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

The HTMT Program Execution Model

Guang R. Gao
Kevin B. Theobald
Andrés Márquez
Thomas Sterling †

CAPSL Technical Memo 09
July 18, 1997

†Jet Propulsion Laboratory, 4800 Oak Grove Dr., Pasadena, California 91109

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

The Hybrid Technology Multi-Threaded (HTMT) Architecture has been proposed to meet the challenges of the design and development of a petaflops-level parallel architecture integrating the combined capabilities of semiconductor, superconductor, and optical technologies. This mix of technologies may yield operational attributes superior to those based solely on semiconductor technology. The HTMT architecture redefines the relationship between memory and processors to provide efficient computation in the presence of drastically disparate cycle times and latencies.

A cornerstone of the HTMT architecture is its program execution model. This paper provides an overview of the current work on the HTMT program execution model conducted at the University of Delaware. It is intended to serve as a base document which different HTMT groups may find useful in their interface with the work at Delaware. The architectural concepts at Delaware are still evolving rapidly, so this paper constitutes a "work in progress" and is subject to change in the future.

This paper contains:

- An overview of the HTMT program execution model including: (1) the thread model, (2) the memory model, and (3) thread percolation.
- An in-depth discussion of the runtime thread model and thread percolation.
- An introduction to the HTMT Operation Set.
- Illustration and analysis of HTMT programming using "peta-kernels."
- An outline of the HTMT runtime system.

Contents

List of Figures

1 Introduction

Major advances in several key technologies, ranging from superconducting processing to optical storage and interconnect, present the opportunity to achieve computing at the scale of 1 petaflops within a decade – far less time than anticipated through the evolution of conventional semiconductor technology. The HTMT (Hybrid Technology Multi-Threaded) architecture is proposed to exploit this opportunity by combining the strengths of these diverse technologies. However, these new technologies are so significantly different from existing technologies that exploiting them effectively will require radical changes in system design. An existing architecture transliterated to the new technologies would suffer due to extreme differences in speed between the superconductor logic and the slowest memory.

Multithreaded architecture techniques hide latency by context switching among concurrent threads. However, while these methods may be effective at hiding latencies in the range of 10 to 100 cycles, a hybrid technology architecture will impose latencies as much as ten thousand times greater. Therefore, a major challenge of the HTMT project will be the development of an innovative multi-level multithreaded program execution and architecture model to manage the latency in a deep memory hierarchy extending from the superconductor processors to the holographic storage.

A program execution model serves as the interface between the system implementors and the compiler writers. The role of the former is to design an efficient implementation of the program execution model. The role of the latter is to translate high-level programs into the machine language corresponding to the program execution model. Thus, the design and development of the HTMT program execution model plays a central role in this proposed project: it provides a basis by which the HTMT architecture can keep all critical resources in the system usefully busy for petaflops-scale applications, while the latency of the multi-level memory hierarchy is smoothly and effectively tolerated and managed.

This paper provides an overview of the current work on the HTMT program execution model conducted at the University of Delaware. It summarizes major advances during the period between the first HTMT workshop held in Pasadena in February, 1997 and the first HTMT Investigators Workshop to be held at the University of Delaware in July, 1997. It should be treated as an interim report of work in progress. Eventually, it is intended to be used as a base document for the various HTMT groups.

We first present an overview of the HTMT program execution model. A threaded program is represented as a *Multithreaded Program Graph* (MPG). The proposed HTMT program execution model is based on an operational model (semantics) of the corresponding MPG. It is beyond the scope of this paper to have a formal definition of the MPG and its operational semantics. Instead, we present an informal outline of the MPG and how an MPG is executed under the proposed HTMT program execution model. This includes an outline of the HTMT thread model, the memory model and the basic notion of thread percolation.

The proposed program execution model is elaborated with an in-depth discussion of the runtime thread model and thread percolation. In particular, we introduce the notion of "parcels," which are threads designated by the users (programmers or compilers) as objects of percolation. The notion of "strands" is also introduced, where a strand is a thread with very light weight – its initiation does not require allocation of a new frame.

We introduce a first version of the HTMT Operation Set, which defines a low-level programming interface to the HTMT program execution and architecture model. We focus on HTMT thread-level primitives, e.g., thread-level control, synchronization and data movement operations which are specific to the HTMT program execution model. Thread percolation operations are highlighted. Many of these thread-level operations may be implemented by one or a number of machine instructions, and some may be implemented through runtime system functions. The HTMT operation set is presented as an extension of Threaded-C, language used successfully to describe the EARTH program execution model. We expect that extensive experience gained by our group from the existing software platform for Threaded-C will be of great value in the development of the HTMT low-level software system.

To illustrate the proposed HTMT program execution model and operation set, we study the mapping of five "peta-kernels" as a case study. We include samples of HTMT code and a performance assessment of the peta-kernels. The assessment is based on the assumptions of the HTMT architectural parameters in the year 2004 (see Section 2.1).

We have also included a brief discussion of the HTMT runtime system. Our focus is on the percolation support, and we include a discussion of the runtime representation of parcels, and the data-gathering (e.g., runtime tiling when needed) and movement operations required for percolating a parcel.

The rest of this paper is organized as follows. Section 2 briefly outline the basic HTMT architecture and its memory hierarchy. Sections 3 and 4 present the HTMT thread (activation) model and memory model, respectively. The HTMT operation set is introduced in Section 5. Peta-kernels are used to illustrate the operation of the HTMT and evaluate its performance in Section 6. Section 7 is a brief introduction to the HTMT runtime support system. The last section summarizes our work.

2 HTMT Architecture

The HTMT architecture combines semiconductor, optical, and superconductor technologies in a single-system structure to achieve performance superior to that which could be attained in the same time and comparable cost using semiconductor devices alone. The HTMT architecture (see Figure 1) is essentially a shared-memory NUMA architecture employing liquid-helium-cooled superconducting processors (SCPs) and data buffers (cryo-RAM or CRAM), liquid-nitrogen-cooled SRAM semiconductor buffers, semiconductor DRAM main memory, and optical

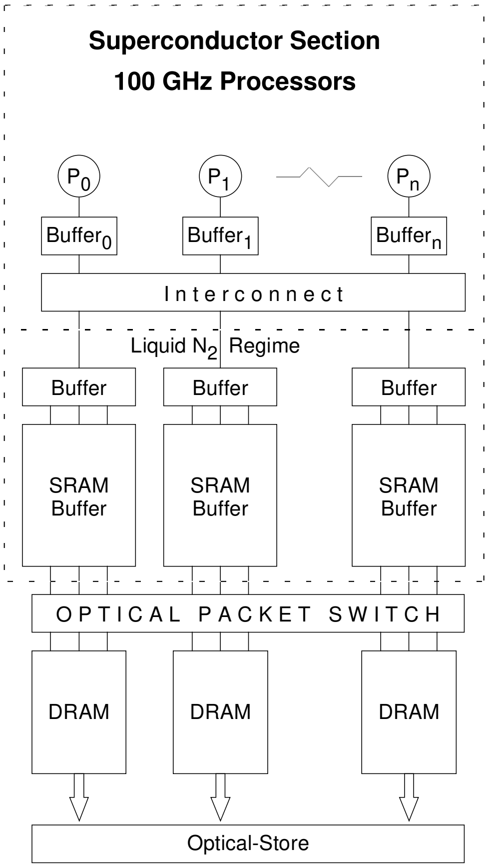

Figure 1: HTMT Architecture

holographic storage. There will also be a superconducting interconnect in the processor section and an optical packet switch network between the SRAM buffers and the highly-interleaved DRAM main memory. The architecture to be studied is anticipated to be feasible within the next ten years.

