the original Threaded-C language. The combination of the original Threaded-C constructs and synchronization mechanism with this new set of primitives will be enough to implement the HTMT Program Execution Model in the first HTMT emulator.

HTMT_INIT()

Initializes internal structures for the memory management of the HTMT emulator. This function must be invoked in all the nodes once before any memory allocation, reservation, or data dispatch can take place.

DISPATCH_DATA(void SRAM *block, int size, SPTR slot1, SPTR slot2)

If there space was previously allocated in CRAM for the corresponding SRAM block, there is no need for new allocation. Otherwise allocates space in the CRAM heap region for a block of the size specified. Copy the data from the address specified by block into the area allocated in CRAM. When the copying is completed, synchronizes the slots specified by slot1 and slot2.

Before allocating space in CRAM, DISPATCH_DATA verifies whether space has been already reserved for block in CRAM. If a reserved space is found, it is used and no new space is allocated.

DISPATCH_PARCEL(parcel_threaded_function_name, ...)

Starts the assembly of a parcel containing the code and the parameters for the parcel threaded function specified. After the parcel is assembled, requests that the Parcel Dispatch and Dispenser send it to the CRAM region through the DMA channel.

Before assembling the parcel, DISPATCH_PARCEL verifies whether the code of the parcel threaded function is already loaded into the CRAM. In this case it assembles a parcel containing only the parameters to be dispatched to the CRAM region.

DATA_RETURN(void SRAM *block, int size, SPTR slot1, SPTR slot2)

Copy a block of data of the size specified from the CRAM to the SRAM. The translation of the SRAM address into a CRAM address is provided by the runtime system. After the data transfer is complete, both slot1 and slot2 are synchronized.

RESERVE(void SRAM *block, int size)
Reserve a block of the size specified in CRAM. The address of the allocated block will be associated by the runtime system with the SRAM address of block. Future references to the reserved region will use the SRAM address that will be translated into a CRAM address by the runtime system.

void SRAM *SRAM_ALLOCATE(int size)
Allocates space for a block of size in the SRAM, returns a void pointer to the SRAM.

CRAM_RELEASE(void SRAM *block)
Releases a block previously allocated in CRAM. Observe that the RTS will perform the translation of SRAM address. The allocation of the CRAM block might have been performed by a RESERVE primitive or by a DISPATCH_DATA primitive.

SRAM_RELEASE(void SRAM *block)
Releases a block of memory that has been previously allocated by the SRAM_ALLOCATE primitive.

7 A HTMT Programming Example: The Matrix Multiplication

In this section we use the matrix multiplication algorithm as a case study to illustrate how an HTMT program language can be derived from the percolation model described in Section 4.

We implement the block matrix multiplication algorithm described in [9]. A matrix A of dimension $m \times k$ is to be multiplied by a matrix B of dimension $k \times n$ to produce a matrix C of dimension $m \times n$. We assume that the matrices to be multiplied are too large to fit in the SRAM of a PIM. We also assume that the matrices have already been divided into blocks in such way that a block of each matrix, A, B, and C can fit in the SRAM. Moreover a block of the matrix A and a block of the matrix B have been loaded into the SRAM. The program that we will present has to multiply this block of matrix A by the block of matrix B to produce a block of matrix C. As we continue with the description we recommend that the reader refer often to the code in the file `matrix3.c` that is an Appendix to this document.

The block of matrix A has dimension `SA_rows` × `SA_columns`, the block of matrix B has dimension `SA_columns` × `SB_columns`. The block of matrix C to be produced has dimensions `SA_rows` × `SB_columns`. The threaded function `multiply_block` receives three pointers for these blocks: `SA`, `SB`, and `SC`.

We assume that the blocks SA, SB and SC are too large to fit in the memory available in the cryogenic area. Therefore each of this blocks has to be divided into a number of sub-blocks. The size of these sub-blocks is determined based on the amount of memory available for data storage in the cryogenic area. We assume that there is a system supplied value called `CRAM_HEAP_AREA` that contains the amount of memory, measured in bytes, available for heap allocation in the cryogenic area[2].

