A

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-258

Application Performance Evaluation of the HTMT Architecture[*]

by

Mark Hereld,[1,2] ***Ivan R. Judson***,[1] ***Rick Stevens***[1,2,3]
{hereld, judson, stevens}@mcs.anl.gov

[1]Mathematics and Computer Science Division, Argonne National Laboratory
[2]Computation Institute, The University of Chicago
[3]Department of Computer Science, The University of Chicago

Mathematics and Computer Science Division

Technical Memorandum No. 258

January 2003

[*]Our work on the study summarized here was supported by NAS7-1260. Preparation of this report was supported by WFO Agreements with Jet Propulsion Laboratory No. 85L70 per ANL Proposal P-01014, *Hybrid Technology Multithreaded (HTMT) Application Benchmarking Final Report*, and by No. 858H3 per ANL Proposal P-97084, *Hybrid Technology Multithreaded (HTMT) Computer Architecture for Petaflops Computing*.

Argonne National Laboratory, a U. S. Department of Energy Office of Science laboratory, is operated by The University of Chicago under contract W-31-109-Eng-38.

Available electronically at http://www.doe.gov/bridge

Available for a processing fee to U.S. Department of Energy and its contractors, in paper, from:

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
phone: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Contents

Abstract……..…1

1. Setting the Scene……..2

1.1 Applications Needing Petascale Computing…….…….…3
1.2 Roadmap to This Report…… 5

2. HTMT Performance Evaluation Framework..…….... 5

2.1 ***HTMT Overview***……6
2.2 ***Guiding Principles***…….……..…8
2.3 ***The Modeling Hierarchy***……..……...…...…9
2.3.1 Tier 1: Application Phase Decomposition…..…… 9
2.3.2 Tier 2: Partitioning into Parcels…… 10
2.3.3 Tier 3: Governing Equations and Parameters…… 11
2.4 ***Applying the Model***…… 13

3. Critical Evaluation of the HTMT Architecture…….. 15

3.1 ***Application Suite Overview***……..…….…… 15
3.2 ***Summary of Benchmark Applications Analysis***……..…….. 15
3.2.1 Dense Matrix Multiply…….…… 15
3.2.2 Synthetic Aperture Radar…….…… 18
3.2.3 Plasma PIC…….……21
3.2.4 Volume Rendering…..……...….25
3.3 ***Summary Evaluation of Application Performance***……... 28

4. Evaluation of HTMT-C as a Tool……...…….. 30

4.1 ***Key Questions***…….. 31
4.2 ***Summation of HTMT-C Remarks***…….. 33

5. Final Comments……..… 33

Acknowledgments……...… 35

References…… 36

Application Performance Evaluation of the HTMT Architecture

by

Mark Hereld, Ivan R. Judson, and Rick Stevens

Abstract. In this report we summarize findings from a study of the predicted performance of a suite of application codes taken from the research environment and analyzed against a modeling framework for the HTMT architecture. We find that the inward bandwidth of the data vortex may be a limiting factor for some applications. We also find that available memory in the cryogenic layer is a constraining factor in the partitioning of applications into parcels. The architecture in several examples may be inadequately exploited; in particular, applications typically did not capitalize well on the available computational power or data organizational capability in the PIM layers. The application suite provided significant examples of wide excursions from the accepted (if simplified) program execution model – in particular, by required complex in-SPELL synchronization between parcels. The availability of the HTMT-C emulation environment did not contribute significantly to the ability to analyze applications, because of the large gap between the available hardware descriptions and parameters in the modeling framework and the types of data that could be collected via HTMT-C emulation runs. Detailed analysis of application performance, and indeed further credible development of the HTMT-inspired program execution model and system architecture, requires development of much better tools. Chief among them are cycle-accurate simulation tools for computational, network, and memory components. Additionally, there is a critical need for a whole system simulation tool to allow detailed programming exercises and performance tests to be developed.

We address three issues in this report:

- The landscape for applications of petaflops computing
- The performance of applications on the HTMT architecture
- The effectiveness of HTMT-C as a tool for studying and developing the HTMT architecture

We set the scene with observations about the course of application development as petaflops computing becomes possible to contemplate. We then address the topic of application performance analysis on this architecture, including our analysis framework and the concepts leading up to its adoption, summary analyses of four computationally distinct test applications, and directions in performance analysis for complex hybrid architectures such as the HTMT. We briefly discuss the strengths and weaknesses of HTMT-C, and we then conclude with comments on future performance analyses.

1 Setting the Scene

In the early 1990s discussions began in the United States (and perhaps elsewhere) to consider the technology paths that could lead to the development of petaflops computers − systems capable of sustaining over one quadrillion (10^{15}) floating-point operations per second. The results of these discussions were well documented in a book and in a series of workshop reports (PetaFLOPS Workshop Series 1994-1999). Chief among the essential results from this work was that many applications domains could effectively use petaflops capabilities and that these applications domains have a variety of memory and turnaround requirements.

At the Pasadena Petaflops Meeting in 1994, the Bodega Bay Summer School on Petaflops in 1995, and the Petaflops II meeting in Santa Barbara in 1999, applications of petaflops systems were discussed. Applications were considered from science (including computer science), engineering, policy, national security, business, and entertainment. Requirements identified for petaflops applications included the following:

- Memory and cache footprints (the amount of memory required at each level of the memory hierarchy)
- Degree of data reuse associated with core kernels of the application, the scaling of those kernels, and the associated estimate of memory bandwidth required at each level of the memory hierarchy
- Instruction mix required by the application
- I/O requirements and secondary storage needed for intermediate results or checkpoints
- Amount of concurrency available in the application, and communications requirements (bisection bandwidth, latency, fast synchronization patterns)
- Use modality (batch, real-time, interactive, multi-user) of the application and expected turnaround times

These requirements are discussed in (Sterling, Messina, and Smith 1995). In many cases the applications analysis can be reduced to understanding the memory bandwidth requirements for kernel algorithms and the scaling properties of these core kernels (i.e., how do the memory capacity and bandwidth requirements scale as problem size increases?). Traditional scientific applications areas such as general circulation models, quantum chromodynamics, and fluid dynamics in astrophysics have relatively well understood requirements. These applications areas also have significant scalability in that the computational complexity grows faster than the memory requirements as the problem scales, implying that sustained petaflops performance could be supported by a memory system that is significantly smaller than a petabyte. Other applications areas such as data mining and decision support have a much higher need for memory. Thus, the relative importance of different types of application use modalities is an important consideration in determining feasible design points for petaflops systems.

A key insight from this work is that designing breakthrough computer architectures without direct feedback from application performance analysis is a risky business at best. Consequently, the Hybrid Technology Multi-Threaded Architecture (HTMT) project included consideration of the performance of current and future application as an integral part. This report summarizes the work of the application performance study group. To execute the application performance study in concert with the rest of the HTMT design work, we assembled a group of application domain experts, sampling a wide range of disparate application types, each relying on a different set of underlying algorithms and computational needs. In addition to participating in regular all-hands HTMT progress meetings, the group convened independent workshops and working meetings to define the performance evaluation framework. For each application domain, an analysis of the performance of a representative code was carried out.