The principal advantage of the hybrid technology approach is that the different strengths of these technologies can be combined to produce a superior aggregate system. However, making this combination work effectively will be HTMT's biggest challenge. In particular, latency management throughout the system hierarchy will be the most critical issue. Latency will be a problem even within a processor, because high clock speeds and relatively low integration levels within the superconducting processors mean that only a small amount of logic and state is reachable within one clock cycle. Latency will become an even greater problem as one moves down the memory hierarchy.

Bandwidth will be another major concern. The memory-processor interfaces of most conventional systems have insufficient bandwidth to match the processor's computing speed; they rely on temporal locality (registers and caches) to compensate. Many of the applications which would need petaflops computing (including most of the peta-kernels in Section 6) have data access patterns that would overwhelm any feasible cache system. In order to keep the superconducting processors running at full speed, it is necessary to have high data throughput between the processors and the memory, and to sustain this same aggregate throughput across all or most of the memory hierarchy.

A multithreaded execution model is adopted as the principal scheduling and latency management mechanism. Even so, conventional multithreading techniques are not enough to handle either such large latencies or such high throughput requirements if only conventional memory systems are used. Therefore, memory-to-memory block moves managed within the memory itself, and an explicitly-addressed memory hierarchy, with small, fast memory buffers at the top, will be added to the system to improve throughput and to eliminate or substantially reduce unpredictable memory latencies.

The difference in clock speeds between the superconducting region and the highest level of semiconductor memory is expected to be around a factor of 100. Generating enough data from SRAM to feed the superconducting processors will require highly-parallel memory accesses. While this will be straightforward for basic vector computations such as SAXPY (see Section 6.1), moving less-regular data structures such as trees (see Section 6.5) will require sophisticated hardware support. Peter Kogge (at Notre Dame) has developed a sophisticated Processor-In-Memory (PIM) technology that combines memory and logic on single chips providing high memory access bandwidth and low chip count (as demonstrated by the Execube). PIM technology will be extended and used in one or more levels of the memory hierarchy to support movement of data between different levels of the hierarchy while satisfying both latency and bandwidth requirements.

2.1 Architecture Assumptions

Our performance projections later in this paper are based primarily on what we expect will be feasible for the year 2004. The clock speeds and memory capacities as projected in our original proposal [11] serve as a basis. All semiconductor performance projections are taken from industry forecasts [10].

We expect the SCP and CRAM to support:

- 128-bit operands;
- Stream operations that can move 2 operands into and 1 operand out of the SCP every clock cycle;
- Multi-ported, pipelined CRAM.

We have made no assumptions about the performance of the superconducting interconnection network, so we have put more effort into partitioning and mapping data over the SRAM

PIMs to encourage PIM/SCP pairing. This effort would reduce the demands on the SCP-PIM interconnect for uniform delay.

We expect PIM to support:

- 128-bit operands;
- N_2-cooled SRAM with average access time $\leq$4ns and large effective bandwidth (enhanced through pipelining and interleaving);
- Address generation and data access matched with SCP demands;
- Multithreading support on chip;
- Support for stream operations.

3 HTMT Thread Model

One of the key ideas in the HTMT program execution model is the partitioning of a program into threads. A thread is a group of instructions scheduled as a unit in order to amortize synchronization and memory access overheads. This section describes the HTMT thread model and its operational behavior. At this early stage in the HTMT project, the model is necessarily fairly abstract and incomplete; it will become more well-defined as research progresses and preliminary simulations are performed.

3.1 Multithreaded Program Graphs

The structure of an HTMT machine-language program can be represented by a directed acyclic graph called a *Multithreaded Program Graph* (or MPG). In an MPG, each node is a thread, and a direct arc between two nodes represents a precedence relation between them. Such a relation may be due to a data or control dependence, or may simply represent an explicit control flow relation between the two nodes.

Because the HTMT architecture has a deep memory hierarchy in which lower levels trade latency for bandwidth, thread sizes will need to be matched to each level in the hierarchy. Therefore, programs are divided into threads at different levels. For example, a thread may be generated by simply grouping a sequence of instructions, such as one or several iterations of a loop. Threading can also occur at the function invocation level, or even at a higher level (clusters of function invocations). As a result, an MPG of a program can be represented hierarchically, i.e., a node in an MPG can itself be an MPG subgraph.

Note that an MPG is not a dataflow graph, although its conception has been strongly influenced by our early work on the McGill Dataflow Signal Graph model [2]. Being directly executable (see its operational semantics in the next section) and inherently parallel, an MPG is neither a control flow graph nor a program dependence graph.

3.2 The Operational Semantics of an HTMT MPG

As in EARTH (and similar models),[1] threads are not ordered sequentially, but are enabled according to data and control dependences.

A necessary condition for a thread to become enabled is that all data required by the thread have been produced, and all control dependences are satisfied. This is ensured by the operational semantics of an MPG: a node in an MPG becomes enabled if all the precedence constraints (as represented by the input arcs of the node) are satisfied.

The satisfying of these constraints can be made explicit using synchronization commands that are part of the HTMT instruction set, as described in the next subsection.

The above enabling condition is sufficient for EARTH (and similar models known to us): a thread will become "logically enabled" once the precedence conditions are met. However, for the HTMT, this is not enough! The HTMT program execution model must be able to solve the problems created by the huge gap between the ultrafast processor register access time and the memory latencies at various levels. One unique feature of the HTMT thread activation model is the introduction of an additional necessary condition for a thread to become enabled: it must also meet all "locality requirements" for efficient execution. We will define such locality requirements formally in the proposed project. Intuitively, all data and code referenced by the thread should become local before a thread can begin execution – we call this "physically enabled." Here "local" means that the corresponding code and data of the candidate thread are resident in the same level of memory hierarchy where it is to be enabled.

3.3 Contexts: Threads and Strands

Every thread will have some context associated with it. The context of a thread will certainly include some registers for holding intermediate data, but will also include some memory so that values can be passed to and from the thread. This memory will typically consist of a *frame* of local variables, similar to a stack frame. However, since many different threads, each with its own context, can be active simultaneously, these frames cannot be arranged in a conventional stack-like structure. As in some other multithreaded models including EARTH [4, 5, 9], the frames in our model are dynamically allocated from a heap, and are typically arranged in a non-linear structure (such as a tree). Each frame represents the instantiation of one function, and contains the local state of that instantiation, including local variables and synchronization slots.

Often, several threads are closely related, e.g., they are part of the same function call, and hence share their data with each other. Such a group of threads will share a common frame,

[1] For terminology and concepts, readers may find it useful to read articles from the Efficient Architecture for Running Threads (EARTH) multiprocessor studies [4, 5, 6, 7, 8].

and use the frame's local variables for passing data between them. We therefore distinguish two kinds of threads. The first type corresponds to a function/procedure invocation – called a *threaded function invocation* – and is always associated with a frame in the heap. The body of a threaded function invocation can itself contain several threads, which do not require the allocation of separate frames, but instead share the same frame as the threaded function invocation. For convenience, we will sometime use the term *strands* to refer to these *frame-less* threads.