This example assumes that each of the blocks of A, B, and C is divided in the same number of sub-blocks. It also assumes that the ratio rows/columns in the sub-blocks is the same as the one in the original blocks. Therefore the size of the sub-blocks can be determined by a parameter that is identified in the program as `num_divisions`. This parameter must obey the following constraint:

$$\frac{\texttt{S\_memory\_used}}{\texttt{num\_divisions}^2} \leq \texttt{CRAM\_HEAP\_AREA} \tag{1}$$

Where `S_memory_used` is the amount of memory used by the original blocks of A, B, and C; `num_divisions` is the parameter that we want to determine to decide the size of the sub-blocks and `CRAM_HEAP_AREA` is the system supplied value for the amount of memory available for data storage in the cryogenic area. To minimize the number of parcels that need to be assembled and dispatched, we would like to use the smallest number of divisions that allows for a sub-block of A, B, and C to fit in CRAM.

Once we obtain a suitable `num_divisions` value, we can determine the number of rows and columns and the size of each sub-block. We can then allocate space in SRAM for each one of these sub-block. These data buffers are designated as `BA`, `BB`, and `BC`. These contiguous blocks of memory are dynamically allocated in SRAM.

In this first program we are using a naive algorithm to implement matrix multiplication. A block of A and a block of B are sent to the cryogenic area, the multiplication is performed, and the resulting block of C is sent back for storage in the SRAM. We do not reuse data in the cryogenic area. Later we will discuss some optimizations.

Figure 5 illustrates how the code manipulates data. The function `corner_turn()` copies a sub-block of matrix A or matrix B into the buffer `BA` or `BB`. The data is then transfered from these buffers to the cryogenic area by the DISPATCH_DATA() primitive. During the computation the product of a sub-block of A by a sub-block of B is added to the sub-block of C that was reserved in the CRAM by the RESERVE() primitive.

When all the necessary computations to compute one sub-block of C are completed, the resulting sub-block is copied back to the `BC` buffer in SRAM. The function reverse_corner_turn() is then used to add the results to the respective sub-block of matrix C.

For the matrix multiplication algorithm that we are using in this program example, there are some data dependencies between phases of the computation across iterations. For instance we are storing the assembled data in a data buffer in SRAM. Therefore the assembly of the data for the next iteration cannot start before the dispatching of the data in the current iteration

[2]This value can be fixed in the runtime system and available at compile time, or it might change dynamically and be provided by the runtime system. The value informed by `CRAM_HEAP_AREA` includes the memory available as SRAM buffer.