1.1 Applications Needing Petascale Computing

Many potential applications for petaflops and petaops computing systems can be imagined (Stevens and Taylor 1995):

- Materials simulations that bridge the gap between nanoscale and macroscale (bulk materials)
- Coupled electromechanical simulations of nanoscale structures (dynamics and mechanics of micromachines)
- Full plant optimization for complex processes (chemical, manufacturing, and assembly problems)
- High-resolution reacting flow problems (combustion, chemical mixing, and multiphase flow)
- High-realism immersive virtual reality based on real-time radiosity modeling and complex scenes
- Time-dependent simulations of complex biomolecules (membranes, synthesis machinery, and DNA)
- Multidisciplinary optimization problems combining structures, fluids, and geometry
- Modeling of integrated earth systems (ocean, atmosphere, biogeosphere)
- Improved data assimilation capability applied to remote sensing and environmental models
- Computational cosmology (integration of particle models, astrophysical fluids, and radiation transport)
- Computational testing and simulation as a replacement for weapons testing (stockpile stewardship)
- Simulation of plasma fusion devices and basic physics for controlled fusion (to optimize design of future reactors)
- Design of new chemical compounds and synthesis pathways (environmental safety and cost improvements)
- Comprehensive modeling of groundwater and oil reservoirs (contamination and management)
- Modeling of complex transportation, communication, and economic systems

The potential user communities, and therefore the applications that they develop and employ, for petascale systems fall into (at least) four categories (Stevens and Taylor 1995). The first two categories represent the traditional users of high-performance computing systems. The third category captures the segment of the community that is primarily concerned with high throughput rather than capability (due either to lack of software infrastructure that scales or to the complexity of the problem solving tool chain). The fourth category represents those users that are primarily concerned with interactive analysis and rapid prototyping and could benefit dramatically from the power and storage resources of a petascale system but would need support for interaction and timesharing, both of which are not currently design goals for petaflops systems.

In the *aggressive category* we place those that have been working for some time at the frontiers of high-end computing (e.g., astrophysics, cosmology, QCD) who are extremely well prepared to move to new architectures. They have codes that are well understood from the standpoint of performance, scalability, and architectural mapping; moreover, the developers are prepared and motivated to produce new versions of these codes targeting machines several orders of magnitude increased scale.

In the *early adopters category* we place other communities that are currently using large-scale machines but whose culture and code development infrastructure is less oriented toward exploiting the very latest hardware (or even experimental hardware). Applications in this category include molecular dynamics, quantum chemistry, biomolecular modeling, computational geophysics, and climate modeling. In general, these codes are slightly more complex (perhaps with more time and space scales and with more embedded physics/chemistry). The "early adopters" communities are generally ready to move to new systems; but because of the complexity of their codes and the relatively smaller size of the their groups, they generally take longer to move the substantial code base onto new systems. Because of this challenge in porting, this group has also developed software infrastructure to ease the migration to new systems (e.g., global arrays). Combined, these two categories consume the bulk of supercomputing cycles at open academic computing centers in the United States. These two groups typically form the core target for non-defense petaflops computing.

In the *high-throughput category* are user communities that have well-defined computational and data analysis problems from fields such as electronic circuit design, bioinformatics, MCAD, ECAD, design optimization, chemical engineering, and medical imaging. In many cases they lack highly scalable algorithms or even implementations that are available for ongoing development. Their systems are characterized by having well-defined interfaces to databases and other tools, and the end user/developer is often able to alter only a small portion of the overall system. This category of user can benefit from petaflops developments in several ways. First, there is the possibility of accelerating existing problems by several orders of magnitude (perhaps by enabled automated optimization) that might dramatically alter the pattern of development or problem solving. Second, the technology that enabled petaflops may also enable inexpensive teraflops for these applications, thus directly reducing the cost of these

computations. Third, petascale systems will enable components to be combined in new ways that may dramatically improve overall throughput.

The fourth category we call the *exploratory computing category*. This category includes a broad collection of users and areas where the computer is being used as a tool for rapid prototyping of ideas or algorithms or where the essential problem being solved involves interactive human-guided search. Examples here include data visualization, proof finding by automated deduction, data mining in sociology, interactive programming, and analysis using tools like Matlab and Mathematica. The key attribute of this category is that the human is in the loop and the problem-solving pattern involves a substantial amount of human interaction with the computer and with the algorithms under development. Ironically, although this category is only barely making use of existing large-scale computers, many of the future scientific and social impacts of computing are likely to come from this segment of the community. The potential benefit of petaflops/petaops computing to this category of user is immense, ranging from increasing the scale at which interactive computational experiments can be conducted to reducing the time from prototype development to widespread use. For example, if a petaflops system can be used to enable a very high level language-interpreted environment to perform at the same rate as a dedicated teraflops system, then it can be used to rapidly test ideas at full performance levels that then could be deployed on lower-cost platforms. An important element of this type of use is the capability to interactively explore terabyte or petabyte datasets that may lead to improved productivity in a number of areas. Because of the human-in-the-loop nature of this category of user, it may be feasible for multiple users to share the same petaflops systems, provided that the system can support true time-shared multi-user access. The ability to support time-shared access to petascale systems is an important new design consideration.

1.2 Roadmap to This Report

In Section 2 we describe the framework that we developed for evaluating the performance of applications on the hypothetical HTMT architecture. At the end of that section we describe the shortcuts we took to streamline the model to the needs and available resources of the present application study. In Sections 3 and 4 we summarize the results of our application study. We end the report with a description of extensions to the present framework that would enable a truly general class of performance analyses.

2 HTMT Performance Evaluation Framework

Our study began with general discussions of performance evaluation of applications on generalized abstract architectures, of which the HTMT was a particular case. At this level we considered hierarchical model descriptions, simulation techniques, emulation techniques, and possible instantiations of these using available tools (extensions to Threaded-C).

Here we describe the results of that process. In particular, we discuss the structure and organization of current and future application codes, the program execution model

developed by the HTMT research team, the parcel percolation model, and the best available parameters of the constituent hardware technologies.

The motivating questions for our study of the performance of applications on the HTMT architecture are as follows:

> *Will the application perform well?* If the application does perform well, then we have effectively hidden the latency in the memory hierarchy while exploiting the aggregate performance of the system.
>
> *What system parameters are responsible for the performance?* The answer to this question can help us determine where to apply additional effort in the hardware design. What are the key systems performance parameters? Is performance particularly sensitive to small changes in certain parameters of the hardware?
>
> *Is the system balanced from an applications viewpoint?* Here we are looking not only for bottlenecks in the system, in the sense of the preceding question, but also for opportunities to capitalize on underutilized resources: bandwidth of the data vortex, computational power of the SPIM and DPIM layers, lateral memory bandwidth in these layers. Are these resources – memory bandwidth, communications, I/O, and compute speed – exercised (after a suitable average over applications) in a balanced way?
>
> *Do we understand the execution model and its performance?* Apart from the performance of the hardware, we can ask whether our conception of the execution model is a source of degraded performance. Is the program execution model sufficiently expressive to enable application programmer's adequate efficient access to the hardware?
>
> *What tools are needed to improve performance analysis and prediction?* What tools will most effectively advance our understanding of the performance of applications running the HTMT architecture beyond the limitations of the current work?