3.4 Thread Activation and Scheduling

An enabled thread can become *ready* when needed execution resources (hardware registers, CPUs, etc.) become available, and can be *activated* for execution by a runtime thread scheduler under certain policies (e.g., FIFO priority). Scheduling issues are a topic to be addressed both by the HTMT program execution model and the architecture design. One scheduling policy is to schedule certain threads "atomically," i.e., a thread, once its execution begins, will hold onto all resources, both allocated CPU resources and the register context, without suspension. Note that atomicity is an orthogonal issue from the issue of sharing CPUs with other threads, e.g., thread interleaving in execution pipelines, That is, multiple atomically-scheduled threads may share the same execution pipeline via interleaving. Such interleaving implies several threads can be simultaneously in activation, each holding its own register context.

Disallowing thread suspension and resumption eliminates the associated overheads of dynamically saving and restoring contexts at runtime. This is particularly helpful since such suspension and resumption of threads is often due to external events, such as cache misses, and, therefore, is mostly not statically analyzable or predictable at compile time. Hence it is difficult for a compiler to optimize the amount of information to be saved and restored. We expect that atomic scheduling would be attractive for thread execution at the superconducting processor level, where efficiency is a foremost concern. Obviously, in this case, a thread should not hold execution resources waiting for non-local memory references. That is the purpose of introducing the "locality requirement" as an enabling condition.

Primitive thread operations include thread initiation and termination, thread synchronization operations, shared-memory operations (loads and stores), block data transfers, etc. Threads synchronize and communicate with each other at different levels through shared memory, which is the topic of the next section. HTMT operations will be covered in Section 5.

Functions are invoked directly by the program, so it is the responsibility of the programmer or compiler to determine that a function is ready to be executed. Strands, on the other hand, are synchronized automatically, using an EARTH-like synchronization and enabling scheme [4, 5, 6]. Such a thread must reside entirely within one function (procedure). This is done using *synchronization signals* and *synchronization slots.* Each such strand is "guarded" by a synchronization counter which keeps track of the number of synchronization signals it needs to receive before it can become enabled. A synchronization signal is used by the producer of a datum to tell consumers of that datum that it is ready. A sync slot is used to coordinate the incoming sync signals, so that a consumer knows when all required data is ready. Each sync signal is directed to a specific sync slot.

A sync slot contains three fields: a *reset count,* a *sync count*, and a strand id. The reset count, which is constant, gives the total number of sync signals required to enable a thread. The sync count indicates the *current* number of sync signals needed to enable the thread. When a sync signal is received by the strand, the sync count in the corresponding sync slot is decremented. If the count reaches 0, it is set back to the reset count, and the strand becomes enabled.

We use a strand id to represent a *specific* strand within *one* instantiation of a function; therefore, a strand id is a pair <*fp, ip*>, consisting of a frame pointer fp pointing to the frame of the associated function and an instruction pointer (i.e., the PC) – a pointer to the first instruction of the strand. The state of a threaded function invocation is defined by the collection of program counters of its component strands and a unique frame pointer.

3.5 Thread Percolation

The HTMT threading model, as presented so far, does not differ substantially from the EARTH threading model on which it is based, except for the introduction of a "physical locality" requirement. However, EARTH and similar multithreaded systems use conventional memory systems. HTMT, on the other hand, will use sophisticated memory management hardware to move data throughout the memory hierarchy and satisfy the latency and bandwidth needs of the high-speed processors. The behavior of this hardware should be incorporated into the HTMT threading model.

Therefore, a new multi-level multithreaded context management strategy, called *thread percolation*, is currently under study. This methodology and the underlying mechanisms it requires can be considered to be a combination of multithreading with dynamic prefetching of coarse-grain contexts. Prefetching in the past has concentrated on moving blocks of *data* within the memory hierarchy. The thread percolation approach manages contexts instead of just contiguous blocks of data, where a context includes data, program instructions, and control state. The objective of this multi-level scheme is to migrate contexts of varying granularity automatically through the memory hierarchy so that the superconducting processors (SCPs) and buffer memories always have access to a supply of executable contexts at the appropriate levels without incurring any unexpected latencies. This is essential to support the physical "locality requirements" discussed earlier.

The dynamics of the multi-level context management strategy cause "hot" contexts to move up towards the processors and "cool" contexts to drift to lower, slower, higher-capacity memory.[2] In their early work, Gao and his colleagues proposed the "register-cache" mechanism to migrate vector-like data [3] and the "register-use cache" mechanism to migrate contexts [6], both under multithreaded program execution models. The context percolation can be considered a generalization and extension of these earlier schemes, where percolation is now performed across the entire memory system and where the size of contexts being percolated increases as one moves down the hierarchy.

Context percolation will be controlled by Processor-In-Memory (PIM) technology (see Section 2) tailored for use at the appropriate levels of the HTMT memory hierarchy. To maximize the generality of thread percolation, the PIM will include sophisticated in-memory operations, such as pointer tracing and compound atomic operations for synchronization and mutual exclusion. In order to match the bandwidth requirements of the superconducting processors, these operations will have to be highly parallelized. Fortunately, given the levels of memory integration expected at the time the HTMT is first implemented, it will be possible to put many PIM processors in each memory chip.

In order to prevent latencies in the superconducting region, the PIM system must not only be able to move contexts to and from that region; it must also be able to *anticipate* the data needs of the superconducting processors. To achieve this, it will be necessary for the PIM system to *control* much of the execution of a program. The HTMT system, therefore, will employ a distributed control scheme in which the PIM-based levels of the hierarchy make coarse-grained decisions about what computations to perform next, and set up this data for the superconducting processors, while the latter make fine-grained decisions and perform the bulk of the computations.

3.5.1 S-Threads and P-Threads

We use the terms *S-threads* and *P-threads* to refer to threads executing in the SCPs and the PIM, respectively. Under the HTMT thread percolation model, an S-thread spends its entire lifetime in the SCPs. Instructions within an S-thread may access the register set and CRAM (typically the frame allocated for that thread, the code for that thread, and any shared global data structures). It does not directly address the global memory (the SRAMs, DRAMs and the optical storage). A P-thread, on the other hand, will run on the PIM processors, where it can access all levels of memory, including CRAM, as well as local PIM registers.

One unique feature of an S-thread is that its memory load and store instructions use only "reduced memory addresses," i.e., short addresses for addressing CRAM. On the other hand, memory access instructions in P-threads may use the full global memory addresses. From the view of an S-thread, CRAM is explicitly and directly addressable, and the allocation of CRAM locations is subject to the programmer or compiler control (optimization).

[2] Ironically, as contexts get "hotter," they move into progressively *colder* regions of the machine!

Within each execution domain, S-threads and P-threads function much as threads under the EARTH program execution model as previously described. That is, the SCPs will support a set of basic S-thread operations: thread creation, termination, synchronization and communication. Similarly, PIM will support a set of analogous P-thread operations.

3.5.2 Parcels

To complete our model of thread percolation, we introduce *parcels*. When a PIM processor decides to execute a function on the SCPs, it will need to get the SCPs ready to execute the threads, mainly by allocating and initializing data in the CRAM. This initialization usually involves copying data from lower levels of memory to CRAM. Sometimes this is a straightforward block copy, while other times, the data may need some form of translation (such as changing pointers).