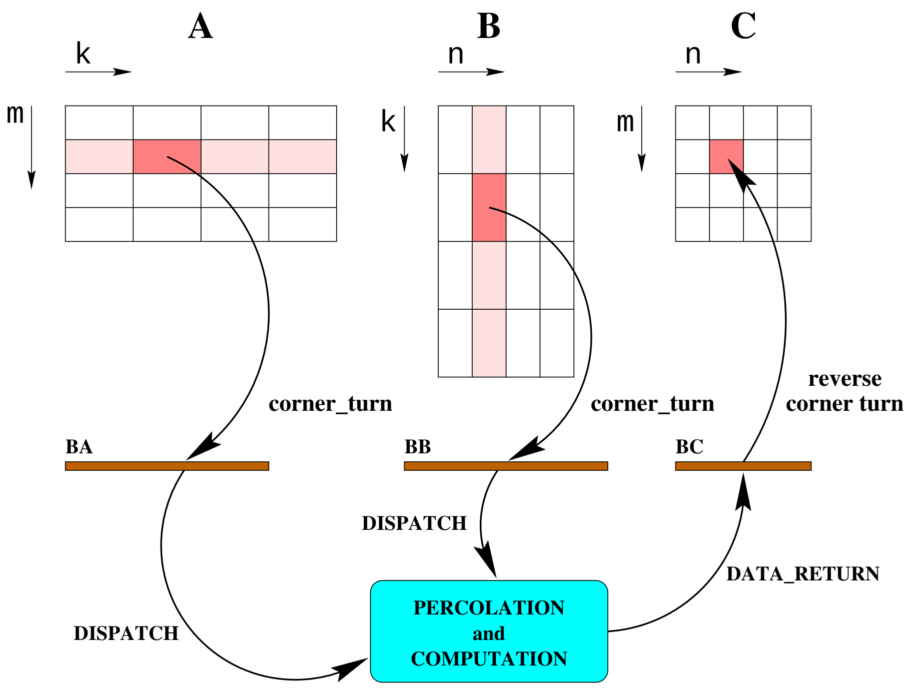

Figure 5: Corner Turn for the Matrix Multiplication Algorithm

has finished. Likewise, because we store the results of the computation in the buffer `BC` in the SRAM area, the write back of the data produced in the next iteration cannot start before the scattering of the data produced in the current iteration has finished. Also the next computation cannot start before the write back of the results of the current computation is completed. These dependencies are shown as full lines in Figure 7.

7.1 A Detailed Examination of the Code

In this section we closely examine the code in the file `matrix.c` to find out how the HTMT program execution model can be implemented using the thread and slot based synchronization mechanism.

After the sizes of the data blocks are computed, the primitive RESERVE is used to reserve enough space for one block of matrix C in CRAM. This primitive reserves space and updates the address conversion tables but does not transfer any data to the reserved space. Typically RESERVE is used to reserve the space necessary to store the results of the computation performed in the CRAM area.

As illustrated in Figure 5, to produce a sub-block of C, it is necessary to multiply a row of blocks of A by a column of blocks of B and accumulate the results. We could dispatch a subblock of A and a sub-block of B before multiplying these subblocks and accumulating the result in a sub-block of C in CRAM. However, to illustrate the retention of data in CRAM between computation phases, we decided to dispatch one sub-block of A to the CRAM region

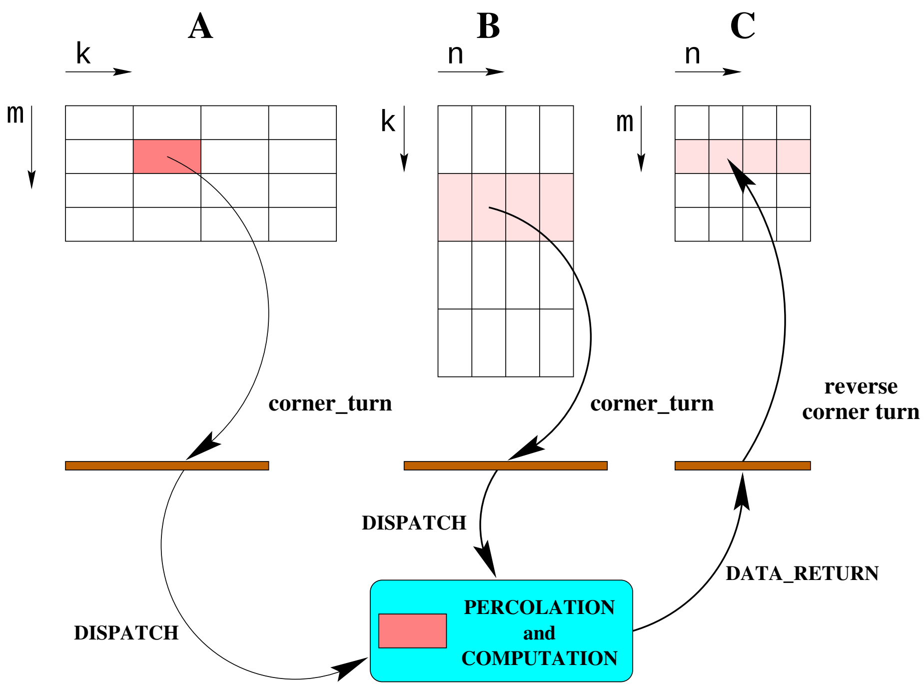

Figure 6: A matrix multiplication algorithm that reuses a sub-block of A in the cryogenic area.

and keep this sub-block there while we consecutively dispatch a row of sub-blocks of B. This scheme is illustrated in Figure 6. A drawback of our approach is that now the accumulation of the partial multiplication results must be performed along with the scattering of the sub-blocks of C.