In the following section we present the modeling framework developed for this study and designed to enable us to address these questions.

2.1 HTMT Overview

Before discussing the performance analysis framework we briefly describe the salient features of the hardware and software architecture as currently envisioned. A schematic of the central large-scale architectural features of the HTMT architecture is given in Figure 1. Depth and details can be found elsewhere regarding overall HTMT design (Sterling and Bergman, 1998), RSFQ technology (Bunyk et al. 1997; Dorojevets et al. 1999), CNET (Wittie et al. 1999), PIM (Kogge et al. 1996; Kogge 1999), VORTEX (Arend, Bergman, and Reed 1998; Arend, Reed, and Bergman 1998), and holographic memory systems (Liu, Chuang, and Psaltis 1998). The HTMT comprises several layers of

computationally enabled memory. The parameters collected in Figure 1 follow the configuration called “Oct 98” in Kogge (1999).

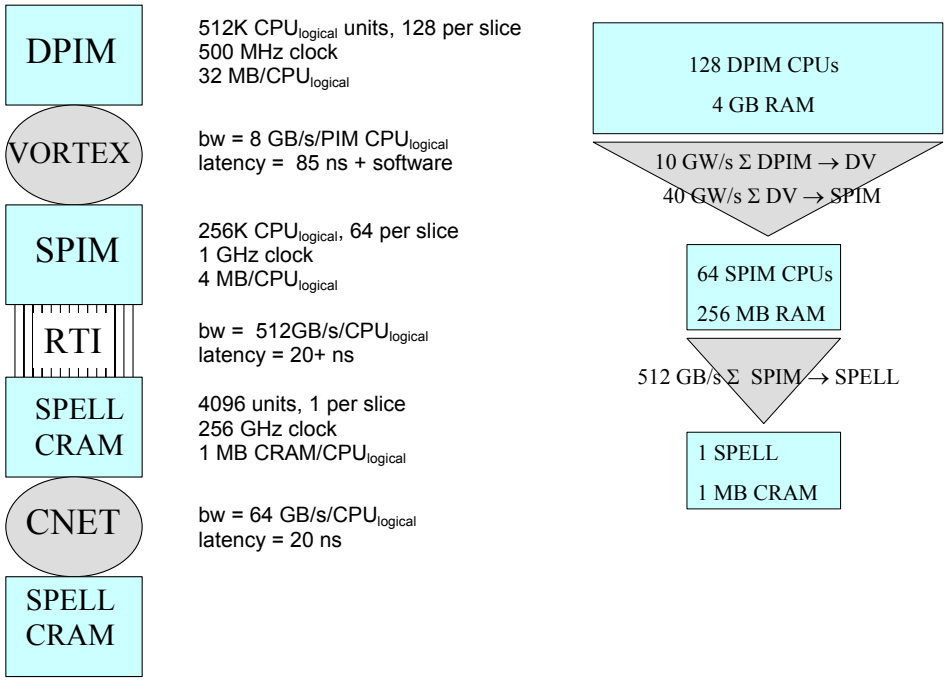

Figure 1 Overview of the HTMT architecture. On the left are the major components of the system with characteristics of individual nodes and channels. On the right is a schematic of the aggregated resources at each layer of the architecture.

The fastest per node processing power is in the cryogenic SPELL layer of the architecture. Built from RSFQ logic, each SPELL runs at about 256 GHz. There are 4,096 of these in the petascale machine. The associated cryogenic memory, CRAM, is limited. Each multithreading SPELL is serviced by a cluster of room-temperature processor-in-memory nodes with associated static memory, the SPIM layer – 64 CPU nodes with 4 Mbytes each, running with 1 GHz clock, packaged 8 nodes to the chip. Communication between these layers is largely vertical. The SPIM layer is broadly connected through an optical network to the DPIM layer, a large collection of PIM nodes endowed with dynamic memory – 128 CPU nodes with 32 Mbytes each, running with a 500 MHz clock, packaged 4 nodes to the chip. For some purposes it is useful to think of vertical *slices* of the HTMT machine, each topped by a single SPELL (even though DPIMs and SPIMs can communicate freely with one another across slice “boundaries”). Each slice of the machine has one optical fiber into and out of the associated DPIM cluster connecting it to the data vortex. There is one fiber out of and four fibers into the SPIM cluster. Each fiber can carry 10 Gwords/sec. The asymmetry in the data vortex port

configuration to the SPIM cluster enables bursts of 40 Gwords/sec into the SPIM, to handle the anticipated occasional higher traffic heading toward the SPELLs. Note that there are not sufficient fibers feeding the data vortex, only two for every four into the SPIM, to keep this data rate up across the entire machine – the maximum sustainable data rate from DPIM to SPIM (and vice versa) averaged over the entire machine is 10 Gwords/sec.

The program execution model, called *parcel percolation*, leverages the hardware hierarchy explicitly. Units of work in both computation and data communication are organized into parcels that contain executable code and data in proportion to their function. Parcels executed in the SPELLs (at least) are constrained by convention to be nonblocking in order to minimize the dead time of these processors. More details of the percolation model are given in the following sections. A more complete guide to the model in contained in (Gao et al. 1997; Gao et al. 1998).

2.2 Guiding Principles

We are guided in the design of our application framework by a few key concepts. Some have been implemented or partially implemented for the current study, while others remain as challenges for further work. As part of our remarks at the end of the paper we outline a possible automatic performance evaluation system encompassing our grandest conception of this framework.

Models targeted at predicting runtime for applications. The highest-level aim is to develop a framework for expressing application models that can be used to predict runtime in complex architectures such as the HTMT. The total execution time of an application running on the machine will have contributions from computation, communication, and overhead. Computation in one part of the algorithm is overlapped with communication and overhead of others. To achieve this basic goal, our models need to address data motion and computation and the costs of the HTMT execution model.

Hierarchy of models. Throughout the development of our framework we thought broadly about the structure of application programs, machine subsystems, architecture topology, and program execution models. We wanted to base our end-to-end performance predictions on accurate models of the pieces of the system. By working with a hierarchy of detail in our modeling strategy we hoped to capture for future use and multiple reuse the work put into characterizing the pieces. We also intend this approach as a means to enable rapid reconfiguration of the applications, machines, and execution models as part of the design cycle and optimization.

Support for model composition. Ultimately, our framework must support model composition in order to enable modular development of an entire model, to allow for rapid reconfiguration of a model by substitution of components for performance comparison, and to facilitate reanalysis of application performance in the face of architectural and parametric changes. We mean to include in our support of model

composition the ability to express overlapping and nonoverlapping factors and coupling between model terms.