An *S-parcel* encapsulates both the S-thread and the setup code that runs in the PIM. This includes code to allocate a frame in the CRAM, to download the S-thread code to the CRAM (if not already there), and to transfer data to the CRAM. Most of these operations are straightforward, and may be handled by library functions. However, some data percolation may be complex and require special code written by the user or compiler.

4 Memory Model

The thesis of the HTMT memory model is that the function of a memory system is to support correct execution of programs by the computer system of which it is a part. Programmers write (or should write) application programs assuming some program execution model. A computer system, with its memory subsystem, must be a correct implementation of the program execution model. This is the thesis taken to define the HTMT memory model. That is, the HTMT memory must support the correct and efficient execution of programs under the HTMT program execution model. However, beyond this, we are not to be bound to implement any "global memory coherence" scheme.

The HTMT memory model description below includes an outline of the memory hierarchy and the model for sharing data among processors.

4.1 HTMT Memory Hierarchy

Physically, the HTMT memory hierarchy resembles the hierarchy of a conventional system. Registers, on-chip caches, second-level caches, main memory (DRAM) and secondary storage (disks) are replaced by superconducting registers, superconducting memory, SRAM, DRAM and optical memory, respectively. However, the severe differences in latency in the HTMT system make it imperative that there be no "surprises" caused by unpredictable cache misses, especially at the higher levels of the hierarchy. Even a miniscule cache miss rate can cause unacceptable latency penalties when multiplied by the very long latencies involved in HTMT. The locality requirement in the enabling conditions discussed earlier mandate that when a thread becomes enabled, the data and code (the necessary "contexts" for the thread) to be referenced must be both logically ready and *physically local*.

Thus, while conventional architectures present to the programmer the appearance of a 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. Each level of memory will be an explicit "buffer" of the next level rather than a cache. Superconducting memory is a buffer of the SRAM, which is itself a buffer of DRAM, and so on. Unlike traditional caches, these buffers are directly addressable. Buffer storage allocation and data movement at each level can be fully or partially under the programmer's (or compiler's) control. That is, the data unit and size of 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.

This unconventional organization is most important at the level of the cryo-memory (CRAM). A superconducting processor may be able to read a local CRAM location in 10 clock cycles, but the latency for reading an SRAM location will probably exceed 500 cycles. Therefore, superconducting processors will be able to access CRAM, but not SRAM. It will be up to the PIM in the SRAM to percolate data to and from the CRAM. Therefore, the PIM must be able to access both CRAM and SRAM. However, there will be no automatic mapping between the two; any mapping between them will be under PIM control.

Similarly, all lower levels of the hierarchy (SRAM, DRAM and HRAM) will have separate address spaces. Explicit addressing, rather than a fixed scheme, provides more flexibility for optimizing percolation between memory levels. However, the differences between SRAM and HRAM are comparable to the differences between second-level caches and disks on current systems; therefore, a conventional single-process virtual memory organization (a large, uniform address space as opposed to three separate spaces) may adequately serve most applications. It would be desirable in these cases for the programmer or compiler to be able to use standard virtual memory over all or part of the hierarchy and dispense with the explicit memory management.

Different addressing schemes may be employed at different levels of the hierarchy. For instance, frame-relative addressing may be used at the thread level. Translations will be performed as contexts are moved up and down the memory hierarchy to avoid performing address translations while a thread is being executed. Multi-process virtual memory is not an inherent part of the HTMT program execution model, but may be provided at the lower levels of the hierarchy in order to support basic multitasking and multiuser functions. We leave this as the subject of another discussion.

4.2 HTMT Memory Sharing

The HTMT memory model supports global shared-memory in a manner quite different from what is in conventional NUMA memory systems. In conventional shared cache-based multiprocessors (e.g., CC-NUMA or COMA), memory consistency models and related cache-coherence schemes are crucial issues in designing the memory models. For HTMT, the need to ensure locality and avoid cache misses means that memory sharing must be less transparent than in conventional systems. The HTMT memory model will provide the programmers/compilers with full and explicit control over the consistency of copies of data in the memory hierarchy. 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). In fact, maintaining memory coherence is not a center requirement in designing our memory model and a user program should not pay any penalty for such coherence, unless explicit requested by the program itself.

The HTMT model will also allow direct transfer of data (e.g., message passing). Many applications can be programmed quite naturally using a distributed memory paradigm. Using such a paradigm also eliminates the need to control memory coherence.

5 First Version of the HTMT Operation Set: A Programming Interface for the HTMT Execution Model

In this section, we present a first version of the HTMT "operation set" which defines a low-level programming interface to the HTMT program execution and architecture model. The definition of the operation set will provide a programming interface to the HTMT program execution model. It will be useful for potential HTMT users and compiler writers to experiment with their applications. This definition is preliminary and has to be assessed through simulation and experimentation with the HTMT model.

We choose the term "operation" instead of "instruction" here because we are focusing on HTMT thread-level primitives, e.g., thread-level control, synchronization and data movement operations which are specific to the HTMT program execution model. Most of these primitives will be mapped onto single machine instructions or small fragments of machine code, while others may be implemented through runtime system functions. The details of such a mapping are best postponed until the hardware architecture design itself is more firm.

Our HTMT operation set is based on the **EARTH** model [4, 5, 6]. The operations can be divided into three classes, based on our previous work:

1. Normal operations not special to multithreading.

2. Multithreading support operations based on EARTH.

3. Operations to support features unique to HTMT.

We concentrate on the last class and refer readers to other publications for the other two.

5.1 Common Operations

Most operations executed by HTMT threads will be normal RISC operations: ALU ops, branches, normal load/store operations, etc. We expect that these operations may be based on existing RISC instruction sets. However, two factors will affect what the final instruction sets contain. First, the hardware underlying HTMT processors is so radically different from existing technology (particularly the SCPs), that their special properties may dictate different instruction mixes. Second, instruction set designs will be tailored to the needs of specific regions of the system. For example, since PIM processors will mostly be percolating threads, it may be best to leave out floating-point arithmetic while providing extra support for pointer operations.

5.2 Basic Multithreading Primitives

The operations that support basic multithreading, particularly in the superconducting region (S-threads), are based on our EARTH model. These operations include operations to do the following:

1. Invoke a threaded function;
2. Initialize and manipulate sync slots;
3. Send synchronization commands to sync slots;
4. Atomically send data and signal a sync slot;
5. Request remote data using split-phase synchronization.

The general meanings of these categories are described with the EARTH model [4, 5, 6]. The operations themselves are likely to be similar to the corresponding EARTH operations, with a syntax similar to Threaded-C [4, 5].

5.3 HTMT Threaded-Level Operations for Thread Percolation

Most operations discussed so far are likely to be similar to EARTH operations. However, new operations must be created for the new features of HTMT, mainly thread percolation. Percolation is supported by the *pragma* constructs **S-PARCELED THREADED** and **END**

S-PARCEL. Code encased between these two keywords is intended by the user to be percolated to the next higher memory level. In the case of the investigated peta-kernels, the memory levels involved are SRAM and CRAM. The advantage of using *pragmas* is that the programmer can write the code in a preliminary attempt without being concerned where the execution will be performed. By later inserting these *pragmas* the user decides where the specific code will be executed. This eases compilation as well.