The dependences among the threads are shown in Figure 7. The arrows in the figure represent synchronization signals. For instance, the synchronization slot associated with the THREAD_A_DISPATCH has an initial count of one and a reset count of two. The first time that this thread is spawned, it is synchronized only once by the end of the THREAD_A_GATHER. However the second time, and for any iteration thereafter, it also has to receive a synchronization signal from the THREAD_A_DISPATCH to assure that new data is not dispatched to the CRAM region before the computation of the previous iteration is concluded.

The data assembly phase is divided into two threads because they might run in parallel. The THREAD_A_GATHER and the THREAD_B_GATHER perform the gathering of data for the sub-blocks of matrices A and B through invocations of the function `corner_turn()`. The `corner_turn()` function transforms a sub-block of the matrix into a contiguously allocated vector as illustrated in Figure 5. Once the data is assembled, the DATA_DISPATCH slots of the dispatching threads are synchronized causing the THREAD_A_DISPATCH and THREAD_B_DISPATCH to be spawned. Observe that these syncrhonization slots are initialized with a count of one, but have a reset count of two because the next dispatches should not start before the current computation is finished.

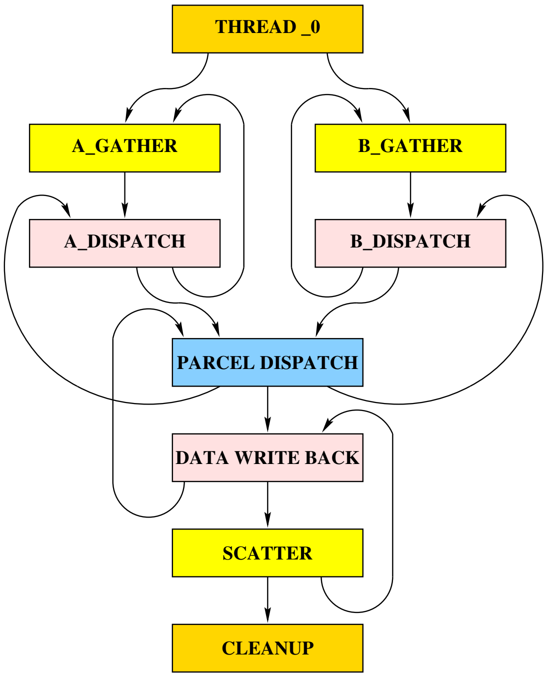

Figure 7: Dependences among threads in our implementation of the block matrix multiply algorithm for HTMT-Threaded-C.

The dispatch of data to the CRAM region is performed through the invocation of the `DISPATCH_DATA()` runtime primitive. DISPATCH_DATA reserves the space in CRAM to store the block to be dispatched, transfers the data to the reserved space, and updates address conversion tables. If space was previously reserved for that block in CRAM, DISPATCH_DATA will use the already reserved space.

The THREAD_PARCEL_DISPATCH calls the primitive DISPATCH_PARCEL to create a parcel with the function code and the parameters of the function `mult_subblock` and then transfers this parcel to the cryogenic area so that the function can be executed by the SCP. The address of the slot DATA_WRITE_BACK is passed as a parameter to the `mult_subblock` so that it can be synchronized at the end of the computation in the cryogenic area.

If this is the end of a row of the matrix B, the THREAD_DATA_WRITE_BACK synchronizes the slot of the THREAD_A_DISPATCH to enable a new block of A to percolate to the CRAM area. Otherwise, an extra synchronization signal is generated to the PARCEL_DISPATCH sync slot to allow it to spawn its thread after a new percolation of a block of B that is started by the synchronization of the sync slot of THREAD_B_DISPATCH.

The THREAD_DATA_WRITE_BACK also turns on the flag `terminate` if the data written back corresponds to the last multiplication that has to be performed. This flag is used in the THREAD_SCATTER to decide whether the THREAD_CLEAN_UP should be spawned, or if a new DATA_WRITE_BACK needs to be synchronized.

The THREAD_SCATTER calls the function `reverse_corner_turn()` to add the freshly produced results that are stored in the `BC` buffer in SRAM to the corresponding sub-block of C. THREAD_CLEAN_UP releases the space used by `BA`, `BB`, and `BC` in CRAM and in SRAM and synchronizes the slot FINISHED, signaling that the parcel threaded function has concluded the computation.