2.3 The Modeling Hierarchy

We explicitly model several levels of composition: application phases, algorithm, parcels, and subsystem timing equations.

In the following paragraphs we begin, in a sense, at the highest level of the model and describe the method by which we decompose a given application into its analyzable segments. We then describe the mapping of these application fragments onto the parcel percolation model. Finally, we describe the computation of resource usage (principally processor time and communication bandwidth) and transcription of these through the higher levels of the model to the ultimate estimates of application execution time and aggregate resource usage.

2.3.1 Tier 1: Application Phase Decomposition

The goal of our highest-level decomposition is to divide the application into its component algorithmic phases. As an example of this breakdown, consider the plasma PIC (Norton, Decyk, and Cwik 1999) code analyzed as part of our application suite and summarized later in this report. After initialization, the code computes the evolution through time of a plasma. For each time step the algorithm can be broken down into separate phases describing the handling of different aspects of the computational physics and management of the forward propagation of the material and field configurations.

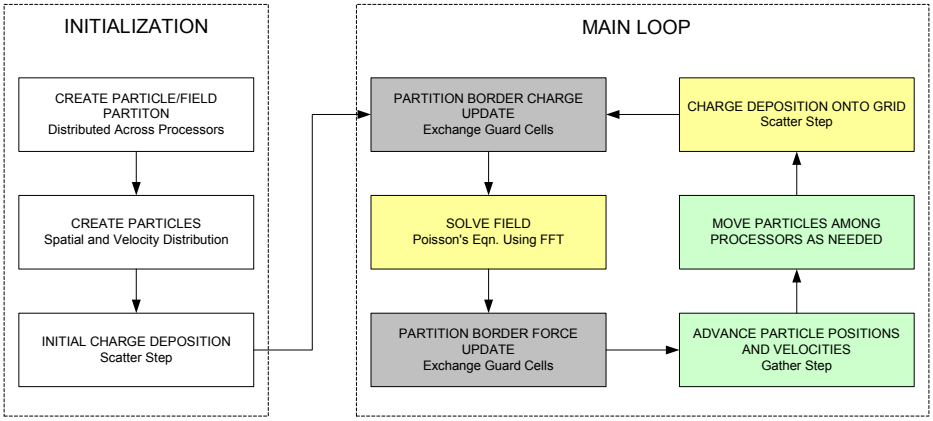

Figure 2 Decomposition of an application into algorithm phases. In this case the PlasmaPIC algorithm passes through several discrete phases per cycle at each time step (adapted from Norton, Decyk, and Cwik 1999).

The major phases of the algorithm are (1) interpolate charge density (charge deposition) to the grid using current particle positions, (2) solve for the resulting electric field, (3)

compute the force on all particles, and (4) move the particles in response to this force. These phases are captured schematically in Figure 2.

2.3.2 Tier 2: Partitioning into Parcels

The next level of detail of our modeling framework bridges the algorithmic description of the application, the program execution model, and some of the aspects of the hardware architecture. At this level our model needs to address the data motion and computation explicitly. Having broken an application into its component phases and identified the principal algorithmic fragments, we now subdivide each phase into pieces consistent with the parcel percolation model (Gao et al. 1997; Gao et al. 1998) and the constraints of the architecture. This process will pave the way for expression of our model at the finest levels of detail, described in the next section.

A parcel, in this context, represents a unit of computational work, accompanied by data, (usually) requiring no additional data to run to completion. The percolation process describes the "life" of the parcel. We decompose the basic percolation of a parcel through the system into something resembling the following five phases:

- assemble – gather data, attach code, build header
- dispatch – administration and deposit parcel in CRAM
- execute – computation in SPELL
- retire – decommission and retrieve from CRAM
- scatter – deliver data results

This particular decomposition does not hold faithfully in all cases, for all applications, or even for all work fragments within a given application. It is merely a guide for this level of explication. The typical associations of these parcel phases with particular layers of the computational and memory hierarchy (e.g., CRAM) are included for the sake of concreteness only. Generally, DPIM and SPIM layers of the HTMT architecture are responsible for building parcels that can be executed at the SPELL layer. DPIM has often been most closely associated with parcel assembly. Likewise, SPIM is often thought of as the executive heart of the machine – tending to the "feeding" of the SPELLs with work sufficient to keep them always busy. The SPELL most typically carries the closest association as the computational heart of the machine. The parcel percolation model underlying the HTMT execution model then suggests that parcels typically migrate, over the course of parcel assembly, from the DPIM to the SPELL. At the SPELL the parcel executes to completion; no blocking is allowed. Results are then passed back downward toward the DPIM in a scatter operation. The architecture and the program execution model are in fact more flexible than this, and the modeling framework is likewise able to express more complicated scenarios.

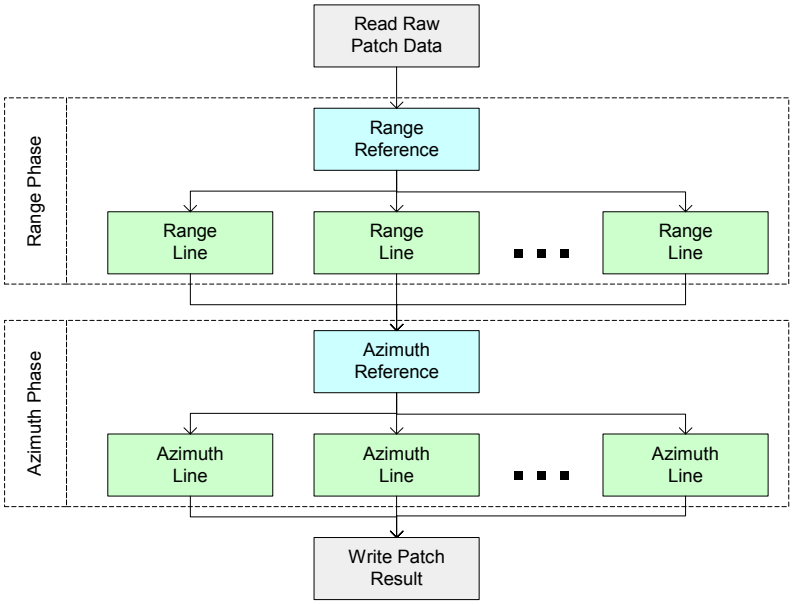

Figure 3 Decomposition into parcels. The synthetic aperture radar (SAR) application processes patches of measurements according to the above decomposition. Patches are processed independently to create a large map. Each patch is processed in two major application phases. The phases are divided here into pieces assigned to parcels.

As an example of dividing an application further into parcels, consider the synthetic aperture radar, SAR, application analyzed in Siegel and Craymer (1999) and summarized in Section 2.3.1. Independent 2-D patches of data are analyzed in two phases, called the Range and Azimuth phases in Figure 3. For each of these phases, a single parcel computes the reference data used by a slew of parcels, each of which computes a line of the SAR reduction. The figure shows the dependencies within a patch reduction. The scattered results from the individual range line calculations are gathered "orthogonally" in the assembly of the azimuth line parcels.