These *pragmas* are *annotated* with a *name* and an *argument list*. The *argument list* gives hints to the code generator as to which data objects will need to be percolated to allow for "dynamic tiling." Data objects which are used in the percolated code but do not show up in the annotated list are deemed to be either locally created or already resident in the higher memory level. As far as the code is concerned, it is up to the compiler to decide if the code has to be percolated as well or not. The compiler may activate a new instantiation of already percolated code after initializing the new frame and data segments.

For the first three peta-kernels investigated (see Sections 6.1–6.3) this approach is sufficient. Starting with the fourth peta-kernel, though, additional information should be provided to the compiler to enable the percolation code generation. Data objects themselves can be annotated with a *range specifier*. This range specifier provides further hints to the code generator as how the percolated dynamic tiles are to be built. This approach requires a good understanding of the percolation mechanisms by the programmer.

The fifth peta-kernel requires complex tree manipulation, and it is assumed the compiler will be unable to generate the percolation code. The user has to provide the code on the compiler's behalf. The keyword **S-PARCELED PERCOL** is defined for this purpose. Functions can be declared to construct a dynamic tile and percolate it up and destruct the tile. The compiler will call the functions in the percolation process. The kinds of operations that will appear in the percolation code are still being studied at this time, so more details cannot be provided at this point. However, it is probably that the programmer will have to be versed with both the nature of thread percolation and the runtime system support functions.

6 Peta-Kernels

In the early stages of the petaflops project, researchers proposing potential petaflops systems were challenged to show how 5 short "peta-kernels" [11] would be executed in their systems. In this section, we will discuss implementations of the five peta-kernels under the proposed HTMT program execution model. This is used as a case study to evaluate the HTMT operations defined in section 5.

The code is based on *Threaded-C* [5], with modifications appropriate for this architecture. We take this approach at this time while the HTMT operation set is still under development, and focus our discussions on the key issues to be illustrated by these kernels. We assume that thread-level operations in Threaded-C (or a future variant) can be translated by a Threaded-C compiler to the corresponding HTMT threaded-level operations as described in the previous section. The normal C operations will be translated to the HTMT base instruction set much as in the conventional code generation process. The feasibility of this approach has been validated by our experience with the EARTH project, where Threaded-C has been used both for illustration of the EARTH execution model and for a handy thread-level programming interface to the EARTH machine.

In all the kernels below, we assume the HTMT architecture has 1000 processors, corresponding to the HTMT architecture anticipated for the year 2004 (see section 2), except for the second peta-kernel, for which we must assume a more advanced architecture due to the size of the kernel's data set.

6.1 Peta-Kernel 1: SAXPY

The first peta-kernel is an implementation of SAXPY. The Fortran 90 source code is listed below:

```
do i=1, 10**8
    c(i) = pi * a(i) + b(i)
enddo
```

In this example, it is reasonable to assume that a user (a compiler or a programmer) can take advantage of the regularity of the loop structure and the array data access pattern. The code will be structured into S-threads and P-threads on the different levels of the memory hierarchy. Figure 2 lists the Threaded-C code for one SCP/PIM processor pair.

We assume that the computation is evenly partitioned among the 1,000 processors, so that each processor is computing a 100,000-word piece of the result vector. Line 8 computes the offset for each processor; where the processor number is returned by the system variable `NODE_ID`. We assume that the vectors are too large to fit into CRAM, but that they can fit in SRAM. To take advantage of the thread percolation model, the code in each processor is subdivided into 100 S-parcels by the forall construct in line 11, with each parcel being inlined into the *SAXPY* code in lines 13-23 (see Section 3.5.2 for a discussion of S-parcels). The space required for each S-parcel under this partition can fit into CRAM.

The keywords which identify the inline code to be percolated up to the SCP region are **S-PARCELED THREADED** to identify the beginning of the inline code and **END S-PARCEL** for the end. **S-PARCELED THREADED** is annotated with the data objects to be percolated in parenthesis. In our case **(a, b, c)**. Since the access pattern as well as the data structures are regular, we defer to the compiler and runtime system the task to determine

```
/* The following is the code for one SCP/PIM pair */
THREADED SAXPY(quad a[], quad b[], quad c[])
/* declares SAXPY to support the Threaded-C extensions between 'THREADED'
and 'END_FUNCTION' */
{
 int i, parcel_vec, offset;

 offset = (NODE_ID-1) * 100000;
/* NODE_ID identifies the current node */
 parcel_vec = 1000;
 forall i=0 to 99
/* fork and join in parallel on the SCP */
   S-PARCELED THREADED  SAXPY_ST (a, b, c)
/* identifies the code and the corresponding data between 'THREADED S-PARCEL'
and 'END S-PARCEL' to be percolated to the SCP */
   {
     int j;

     for (j=0; j<parcel_vec; j++)
       c[offset+i*parcel_vec+j] =
          pi * a[offset+i*parcel_vec+j] + b[offset+i*parcel_vec+j];
   }
   END S-PARCEL;
END_FUNCTION();
}
```

Figure 2: HTMT SAXPY

the allocation and data movement schemes. It is possible to provide range parameters for each of the data objects, as will be shown with the fourth peta-kernel (Section 6.4).

The runtime descriptor of an S-parcel includes the information required for its percolation, e.g., the amount of CRAM it needs to reserve for the data to be accessed, and the data move operations between the source SRAM locations and the destination CRAM locations to prepare for its activation. We assume the code generator will produce the appropriate S-parcel runtime descriptor for the PIM. This information will be used for processing the S-parcel and percolating the enclosing S-thread. We also assume that the data movement operations to bring arrays **a** and **b** into CRAM are stream-based PIM functions which are able to fetch vectors of array elements from SRAM into CRAM at the full bandwidth of the interconnection network between the superconducting region and the PIMs. For this example, the memory layouts for the arrays **a** and **b** are properly structured, ready for such stream functions.

One important point to note is that the code generator for the S-threads will translate the array accesses in lines 20–21 into CRAM frame accesses. Of course, the particular block (frame) of CRAM lines to be reserved for the S-parcel cannot be known at compile time. However, when the enclosing S-thread is successfully percolated, the corresponding CRAM frame pointer is determined and loaded as part of the thread context. Therefore, the compiler only needs to generate static offsets to the lines within the frame.

The sub-vectors loaded into CRAM should be small enough so that several frames can be allocated simultaneously. This would allow an efficient pipelining of vector frames. For example, three frames could be active simultaneously in an SCP. The SCP would execute the S-thread on the middle frame, producing vector **c**, while at the same time, the PIM is downloading **a** and **b** into the next frame while uploading **c** from the previous frame. With this pipelining, the architecture described in Section 2 can percolate one S-PARCEL from SRAM to CRAM in 10ns and move a previous result back from CRAM to SRAM during this time, while the SCP can work on its frame in the same time period.[3]. The ratio of computation to communication is 1:1. Thus, this computation achieves 100 TFlops total performance.

6.2 Peta-Kernel 2: Sparse Vector Move

The second peta-kernel presents a vector move operation which moves words in a vector with a very large stride. The Fortran 90 source code is as follows:

```
do i=1, 10**8
    b(131*i) = a(131313*i)
enddo
```

If this is indeed a sparse matrix application, it would be possible to store **a** and **b** compactly, using hashed index/value pairs, for instance. However, we assume that the motivation of this peta-kernel is to stress the memory hierarchy and measure the ability to stride over non-compact data structures, so we assume that **a** and **b** are stored as uniform arrays. The architecture described in Section 2 could not hold all of **a**, even in HRAM, since this vector holds over 10 terawords, or 166 terabytes. Only the HTMT architecture expected for 2007 would deliver up to a petabyte of HRAM to accommodate this problem size. Since we don't want to consider disk I/O at this point, we code this problem for the 2007 machine.