The PARCEL THREADED function `mult_subblock()` multiplies a sub-block of A by a sub-block of B and adds the result to a sub-block of matrix C. Observe that the data transfered from the SRAM into the CRAM area is accessed by the parcel threaded function using the same symbols (`BA`, `BB`, and `BC`) that were used when this data was stored in SRAM. The runtime system will provide an automatic mechanism to translate the addresses of the dispatched data to its physical location in CRAM. This mechanism in the program execution model allows for versatility in the implementation of the HTMT machine. In a future implementation of HTMT a block of data dispatched from the SRAM to the CRAM may not be necessarily stored as a contiguous block in the CRAM as long as the runtime system provides the appropriate address translation mechanisms.

8 Performance Measurements

The main goal of this preliminary implementation of a HTMT emulator is to provide a platform for obtaining initial estimates for the performance of a future implementation of HTMT. This document introduces a minimal HTMT-Threaded-C language that allows for the programming of key applications. This language leads to the implementation of an emulator of the HTMT on the Earth architecture. The main advantage of this approach is that it builds on the extensive research efforts of other groups, such as the Delaware/McGill group development and implementation of the EARTH architecture, the Beowulf project, the MANNA and the SP2 machines.

However, it is difficult to obtain direct performance measurements from an emulator because it does not have the actual hardware of the machine that we want to investigate. Even a carefully designed simulator would depend on a clearly defined model for the characteristics of the different networks in the architecture as well as the execution time of the instructions in the processors to yield a close performance estimate.

Nonetheless, the simple emulator proposed in this document will yield runtime statistics that can be used by the application programmer to estimate the actual performance of the machine. Among the runtime statistics that can be provided by this emulator, we list:

- number of data dispatch;
- average size of data blocks dispatched;
- number of parcels assembled and dispatched for each parcel threaded function;
- number of synchronization signals exchanged between the SCP and the SRAM-PIM;