2.3.3 Tier 3: Governing Equations and Parameters

Each phase in the parcel's passage from assembly to deconstruction consumes hardware resources in the form of memory space, processor cycles, and communication bandwidth. It also consumes software system resources such as queue and buffer slots. In this section we present the schematic form of the equations we incorporate into our model.

We emphasize that the sums presented in these equations are only placeholders for the actual composition operations they represent. They do not explicitly represent the parallelism and pipelining that must be accounted for once the individual terms have been computed. The composition of these terms into a final estimate of the application execution time, for example, must take into account application-dependent overlapping of

work and communication terms. This composition is handled separately, and for the present it is handled manually.

We consider three components to the total execution time of a piece of computational work: the time it takes to carry out the calculation itself, the time to move data (communication), and system overhead associated with this work. When these components can be carried out entirely in parallel, the total execution time is the maximum of the three. When they must be carried out entirely serially, then the total execution time is given by the sum of the three. We represent this kind of relation generally as $f(\ )$,

$$T_{TOTAL} = f(T_{WORK} , T_{COMMICATION} , T_{OVERHEAD}),$$

and leave its final expression the circumstances dictated by the application. The same kind of function appears often in the following relations – taking on a value between the maximum of its terms and their sum depending on how much of the described work can be overlapped. Using this shorthand notation, we do not imply that the function is the *same* in all cases. It depends in detail on the interaction of the architecture, the program execution model, and the application.

Computational work can be carried out in the DPIM, SPIM, and SPELL layers of the HTMT architecture. Often, the SPELL may be used for computations and the other layers used solely to carry out operations in support of feeding the SPELLs. Borrowing the notation introduced in the previous relation, we have

$$T_{WORK} = f(W_{SPELL} , W_{SPIM} , W_{DPIM}).$$

Short of detailed microkernel timing simulations, one can characterize a kernel accounting for a simple tabulation of basic operations. By using this estimation method we are assuming that the kernel can be arranged to keep the pipeline full and free of internal contentions. Lowest-order corrections to this ideal model can be affected by adjusting the values of t_X to reflect effective time for each operation averaged over an ensemble of instructions. Accordingly, we model work done by each computing node of the architecture by breaking the kernel into contributions from computation and local or register data movement:

$$W_X = f(t_{FLOP} * N_{FLOP},\ t_{IOP} * N_{IOP},\ t_{LOAD} * N_{LOAD},\ t_{STORE} * N_{STORE}).$$

The time to move data from one part of the architecture to another includes contributions from each of the interconnection fabric layers of the architecture:

$$T_{COMMUNICATION} = f(C_{CNET} , C_{VORTEX} , C_{SPIM} , C_{DPIM} , C_{RTI}).$$

The parameters of the model depend on the source and destination within the architecture. For example, the time to move a word from SRAM to CRAM over the RTI

is shorter than to move it from DRAM to SRAM over the Data Vortex. The constituent communications contributions are modeled with startup time, data transfer time, and account of topology in the form of hop counting. We explicitly model data motion between neighbors in the architecture and compute the end-to-end performance by suitable composition of these steps. For moves that span large lateral distances we use the algebra of hops to model the additional time required:

$$C_X = t_{STARTUP\_X} + n * W_X + DISTANCE_X.$$

This model has its shortcomings. In using it we are relying on effective values of these parameters to adequately account for many effects, including contention. We are also, for example, using full channel bandwidth as the basis for t_{MOVE}. If a single processor cannot actually fill the channel, we implicitly assume that the process has been spread across enough processors so that the data can be moved fast enough.

Finally, the overhead is broken down in our model into contributions from parcel handling, work scheduling, and algorithmic overhead. Again, the time required is bounded by the max( ) and sum( ) depending on how the component terms interact:

$$T_{OVERHEAD} = f(O_{PARCEL}, O_{SCHEDULING}, O_{ALGORITHM}).$$

2.4 Applying the Model

Having described in some detail the modeling framework that drove our application analysis, we now describe the steps taken to apply it. The scope of the task and resources available force us take aggressive measures to distill from this framework an approach that captures the essential essence, is extendible, and achieves the primary goals. Our primary goal for this study has been to aid in the design phase of the HTMT architecture. To this end we took as our highest priorities to estimate the performance of real applications as they would run on the HTMT and to identify bottlenecks in the hardware architecture and in the program execution model that would impact the HTMT design.

For each application we do the following:

1. Identify key phases of the main algorithms
    - Distill to key phase or phases based on judgement and/or performance data from profiling
2. Break the key phase up into parcels
    - Identify closures
    - Consider data partition size vs. available memory (CRAM)
3. Develop a functional form capturing complexity of each phase
    - Measure or derive complexity coefficients for each term in the model
    - Fit parameters to simulation data (machine model and applications model terms)

4. Evaluate the model components by hand, using spreadsheets, etc.
    - Compute time cost for each basic resource
    - Normalize to use entire slice (for example all DPIMs)
5. Compose models for end-to-end prediction
    - Compute resource utilization to identify bottlenecks in the architecture
    - Assess impact on available memory at all layers
    - Calculate pipeline timing for critical parcels

Table 1 Summary of parameters describing the basic HTMT subsystem performance and the configuration of the entire system. (* For the steady-state calculations of this study, we cannot take advantage of this factor of 4 because there is only one fiber from the DPIM layer to feed the SPIM.)

Basic Parameters of the Model					
Parameter	**Value**	**Available Parallelism per Slice**	**Aggregate Slice Value**	**Aggregate HTMT Value (4096 Slices)**	**Units**
Work					
T_{SPELL}^{-1}	2.56 E+11	1	2.56 E+11	1.05 E+15	[cycles/sec]
T_{SPIM}^{-1}	1.00 E+09	64	6.40 E+10	2.62 E+14	[cycles/sec]
T_{DPIM}^{-1}	5.00 E+08	128	6.40 E+10	2.62 E+14	[cycles/sec]
Communication					
$T_{S\ to\ C}^{-1}$	6.41 E+10	1	6.41 E+10	2.63 E+14	[W/sec]
$T_{C\ to\ S}^{-1}$	6.41 E+10	1	6.41 E+10	2.63 E+14	[W/sec]
$T_{DV\ to\ S}^{-1}$	1.00 E+10	4*	4.00 E+10	1.64 E+14	[W/sec]
$T_{S\ to\ DV}^{-1}$	1.00 E+10	1	1.00 E+10	4.10 E+13	[W/sec]
$T_{D\ to\ DV}^{-1}$	1.00 E+10	1	1.00 E+10	4.10 E+13	[W/sec]
$T_{DV\ to\ D}^{-1}$	1.00 E+10	1	1.00 E+10	4.10 E+13	[W/sec]
$T_{H\ to\ D}^{-1}$	5.00 E+10	1	5.00 E+10	2.05 E+14	[W/sec]
$T_{D\ to\ H}^{-1}$	1.25 E+09	1	1.25 E+09	5.12 E+12	[W/sec]
Memory					
M_{CRAM}	1.31 E+05	1	1.31 E+05	5.37 E+08	[W]
M_{SRAM}	5.24 E+05	64	3.36 E+07	1.37 E+11	[W]
M_{DRAM}	4.19 E+06	128	5.37 E+08	2.20 E+12	[W]
M_{HRAM}	1.00 E+09	32	3.20 E+10	1.31 E+14	[W]

Table 1 summarizes some of the parameters that drive our model. The per element values are in the second column – scaled, for example, to the single logical computing element or the data vortex optical fiber. The number of elements is given under the heading *Available Parallelism* and is used to compute the aggregate value available per slice. In the penultimate column the parameters are scaled to aggregate the resource over the entire 4096-slice HTMT machine. Units are given in the final column. Note that the memory capacity is given here in words whereas in Figure 1 it is given in bytes.