At the point where the code (see Figure 3) is activated, we expect the arrays **a** and **b** to reside in HRAM. The work is partitioned uniformly, as with the previous peta-kernel. The code will be executed by PIMs without going to SCPs, as there is no real computation other than

[3] This would require 640 wires to stream data in and 320 wires to stream data out of the SCP at 20GHz/bit assuming that each c[] computation requires 2 Flops. Faster network technology (e.g., optical) could reduce the number of interconnects needed.

```
/* The following is the code for one PIM at DRAM level */
THREADED SVM(quad a[], quad b[])
{
 long i;

 offset = (NODE_ID-1) * 25000;
 for (i=0; i<25000; i++)
   b[offset+131*i] = a[offset+131313*i];
END_FUNCTION();
}
```

Figure 3: HTMT SPARSE VECTOR MOVE

address calculation. (We assume that the compiler will use strength-reduction to compute the indices with successive adds.)

In this example, the read data **a** is linearly scattered in HRAM at an approximate distance of 2 megabytes while the stride of the write data **b** is approximately 2 kilobytes. With an assumed HRAM page size of around 128 KBytes[4] we determine that the read stream might actually skip HRAM pages while the write stream might score several times per page. For this reason and to avoid false sharing we partition the HRAM pages according to page boundaries specified by the **b** vector. For the sake of simplicity we assume that the management of data movement at HRAM page level is done automatically by the runtime system since it would not incur any severe performance penalties. An alternative approach, which we don't consider here, would be to let the users (compilers or programmers) handle explicitly the management.

Note that the code is threaded. We anticipate that there will be other program sections running at the same time, so that the latency of the HRAM-DRAM data movement can be hidden by other parts of the running program. In the above example, the code in each processor is structured into one thread. There are benefits to further partitioning this code into multiple threads in real programs where additional computation on the fetched arrays may be performed.

We anticipate that this kernel presents a worst case scenario of all architectures proposed for petaflops machine studies. Nevertheless, we achieve an average read latency per access of $\sim 10\mu s$ given by the HRAM latency and an average write latency of $\sim 0.4\mu s$ composed of 2 accesses to HRAM and 64 accesses to DRAM. Therefore, the expected peak performance is determined by the read accesses. We do not know at this time how many HRAM CCD readers there will be in HTMT, but if they are in a one-to-one correspondence with SCPs, then the total expected execution time will be $\sim$250ms.

[4] We don't know yet what the pixel density of a CCD will be in 2007. It is reasonable to assume, though, that there will be CCDs which support HDTV.

6.3 Peta-Kernel 3: Random Scatter

The third peta-kernel presents a vector scatter operation. In contrast to the second peta-kernel, it involves data movement with unstructured strides. The Fortran 90 source code is listed below:

```
do i=1, 10**8
    idx(i) = 1 + mod(13131313*i, 10**8)
enddo
do i=1, 10**8
    b(idx(i)) = a(i)
enddo
```

A *scatter index* **idx(i)** is precomputed in the first loop. In contrast to the previous peta-kernel, the source **a** and destination vector **b** are bounded to the much smaller size of 10^8, making it possible to fit **a** and **b** into SRAM (see Section 2). In this example, we assume the source vector **a** has been already percolated from DRAM into SRAM. Therefore **b** will be computed in SRAM. The integer division required by the index calculation may not be available in PIM (without sacrificing other components), so we assume that blocks of indices are computed in the SCP and streamed through in a manner similar to the SAXPY computation in peta-kernel 1. (Alternately, the programmer can write strength-reduction code to generate indices using adds and subtracts.) With the indices computed, the data movement is now only restricted by the average access latencies of **a** and **b**.

The index computation idx(i)-1 maps multiples of 13131313 to the integer ring Z_{10^8}. Since 13131313 and 10^8 are relatively prime, it follows that all mapped indices into the integer ring are non-congruent. Therefore, there are no output dependences and the iterations of the second loop can be executed in parallel. We distribute the task evenly over all 1000 processors and show the code for one processor with the processor number `NODE_ID`.

In Figure 4 we present the code for one SCP/PIM processor pair. Note that the S-parcel in lines 7–14 encapsulates the index computation. Each processor will perform the index calculation of 10^5 elements. The index array idx(i) will be streamed out from the SCP region after the index computation, and will be used subsequently by the load-scatter operations in lines 16–17.

As mentioned above, the average access times to **a** and **b** dominate the execution time. The average latency depends strongly on the memory organization and on the page size. Both parameters are not available at the time this paper is written. We can do an estimation, though. Assuming, **a** will be accessed local to the PIM, the large internal bandwidth will hopefully yield access times which are below 10ps. The assumptions for **b** are less conservative since interchip communication and large strides might waste large amounts of bandwidth. Nevertheless, if we assume that the average access time for **b** is 10ps (no conflicts assumed), the total performance would be 100 tera-ops per second.

```
/* The following is the code for one SCP/PIM pair */
THREADED SG(quad a[], quad b[], int idx[])
{
  int i;

  S-PARCELED THREADED  SG_ST (idx)
/*{Compute index in SCP */
    int i;

    for (i=(NODE_ID-1) * 10^5; i<NODE_ID * 10^5; i++)
      idx[i] = 1 + (13131313 * i) % 10^8;
  }
  END S-PARCEL;

  for (i=(NODE_ID-1) * 10^5; i<NODE_ID * 10^5; i++)
    b[idx[i]] = a[i];
END_FUNCTION();
}
```

Figure 4: HTMT SCATTER/GATHER

6.4 Peta-Kernel 4: 3-D Jacobi

The fourth peta-kernel performs a 3-D Jacobi computation. The Fortran 90 source code is as follows:

```
do k=1, 1000
    do j=1, 1000
        do i=1, 1000
            do m=1, 5
                do mp=1, 5
                    a(i,j,k,m) = a(i,j,k,m) + u(mp,m) *
                       (b(i+1,j,k,mp) + b(i-1,j,k,mp) + b(i,j+1,k,mp) +
                        b(i,j-1,k,mp) + b(i,j,k+1,mp) + b(i,j,k-1,mp))
                enddo
            enddo
        enddo
    enddo
enddo
```

A five level deep, perfectly nested loop without loop carry dependences describes the 3-D Jacobi. We can distribute the problem over all processors and therefore describe the HTMT code for one SCP, with the assumption that **a**, **b** and **u** are initially laid out in SRAM. The obvious goal in coding this problem for HTMT is to increase the locality of accesses in CRAM. To increase temporal locality we can permute the innermost two loops m and mp, allowing each of the **b** elements to be reused 5 times in the inner loop. Furthermore, we statically allocate the **u** matrix in CRAM since temporal locality of its elements is ensured every 25 loop cycles. Permuting the loop coordinates i, j, k would increase spatial locality depending on our memory layout. We assume at this point that our memory layout already ensures good spatial locality (details of the layout issue are irrelevant to our discussion here and are thus not included) and therefore it is not necessary to do this additional permutation. We perform a 3-D tiling where each of the 1000 processors computes a $100 \times 100 \times 100$ cube.[5] Each processor is identified by the processor number `NODE_ID`.