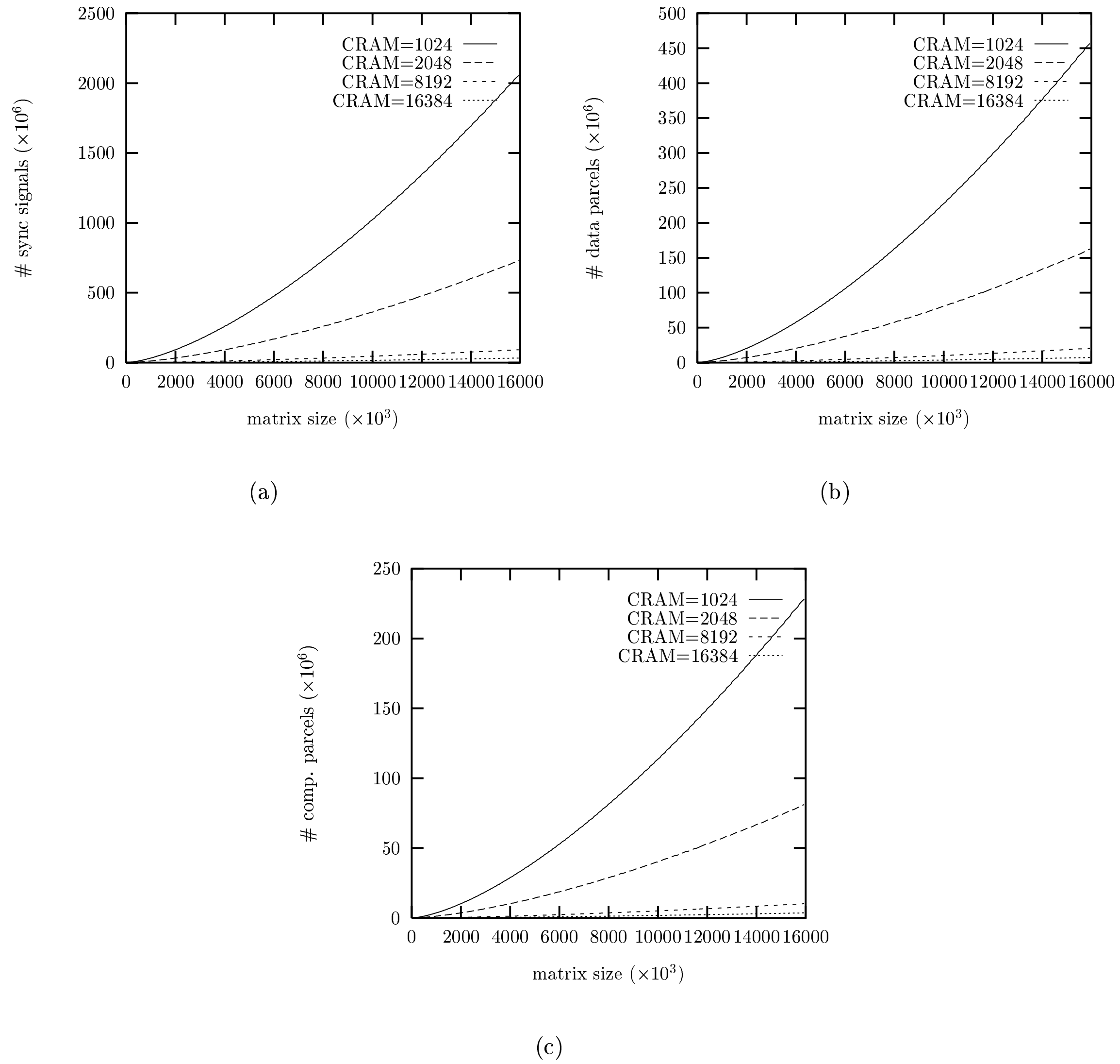

Figure 8: Statistics for our implementation of the block matrix multiplication algorithm in HTMT-Threaded-C. All three graphs are functions of the size of the square matrices multiplied and the amount of CRAM memory available in the machine. (a) Number of syncrhonization signals generated; (b) Number of data parcels sent to and from the cryogenic area; (c) Number of parcel threaded functions dispatched to the cryogenic area.

- Amount of memory reserved in CRAM;
- Amount of memory reserved in SRAM;

As we write this document, we are in the process of defining a reasonable set of runtime statistics that is of interest for an application programmer. This runtime statistics will be always collected internally in the emulator and will be built in the language primitives. They will be printed out when required by the programmer through a yet to be defined primitive. [We would appreciate to receiving contributions from the application scientists in the HTMT project to help us define a meaningful set of runtime statistics to be implemented in the emulator].

In regular applications, some runtime statistics can be obtained from an theoretical analysis of the HTMT-Threaded-C code. For instance, Figure 8 displays runtime statistics obtained this way. To make this analysis easier we consider only the case in which we are multiplying square matrices. All graphs in the figure are functions of the size of the matrices to be multiplied, and the amount of CRAM memory available in the cryogenic area. Graph (a) displays the number of synchronization signals generated during the execution of the program. These synchronization signals include the synchronization within the SRAM-PIM and with the cryogenic area. Graph (b) displays the number of data parcels sent to and from the cryogenic area. This number includes the sub-blocs of A, B, and C exchanged between the SRAM-PIM and the cryogenic area. Graph (c) displays the number of parcel threaded functions dispathed to the cryogenic area. This is the number of times that the function `mult_subblock()` is called.

The curves presented in Figure 8 illustrate that the amount of communication necessary between the cryogenic region and the SRAM-PIM is related to the amount of the CRAM available for storage of data. The amount of CRAM in the labels of the curves in the figures is only the space available for allocation for data storage. Care should be exercised when associating these runtime statistics with actual performance. For instance when the CRAM is larger, the number of data parcels transmitted and the number of parcel threaded functions executed is smaller. However the size of the data parcels is larger and the amount of coumputation performed at the SCP for each parcel threaded function invocation is also larger.

With the runtime statistics provided by the HTMT emulator and estimates of parameters of the HTMT architecture such as data transmission rate between the distinct memory regions, memory capacities, processing speed in the SCP, SRAM-PIM, and DRAM-PIM, it will be possible to arrive at performance estimates for key applications in the HTMT machine.

9 Future Work

This document introduced a first definition of a HTMT-Threaded-C language. This language was designed with a number of important constraints. Taking advantage of the experience and effort already invested in the design of Threaded-C which is operational in a number of important parallel platforms allowed us to deliver a preliminary emulator for application experiments in a short period of time.