3 Critical Evaluation of the HTMT Architecture

In this section we summarize the results of our evaluation of the HTMT design.

3.1 Application Suite Overview

We considered in detail a fairly large number of applications as part of our survey for this analysis:

- adaptive N-body problem algorithm
- plasma particle-in-cell code
- Cannon's algorithm for matrix multiplication
- volume rendering
- synthetic aperture radar
- molecular dynamics

We attempted to sample the space of application domains broadly in an effort to find examples for our test suite that would stress the architecture differently and with different data access patterns. The applications finally chosen for inclusion in this report were those analyzed in the greatest detail. In the following section we summarize the results from the individual applications: a dense matrix multiplication kernel, synthetic aperture radar, plasma PIC, and volume rendering.

3.2 Summary of Benchmark Applications Analysis

In this section we present a summary of each analysis, including a brief description of the application, its principal computational elements, a snapshot of its equilibrium resource utilization, and an estimate of the largest job size that will run on the petascale HTMT.

3.2.1 Dense Matrix Multiply

Matrix multiplication plays a key role in many applications, and so we are interested in whether it poses any particular problems for the HTMT architecture and program execution model. Following is a brief summary of the analysis of an implementation of Cannon's algorithm for dense matrix multiplication reported by Amaral et al. (1999). In the basic algorithm, two large matrices (call them A and B) to be multiplied (giving C) are first partitioned into t^2 blocks each. Each of the matrices is M by M, where $M = t * s * b_c$. The final result requires calculating the products of blocks from A with blocks from B. Cannon's algorithm is used to compute this intermediate result, the product of two blocks, wherein each is partitioned into s^2 subblocks (each b_c by b_c elements in size) and distributed among s^2 processing elements (the SPELLs) in a special initial pattern. The processors are imagined to be arranged in an s by s grid with toroidally wrapped communication paths. The algorithm prescribes a data exchange pattern wherein the subblocks from A are moved to the left on the grid and the subblocks from B are moved up between subblock multiplications. At the end of s iterations, each processor has accumulated a subblock of the final result for the current block multiplication. For each subblock of C to be computed, t of these A and B subblocks are multiplied and accumulated. By the end, this kernel operation, the subblock multiplication, is performed t^3 times to compute the final t^2 blocks of C.

The b_c by b_c element subblocks are limited in size by available CRAM. The value of s^2 is chosen to match the total number of SPELLs available, 4096 for the full HTMT design. And the value of t^2 is set by the requirement that 3 bundles of t^2 subblocks fit in SRAM. This sets the natural limit to the size, M, of the matrix that can be multiplied with this method. As Amaral et al. (1999) point out, the algorithm can be scaled to larger matrices by breaking them down into this natural size.

This application fragment differs from the others analysed in that it introduces substantial interparcel communication between SPELLs over the CNET. The data exchange step described above is carried out over the CNET. Unlike the other applications, the s^2 SPELLs are all running threads that are synchronized without communicating to the SPIMs for the s iterations required to compute the block product.

Also interesting, in this implementation the t subblock multiplications needed for a single C subblock are accumulated in the CRAM, meaning that there is a further unusual dependency: t generations of subblock pairs must be percolated into the CRAM sequentially while code and data remain *live*. The parcels sent to the SPELL in this application are not nonblocking. The execution time was estimated to be

$$T = D_t[A] + D_t[B] + 2*IP_{DS} + t^2 * T_B + OP_{SD} + D_t[C]$$

$$T_B = t * s * M_S + (2t + 1) * IP_{SC} .$$

The first equation describes the general structure of the calculation. The $D_t[\ ]$ terms represent the transformation on the matrices that must be performed to optimally match them to the streaming communication pattern of the actual multiplication. These are nonoverlapping transformations carried out by the DPIMs. The third and fifth terms represent the percolation of the data between DPIM and SPIM. And the fourth term describes the t^2 block sets that must be injected into the CRAM and multiplied by the SPELLs. The details of that process are shown in the second equation, including the cost of the multiplication and the communication terms, in that order. The parameter descriptions and their estimated values are collected in Table 2. The total time for data transformation, done in the DPIMs, is 2.68 seconds. The total time spent multiplying subblocks in the SPELLs, given by $t^3 * s * M_S$, is 13.2 seconds. The remaining total time figures for different portions of the execution are in the last column of Table 3.

The results from the analysis of the model are organized into parcels in Table 3. Before the matrices can be efficiently broken into parcels for the multiplication described above, they must undergo a transformation. This is estimated to take 1.33 seconds for each matrix. With both A and B transformed, we then begin moving a coordinated stream of subblock parcels up to the SPELLs to be multiplied. The *Bundle Up* parcel requires 13.2 μsec to move subblocks for both of the multipliers; this operation is performed in parallel across the entire machine to install all of the subblocks (two per SPELL) needed to compute the product of a full block. For each of these, the *Block Multiply* parcel iterates s times on subblock multiplications (costing 11.7 μsec each) interleaved with CNET-mediated subblock exchanges (costing 0.94 μsec each). At the end of this 808 μsec the

result is sent back down to the DPIM (*Bundle Down*) to be transformed into normal order.

Table 2 Parameters and calculated values for the Cannon algorithm dense matrix multiplication.

A data transform in DRAM	$D_t[A]$	1.37 G cycles	1.33 seconds
B data transform in DRAM	$D_t[B]$	1.37 G cycles	1.33 seconds
C data transform in DRAM	$D_t[C]$	21.1 M cycles	20.5 E-03 seconds
Percolation of block D -> S RAM	IP_{DS}	10.6 M W	10.3 E-03 seconds
Percolation of subblock S->C RAM	IP_{SC}	15.6 K W	6.6 E-06 seconds
Subblock multiply	Ms	250. M cycles	11.7 E-06 seconds
Subblock exchange among SPELLs	Ds	W	0.94 E-06 seconds
Percolation of C subblock C->S RAM	OP_{CS}	15.6 K W	6.6 E-06 seconds
Percolation of C block S->D RAM	OP_{SD}	10.6 M W	10.3 E-03 seconds
Matrix partition factor (t^2 blocks)	t		26
Block partition factor (s^2 subblocks)	s		64
Subblock dimension	b_c		125
Matrix dimension (= t * s * b_c)	M		208000

The utilization numbers in the second half of Table 3 are normalized to the principal resource used by each parcel. For example, the *Block Multiply* parcel takes 750 μsec of SPELL execution time. Percolation of the parcel from SRAM to CRAM takes 13.2 μsec, or 1.8 % of the SPELL execution time. The resources are normalized in this way on a parcel-by-parcel basis. This view tells us the how resource use is balanced within a parcel. When weighted by the total execution time of the parcel and the number of times it is repeated, it tells us how system resources are distributed among the ensemble of parcels. Finally, it gives us a quick way to see how the parcels will pipeline.