Figure 5 presents the tiled code for the Jacobi kernel. We assume that the **u** matrix has already been allocated in CRAM. The S-parcel, shown in lines 16–27, performs the actual computation. Each invocation of *3DJ_tile_ST* (line 16) computes **a** over a 10x10x10 matrix. The list given to the S-parcel states that **a** and **b** have to be percolated. **b** is given with additional ranges in square brackets []. The range tells the code generator that the **b** tile has to be extended with respect to the lower bound *lo* and higher bound *hi* of the 3 coordinates by one unit. These semantics assume that the programmer is very familiar with the generation of percolation objects generated by the compiler. All this information will be stored in the runtime descriptor of the S-parcel. Assuming that the additional complexity of the data layout of this problem compared to peta-kernel 1 does not impose an additional penalty, the expected aggregate performance can be computed analogously totaling 100 teraflops.

6.5 Peta-Kernel 5: Subtree Matching

The fifth peta-kernel decides if a small tree S with 10^4 nodes and an integer key at each node is a subtree of a similar tree T with 10^9 nodes. It is not explicitly stated whether the trees are rooted, but this is implied by the example provided in the description of the kernel. This application is considerably more difficult to parallelize than the other four peta-kernels, and a complete presentation of the code is beyond the scope of this paper. Here we will only provide a high-level description, beginning with an overview of the algorithm itself, followed by a discussion of specific HTMT implementation details.

For maximum generality, we assume arbitrary pointers are used to link nodes in the trees, with each node having a pointer to each of its children. Even so, S should be able to fit in each SCP's CRAM, assuming 4 bytes per key and 2 bytes per pointer in the limited CRAM space,

[5] Non-symmetric 3-D tilings might be used to increase locality depending on the layouts of **a** and **b**.

```
/* The following is the code for one SCP/PIM pair */
THREADED 3DJ(quad u[], quad a[], quad b[])
{
  int kt, jt, it, k, j, i, ts;

  ts=10; /* tile size for each coordinate */
  offset = (NODE_ID-1) * 100;
/* tiles are represented by kt, jt, it in all three Cartesian coordinates */
/* offsets in tiles are k, j, i */
  for (kt=1+offset; kt<=100+offset; kt+=ts)
    for (k=kt; k<=kt+ts-1; k++)
      for (jt=1+offset; jt<=100+offset; jt+=ts)
        for (j=jt; j<=jt+ts-1; j++)
          for (it=1+offset; it<=100+offset; it+=ts)
            for (i=it; i<=it+ts-1; i++)
            S-PARCELED THREADED 3DJ_tile_ST
               (a, b[lo-1:hi+1, lo-1:hi+1, lo-1:hi+1], i, j, k)
            {
              int m, mp;

              for (mp=1; mp<=5; mp++)
                for (m=1; m<=5; m++)
                a[i][j][k][m] = a[i][j][k][m] + u[mp][m] * (b[i+1][j][k][mp]
                   + b[i-1][j][k][mp] + b[i][j+1][k][mp] + b[i][j-1][k][mp]
                   + b[i][j][k+1][mp] + b[i][j][k-1][mp])
            }
            END S-PARCEL;
 END_FUNCTION();
 }
```

Figure 5: HTMT 3-D Jacobi

for a total of 60 Kbytes.[6] On the other hand, T is far too large even for the aggregate CRAM address space, although we assume it will fit within the aggregate SRAM.

The first step of the general algorithm is to divide the tree into equal partitions, as shown in Figure 6. The tree is traversed, breadth-first, until a level is reached in which the number of nodes equals p, the number of processors. Call the nodes at this level $\{t_0, t_1, \ldots, t_{p-1}\}$. (This probably won't be the exact size of any level of T, so this set will have to be split between

[6] Other, more space-efficient ways of representing S could be used as well.

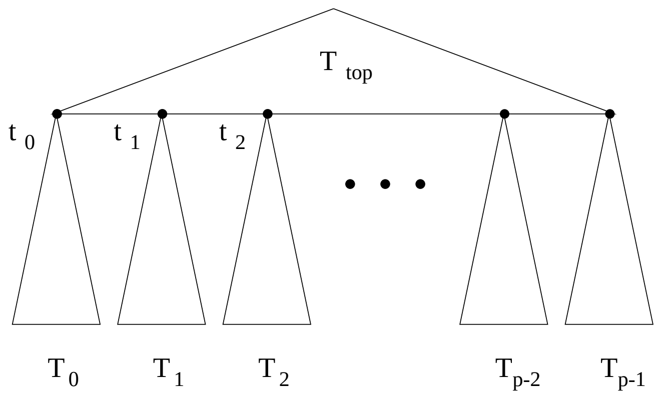

Figure 6: Partitioning T

two levels.) Each node in $\{t_0, t_1, \ldots, t_{p-1}\}$ is the root of a subtree; call these disjoint subtrees $\{T_0, T_1, \ldots, T_{p-1}\}$. Let T_{top} be T with all these subtrees removed. At this point, we assume T is balanced, so that the subtrees are roughly the same size, though we consider later what to do about unbalanced trees.

Once the tree has been subdivided, the root of S (call it s) is compared with every node in T_{top}. If a matching node r is found, the children of s are compared with the children of r, and so on down to the bottom of T_{top}. If no mis-match is found in T_{top}, then the problem reduces to comparing specific subtrees of S with specific subtrees in the set $\{T_0, T_1, \ldots, T_{p-1}\}$. Since there may be more than one node in T_{top} which matches S within the confines of T_{top}, each subtree in $\{T_0, T_1, \ldots, T_{p-1}\}$ may be assigned more than one node in S as a potential match.

Thus, each SRAM PIM will receive one subtree T_i and a (possibly empty) set of nodes $\{s_0, \ldots, s_k\}$, all of which belong to S. Each PIM will then answer two questions:

1. Is S a subtree of T_i?
2. For each s_k, does the subtree of S with root s_k match the top of T_i?

An affirmative answer to the first question is a solution to the complete problem. Affirmative answers to the second question get passed up to the processor with T_{top}, which combines them with comparison queries from other processors. If *all* comparison subproblems derived from the same match r in T_{top} are satisfied, then r is a solution to the problem.

Each subtree T_i will still be too large to fit in CRAM. Therefore, a similar divide-and-conquer strategy is needed for the next level. Each subtree T_i is divided further into a set of smaller subtrees, each of which can fit in CRAM. The comparisons over T_i are broken down into smaller problems, each of which can be formulated as both searching a subtree for an occurrence of S and comparing the top of the subtree against zero or more subtrees in S.

In HTMT, each of these subproblems can be handled by an S-parcel. Each PIM traverses its subtree T_i, invoking an S-parcel for each sub-subtree. The S-parcel percolates the sub-subtree to the SCP. This percolation will be more complex than with the other peta-kernels that use the SCPs, because the sub-subtrees in SRAM may be spread out over a range of memory addresses larger than one CRAM (spatial locality is not guaranteed), which means that the PIM will have to build a new, more compact tree. This mapping is sufficiently complex that it probably can't be generated automatically by a compiler, although many applications could use such a tree-to-tree mapping, so it may be put in a library.