In our definition of HTMT-Threaded-C we tried to make minimal modifications to the original Threaded-C, and yet capture the complexity of the HTMT percolation model. This minimal implementation is already working at the University of Delaware as the matrix multiplication example appended to this document evidences. In the near future, with the input from other research groups, we must revisit this language definition to try to capture nuances of the architecture that we might have overlooked in this first definition. Also we will work on the definition of runtime system statistics measurements and on the inclusion of such statistics in our HTMT primitives.

The process of defining HTMT-Threaded-C and implementing the first example in the language reinforced our feeling that it is very important to write a program that runs to completion and produces correct results in an emulation platform in order to understand an architecture model. [We encourage HTMT application scientists who are experienced with Threaded-C program to try their hands in this first version of HTMT-Threaded-C and provide suggestions for ways to improve this language specification. Please send email to *amaral@capsl.udel.edu* to receive the files necessary to transform HTMT-Threaded-C code into standard Threaded-C code].

At the Princeton HTMT Workshop we had the opportunity to discuss the propositions presented in this technical memo with many of the application scientists participating in the HTMT project. We collect the following list of suggestions for extensions and improvements to this original proposal:

- Include the DRAM region in the model;
- Provide the illusion of a single address space in each memory region (CRAM, SRAM, DRAM). Considering that a portion of each memory region will be emulated in each node of the EARTH system, this will require some new HTMT-Threaded-C primitives with underlying communication and synchronization mechanisms.
- Provide dispatching of multiple copies of the same parcel threaded function with a single statement.
- Create a primitive that allows the programmer to find out how much memory is still available for heap allocation in each memory region.
- Add a mechanism to allow the programmer to deal with the situation in which a memory reservation or allocation fails.
- Record and print profile information, including run time statistics.
- Provide some limited printf based debugging facility.
- Offer at least one higher level synchronization mechanism (lock/unlock, acquire/release, semaphore) to facilitate the sharing of information in the shared memory machine.

Acknowledgments

The writing of this document derives from fruitful discussions with a number of important researchers and scientists in the HTMT community, including Burton Smith, Peter Kogge, Vincent Freeh, Nikos Chrisochoide, Larry Bergman, Loring Cramer, John Salmon, Rick Stevens, and Thomas Sterling. Members of the Delaware team who actively participated in the discussions and helped the authors with our understanding of the model include Gerd Heber, Ruppa

Thulasiram, Thomas Geiger, and Cheng Li. The help of Zach Ruiz with the code debugging of the program example is greatly appreciated. We also thank for the support of all the members of CAPSL, especially Patrick O'neill that helped with the designing of slides. Thanks to Peggy Gao for the great help with the creation of electronic presentations. Sean Ryan and Ruppa Thulasiram offered valuable comments after their careful review of the document.

HTMT-Threaded-C builds on the impressive earlier effort towards the EARTH project by the ACAPS group at McGill University, continued by CAPSL at University of Delaware. Threaded-C is based on the EARTH multithreaded execution model, which was proposed by Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao [12]. Herbert Hum lead the team that developed the first implementation of EARTH. The original definition of the EARTH operations was used as the basis for Threaded-C, which was designed and implemented mainly by Olivier Maquelin. He oversaw the development of both the EARTH runtime system and Threaded-C compiler.

ACAPS and CAPSL members who have contributed to the EARTH runtime system include Haiying Cai, Prasad Kakulavarapu, Cheng Li, Andres Marquez, Shashank Nemawarkar, and Kevin Theobald. Xinan Tang ported the Threaded-C compiler to the Beowulf system and currently maintains the portable version of the Threaded-C compiler. Many people have contributed to the the EARTH benchmark suite, including Nasser Elmasri, Gerd Heber, Alberto Jimenez, Olivier Maquelin, Xinan Tang, Kevin Theobald, Parimala Thulasiraman, Xinmin Tian, and Yingchun Zhu. Ruppa Thulasiram has done a lot of work to help edit this document. Finally, we would like to thank Laurie J. Hendren and her team at ACAPS for many valuable contributions and suggestions.