The *Block Multiply* parcels require 13.2 seconds to execute (product of parcel time and parcel reps) and dominates the overall execution of the algorithm. Both CRAM and DRAM are well used. The only other resource that sees significant use is the DPIM execution of the matrix *Transform* parcels, contributing a total of 2.7 seconds, which does not overlap with the *Block Multiply*. DRAM is not heavily subscribed, leading to the possibility of performing several matrix multiplications in pipelined fashion, overlapping the transformation of one with the block multiplies of another to approach 100% utilization of the SPELL.

Table 3 Parcel summary for the Cannon algorithm. The top half of the table gives the fundamental resources required for each parcel in terms of processor cycles, words to be moved, and words of storage. The bottom half restates these resources in terms of the parameters of the architecture to give execution time and communication time. (An entire subblock is percolated out of the CRAM only after t=26 executions of the parcel have accumulated the result – the value here is the entire subblock amortized over the t=26 executions of the basic block multiply.)

Parcel Summary		Transform A	Transform B	Bundle Up	Block Multiply	Bundle Down	Transform C	
Parcel Time		1.33 s	1.33 s	21. ms	750 μs	10.5 ms	20.5 ms	Seconds
Parcel Reps		1	1	1	17.6 K	1	1	Reps
Processor	SPELL				250. M			Cycles
	SPIM							Cycles
	DPIM	1.37 G	1.37 G				21.1 M	Cycles
Communication	S => C				15.6 K			W
	S <= C				600*			W
	S => S							W
	D => S			21.1 M				W
	D <= S					10.6 M		W
	D => D							W
	H => D							W
	H <= D							W
Memory	CRAM				65.5 K			W
	SRAM			125. K		125. K		W
	DRAM	14.1 M	14.1 M				14.1 M	W
	HRAM							W
Processor Utilization	SPELL				100 %			13.2 s
	SPIM							
	DPIM	100 %	100 %				100 %	2.7 s
Bandwidth Utilization	S => C				1.8 %			0.230 s
	S <= C				0.03 %			0.004 s
	S => S							
	D => S			100 %				0.021 s
	D <= S					100 %		0.010 s
	D => D							
	H => D							
	H <= D							
Memory Utilization	CRAM				49%			
	SRAM			99 %		99 %		
	DRAM	2.8%	2.8%				2.8%	
	HRAM							

3.2.2 Synthetic Aperture Radar

The results presented here are derived from the analysis and report by Siegel and Craymer (1999). With a technique known as synthetic aperture radar (SAR), Earth orbiting radar instrumentation can be processed to extract extremely detailed relief

images of the Earth's surface using wavelengths that are relatively insensitive to water vapor in the atmosphere. Furthermore, because phase information is maintained at these relatively long wavelengths, images from successive orbits can be combined to form difference images sufficiently accurate to shed light on the effects of seismic activity. The return signal from a transmitted chirp is analyzed in the SAR algorithm/

3.2.2.1 Performance Analysis of SAR on HTMT

The algorithm proceeds in the following general way. The basic unit of work in the SAR calculation is processing a patch of data, a series of 1-D return signals each contributing a line to the 2-D patch. The patch computation is decomposed into two phases called range focusing and azimuth focusing. The phases are quite similar in general construction, beginning with per patch initialization computations followed by serial processing of lines.

Figure 3, presented above in Section 2.3.2, sketches the key features of the algorithm in terms of its decomposition into phases and parcels. Salient facts include the following. Reductions of data from separate patches are independent and are carried out in parallel. Range line computations within a patch can be carried out in parallel. Azimuth line computations for a given patch cannot begin until all of the range line computations have completed. Parcels are assembled by the DPIM in the DRAM. A typical patch is 11,812 x 4096 points.

The SAR calculation comprises six basic parcels (Figure 3): one to stage raw patch data from HRAM into DRAM, two each for each of the two phases of the computation, and one that posts the resulting patch image to HRAM. There is one instance of the range reference parcel for each 11,812 instances of a range line parcel. These are followed by one instance of the azimuth reference parcel and 4096 instances of the azimuth line parcel. Other miscellaneous parts of the computation are neglected in this analysis.

A few notes about the model:

1. If the SPELL computation takes a lot of time compared with communication and computational work at lower levels, then the system overhead will probably not be critical in our model. In particular, the model doesn't use many SPIM cycles explicitly, so we relegate the SPIM layer to a "pass through" role and ignore it.
2. In this application the Range Reference parcel provides data that is needed by each of the Range Line parcels. The reference data is left in CRAM while Range Line parcels are fed to the SPELL to use it. In some sense the Range Reference parcel has been dispatched to the SPELL without its closure (i.e., the rest of the Range Line data).
3. The table lookup and conversion of raw bytes into complex numbers are currently modeled as part of the Read Patch operation. The process could be done on a line-by-line basis as part of the line parcel work to greatly decrease the time to gather data as the raw data is 16 times smaller than the formatted data.

Execution time is modeled by a combination of two terms for computation in the SPELL and one term for work done in the DPIM. Table 4 gives the model coefficients for the computational work done in each parcel type. As an example, the time spent executing a Range Line parcel in the SPELL is modeled as

$$T_{EXECUTE} = T_{SPELL} * (6 * N_{RANGE} + 10 * N_{RANGE}' * \log_2(N_{RANGE}')).$$

Table 4 Shorthand for the models describing principal compute cycles for each of the SAR parcel types. *Nr* and *Na* are the number of elements in the range and azimuth directions, respectively. *Ni* represents one of these, depending on context. The primed quantities in the log term indicate that the number, *Ni*, must be padded to the next highest power of two.

SAR Execution Coefficients	Read Patch	Range Reference	Range Line	Azimuth Reference	Azimuth Line	Write Results
SPELL						
Ni		22	6	28	6	
*Ni' * log2(Ni')*		5	10	5	10	
DPIM						
*Nr * Na*	10					10

The time required to assemble, gather, execute, scatter, and dispense each of the six parcel types is computed along similar lines. Table 5 summarizes these estimates in a way intended to highlight bottlenecks and overall distribution of resources. For each parcel type the time required to carry out the execution phase of its progress through the percolation is listed in the first line. In the case of the Range Line parcel, this is the time it spends in the SPELL. The total time for a single SPELL to calculate a patch of 4096 range lines and 11,812 azimuth lines is approximately 62 msec, giving an execution rate of 66,000 patches per second for a 4096-slice HTMT system.