Percolation of S-threads will be overlapped with computation within the SCPs, as with peta-kernels 1 and 4. Computation within an S-thread has the potential for significant amounts of parallelism. Both sub-problems (checking the sub-subtree for an occurrence of S and checking subtrees of S against the root of the sub-subtree) can be run as separate strands, and these may in turn be sub-divided further using tree recursion.

Imbalanced trees can be handled using load balancing. This can occur in several places. For instance, when T is being distributed among the PIMs during the partitioning phase, the subtrees can be counted while being traversed, and these counts can be used to adjust the boundary of T_{top}. In other words, the "leaves" of T_{top} (i.e., the roots of the distributed subtrees) don't have to be confined to one or two levels, but can go as far up or down as is needed to make the subtrees $\{T_0, T_1, \ldots, T_{p-1}\}$ approximately equal. Load balancing can also occur at the S-parcel level. EARTH allows a threaded function to be invoked using a **TOKEN** command, which causes the EARTH run-time system to start up the function on the node believed to have the least amount of pending work [4]. A similar mechanism can be supported in HTMT.

7 Run Time Support (RTS)

In this section, we present an overview of the HTMT runtime system which supports our program execution model. The runtime system design is a topic of its own, and will be left as the subject of a second paper. Here we only want to provide some insights which may be important in the study and development of the runtime system design.

We believe that the base threaded-level runtime system support in the HTMT runtime system can start by extending and refining the functions of the current EARTH runtime system. This includes support for efficient *synchronization*, *thread activation*, *communication* and *load balancing*. For a discussion of these runtime system functions, the readers are referred to the EARTH references [1, 4, 5, 6, 7]. However, to meet the demands of HTMT, innovative additions to the base runtime system are required. These include primarily the functions to support the thread percolation model as illustrated earlier in this paper.

First, we need to have an efficient runtime representation of the parcels. The runtime structure of a parcel should be initialized to contain:

- A pointer to the code ready to be percolated and the size of the code region;

- A set of addresses pointing to the data regions to be percolated, and the size of each data region;
- For each data region, a template "gathering function" to perform "runtime tiling," if necessary.

During the execution, a parcel may be "triggered" when its enclosing thread becomes logical enabled. At this time the runtime system will start the process of percolation, by a "preparation" and "reservation" process. The preparation procedure may need to gather the data using the corresponding gathering if runtime tiling is required, and calculate the size of the result data region. The reservation procedure will reserve the space in CRAM according to the sizes of the code and data region. Once the reservation is complete, the parcel is ready to be percolated, and the actual percolation will start by moving code and data from the SRAM to CRAM using a set of move functions. After the percolation is done, the thread id will be appended to the ready queue of the corresponding SCP for activation.[7]

In the following, we identify a number of runtime system functions which may be required (based on our preliminary experience with the peta-kernels) to perform the percolation according to the above description:

- Functions for data percolation preparation such as:
  - address translation functions used in the gathering function,
  - specialized gathering functions;
- Functions for CRAM space reservation;
- Functions for data movement between SRAM and CRAM for actual percolation.

Some of these HTMT percolation-specific functions can be made user visible, e.g., the specialized gathering functions. The design tradeoffs will be studied in another paper.

Further operations will be introduced as we progress in our investigation of required RTS operations and the study of additional kernels and benchmarks.

8 Conclusions

The objective of the Hybrid Technology Multi-Threaded (HTMT) architecture research is to determine the feasibility and structure of a parallel architecture integrating the combined capabilities of semiconductor, superconductor, and optical technologies. This mix of technologies

[7] the percolation just described can actually be cascaded between several memory levels.

may yield operational attributes superior to those based solely on semiconductor technology. An important challenge is to handle the significant latency disparity between the superconducting processors and the memory latency at various levels within the deep memory hierarchy.

In this paper, we present an outline of the proposed HTMT program execution model which provides a cornerstone upon which the HTMT architecture can keep all critical resources in the system usefully busy for petaflops-scale applications, while the latency of the multi-level memory hierarchy is effectively managed.

The paper describes the operational model (semantics) of the corresponding multithreaded programs expressed as multithreaded program graphs. We present the runtime thread model, and illustrate the notion of "parcels," which are threads encapsulated to be enabled in PIM and percolated to the superconductor processors for execution. We also introduce the notion of a *strand* which, unlike threaded function invocations, does not require the allocation of a new frame, thus minimizing the overhead associated with its initiation and termination.

We present a first version of the HTMT Operation Set, focusing on HTMT thread-level primitives, e.g., thread-level control, synchronization and data movement operations. In particular, we explain the operations associated with thread percolation, as this is an unusual feature which is at the center of the proposed HTMT program execution model. Much of the discussion of the operation set has referred to the Threaded-C operations which have been successfully used in the EARTH project, both to illustrate and define the EARTH model as well as to serve as a thread-level programming interface for the underlying EARTH architecture. We are capitalizing on our previous experiences and are working toward the extension of Threaded-C for a similar purpose in defining the HTMT program execution model.

The proposed HTMT program execution model must be examined through the mapping of different benchmarks and applications. In this paper, we used the five peta-kernels as a case study to asses the functionality, performance and programmability of the proposed model and the HTMT operation set. Although some of the kernels are quite simple, the case study has been a worthwhile exercise, and a number of issues have been exposed during this study. Some have been resolved, while others have been identified as future research topics.

The paper is open to criticism and comments from the HTMT processor/PIM architects, potential compiler writers. and applications researchers. We will extend and modify the HTMT program execution model and its operation set based on the feedback we receive.

References

[1] Haiying Cai, "Dynamic Load Balancing on the EARTH-SP System," Master's thesis, McGill University, Montréal, Québec, May 1997.

[2] G. R. Gao, "An Efficient Hybrid Dataflow Architecture Model," *Journal of Parallel and Distributed Computing*, 19(4):293–307, December 1993.

[3] Herbert H. J. Hum and Guang R. Gao, "A Novel High-Speed Memory Organization for Fine-Grain Multi-Thread Computing," in *Proceedings of PARLE '91 – Parallel Architectures and Languages Europe,* volume I, no. 505 in Lecture Notes in Computer Science, Eindhoven, The Netherlands, pp. 34–51, Springer-Verlag, June 1991.

[4] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren, "A Study of the EARTH-MANNA Multithreaded System," *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

[5] 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 *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, Limassol, Cyprus, pp. 59–68, ACM Press, June 1995.

[6] 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*, Cancún, Mexico, pp. 288–294, IEEE Computer Society, April 1994.

[7] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian, "Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, Philadelphia, Pennsylvania, pp. 178–188, May 1996.

[8] Olivier C. Maquelin, Herbert H. J. Hum, and Guang R. Gao, "Costs and Benefits of Multithreading with Off-the-Shelf RISC Processors," in *Proceedings of the First International EURO-PAR Conference*, no. 966 in Lecture Notes in Computer Science, Stockholm, Sweden, pp. 117–128, Springer-Verlag, August 1995.

[9] James Rumbaugh, "A Data Flow Multiprocessor," *IEEE Transactions on Computers*, 26(2):138–146, February 1977.

[10] Semiconductor Industry Association, San Jose, California, *The National Technology Roadmap for Semiconductors*, 1994. Revised.

[11] Thomas Sterling *et al.*, "A hybrid technology multithreaded computer architecture for petaflops computing." Grant proposal submitted to DARPA in response to announcement BAA-97-03, March 1997.