This work was performed for the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the National Security Agency (NSA) through an agreement with the National Aeronautics and Space Administration.

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government, the (name of the contractor), or the Jet Propulsion Laboratory, California Institute of Technology. The initial EARTH work was partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

List of Abbreviations and Acronyms

API	Application Program Interface
CNet	Cryogenic Network
CRAM	Cryogenic Random Access Memory
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
HTMT	Hybrid Technology MultiThreaded Architecture

PAD	Parcel Assembly and Disassembly Module
PDD	Parcel Dispatcher and Dispenser Module
PIM	Processor In Memory
PIT	Parcel Invocation and Termination Module
PXM	Program Execution Model
RTS	RunTime System
SPELL	SuperConductor Processing ELement
SRAM	Static Random Access Memory
SPTR	Synchronization Slot Pointer

References

[1] ACM SIGARCH and IEEE Computer Society. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, California, May 17–19, 1993. *Computer Architecture News,* 21(2), May 1993.

[2] ACM SIGARCH and IEEE Computer Society. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 22–24, 1995. *Computer Architecture News,* 23(2), May 1995.

[3] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* [3], pages 2–13. *Computer Architecture News,* 23(2), May 1995.

[4] Derek Chiou, Boon S. Ang, Robert Greiner, Arvind, James C. Hoe, Michael J. Beckerle, James E. Hicks, and Andy Boughton. StarT-NG: Delivering seamless parallel computing. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *Proceedings of the First International EURO-PAR Conference*, number 966 in Lecture Notes in Computer Science, pages 101–116, Stockholm, Sweden, August 29–31, 1995. Springer-Verlag.

[5] G. R. Gao, K. B. Theobald, A. Marquez, and T. Sterling. The htmt program execution model. CAPSL Technical Memo 09, University of Delaware, Newark, Delaware, jul 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[6] Guang R. Gao and Vivek Sarkar. Location consistency – a new memory model and cache consistency protocol. CAPSL Technical Memo 16, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, February 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[7] Guang R. Gao, Kevin B. Theobald, Andrés Márquez, and Thomas Sterling. The HTMT program execution model. CAPSL Technical Memo 09, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, July 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[8] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, Mariland, 3rd edition, 1996.

[9] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 2nd edition, 1996.

[10] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Xinan Tang, Guang R. Gao, Phil Cupryk, Nasser Elmasri, Laurie J. Hendren, Alberto Jimenez, Shoba Krishnan, Andres Marquez, Shamir Merali, Shashank S. Nemawarkar, Prakash Panangaden, Xun Xue, and Yingchun Zhu. A design study of the EARTH multiprocessor. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 59–68, Limassol, Cyprus, June 27–29, 1995. ACM Press.

[11] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 288–294, Cancún, Mexico, April 26–29, 1994. IEEE Computer Society.

[12] Yuetsu Kodama, Hirohumi Sakane, Mitsuhisa Sato, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi. The EM-X parallel computer: Architecture and basic performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* [3], pages 14–23. *Computer Architecture News,* 23(2), May 1995.

[13] Andrés Márquez, Kevin B. Theobald, Xinan Tang, and Guang R. Gao. A superstrand architecture. CAPSL Technical Memo 14, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, December 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[14] Andrés Márquez, Kevin B. Theobald, Xinan Tang, Thomas Sterling, and Guang R. Gao. A superstrand architecture and its compilation. CAPSL Technical Memo 18, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 1998.

[15] Michael D. Noakes, Deborah A. Wallah, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* [2], pages 224–235. *Computer Architecture News,* 21(2), May 1993.

[16] Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally. Evaluation of mechanisms for fine-grained parallel programs

in the J-Machine and the CM-5. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* [2], pages 302–313. *Computer Architecture News,* 21(2), May 1993.

[17] Kevin B. Theobald, José Nelson Amaral, Gerd Heber, Olivier Maquelin, Xinan Tang, and Guang R. Gao. Overview of the Threaded-C language. CAPSL Technical Memo 19, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[18] Kevin B. Theobald, Guang R. Gao, and Thomas Sterling. Superconducting processors for HTMT: Issues and challenges. CAPSL Technical Memo 15, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, December 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.