The remaining rows show the resource utilization. The computing utilization, typically in the SPELL, is defined as 100%. The communication utilization factors are expressed as the fraction of the available communication bandwidth used to deliver the data in the time it takes to execute the parcel in the SPELL. Anything below 100% is interpreted as meaning there is bandwidth to spare. The memory used is expressed as a fraction of the memory available.

What is the largest dataset we can process in this way? CRAM utilization is determined by the need to keep at least one thread running, another ready to run, and the always resident data computed in the *reference* parcel – *Range Line* parcels dominate the computation. Siegel and Craymer calculate that each 11812 x 4096 point patch of input data requires 92 MB of storage (2 bytes per data value). If half of the available HRAM is used to hold input data, then the scale of our job is set at 5.4 million raw patches; we will use 50% utilization of the HRAM in our summary in Table 5. At 66,000 patches per second, it will take 82 seconds to reduce this stash of patches, which represents a rate into the DPIM of raw patch date of 5.8 TB/sec. Table 5 shows that the flow of data to the SPELLs for processing can be sustained by using very little of the available SRAM,

DRAM, and HRAM. This reassures us that the data can be staged out of HRAM as needed.

Table 5 Summary of the execution time and relative resource utilization factors for each of the six parcels that make up the basic SAR application.

PARCEL SUMMARY		*READ PATCH*	*RANGE REFERENCE*	*RANGE LINE*	*AZIMUTH REFERENCE*	*AZIMUTH LINE*	*WRITE PATCH*	
Parcel Execution Time		1900 μs	5.5 μs	9.3 μs	1.4 μs	2.0 μs	1900 μs	
Repetitions per Patch		1	1	4096	1	11812	1	
Processor Work	**SPELL**		1.41 M	2.39 M	360 K	516 K		Cycles
	SPIM							Cycles
	DPIM	484 M					484 M	Cycles
Communication Load	**S => C**		32.8 K	32.8 K	8.2 K	8.2 K		W
	S <= C			32.8 K		8.2 K		W
	S => S							W
	D => S		23.6 K	23.6 K	8.2 K	8.2 K		W
	D <= S			32.8 K		8.2 K		W
	D => D							W
	H => D	48 M					48 M	W
	H <= D							W
Memory Requirements	**CRAM**		524 K	262 K	131 K	66 K		B
	SRAM		189 K	189 K	66 K	66 K		B
	DRAM	774 M	189 K	189 K	66 K	66 K	774 M	B
	HRAM	96 M					96 M	B
Processor Utilization	**SPELL**		100 %	100 %	100 %	100 %		
	SPIM							
	DPIM	100 %					100 %	
Bandwidth Utilization	**S => C**		9.3 %	5.5 %	9.1 %	6.3 %		
	S <= C			5.5 %		6.3 %		
	S => S							
	D => S		43 %	25 %	58 %	41 %		
	D <= S			35 %		41 %		
	D => D							
	H => D	51 %						
	H <= D						51 %	
Memory Utilization	**CRAM**		51 %	26 %	13 %	6.4 %		
	SRAM		.07 %	.07 %	.03 %	.03 %		
	DRAM	19 %	.005 %	.005 %	.002 %	.002 %	19 %	
	HRAM	.04 %					.04 %	

3.2.3 Plasma PIC

The plasma PIC code simulates the interaction of millions of charged particles with the electromagnetic field that they produce. It is an example of a code that is used to understand the behavior of a plasma such as that generated by proposed fusion reactors. The analysis described here is paraphrased from Norton, Decyk, and Cwik (1999).

3.2.3.1 Classical MPP Algorithm for Plasma PIC

Figure 2, shown in Section 2.3.1, depicts the general steps in the plasma PIC algorithm as implemented for a distributed-memory machine. The application loop alternates between computing fields based on current charged particle positions (and velocities) and advancing the positions (and velocities) of these particles under the force of this field. These can be thought of as the two principal phases in the application. Between these two phases the problem data is repartitioned to improve the efficiency of each. The repartitioning is accomplished by data exchange, called guard cell exchange in the figure, between the processors of the MPP implementation.

Particle push, as the position (and velocity) update calculation is called, is the most computationally expensive phase of the algorithm. The analysis focuses on this phase.

For each processor in the MPP implementation, the problem is laid out on a grid of $(N_G+1)^3$ field points, with an associated field (E_X, E_Y, E_Z) at each point. These grid points partition the volume into $(N_G+1)^3$ cells with an average of N_P particles per cell. The memory required is $(3 + 6 * N_P) * N_G^3$ words per processor.

By using a quadratic interpolation of the electrostatic field, the approximate number of operations is extracted from an existing implementation and found to be 200 floating-point operations per particle. The model discussed here accounts for additional work in the form of integer operations, loads, and stores, by adjusting this figure by an empirically determined floating-point utilization factor of 88% to correct for the instruction mix. Hence the net time to carry out the particle push for a single particle is taken to be $(200 * t_{FLOAT} / .88)$.

3.2.3.2 Performance Analysis of Plasma PIC on HTMT

Analysis of the performance of the plasma PIC code on the HTMT begins by mapping the computation of the particle push onto the SPELL and accounting for the data motion through the memory hierarchy. The values of N_G and N_P are tuned to fit CRAM. If we use the typical value of $N_P = 16$, N_G becomes 10, for a parcel size of 100K words. We first consider a single slice of the HTMT architecture, topped by a single SPELL cryogenic processor.

Table 6 Summary of parcel processing times for plasma PIC particle push parcel.

Parcel Phase	Quantity		Time	Available Parallelism per Slice	Utilization
Assembly (DPIM)	140. K	cycles	281. μsec	128	15 %
DRAM → SRAM (DV)	101. K	W	10.1 μsec	1	71 %
Dispatch (SPIM)	37.9 K	cycles	37.9 μsec	64	4 %
SRAM → CRAM (RTI)	101. K	W	1.58 μsec	1	11 %
SPELL Execution Time	3.64 M	cycles	14.2 μsec	1	100 %

This model defines a single parcel. The progress of the parcel as it is assembled in the DPIM layer, transported up the memory hierarchy, enqueued, and finally executed in the SPELL is summarized in Table 6. The analysis presented here differs from that of the original analysis (Norton, Decyk, and Cwik 1999) in that the values used for number of DPIM and SPIM nodes per slice are higher, in keeping with more recently available parameters from the HTMT design. Another view of this parcel migration and execution is shown in the pipeline timing diagram (Fig. 4). With the parameters presented here, the parallelism available in the DPIM and SPIM layers easily hides the corresponding latency.

To compare this analysis to the other applications we had to adjust model parameters. In particular, we expressed the assembly and dispatch time in terms of cycles and used the DPIM, SPIM, and SPELL cycle times in Table 1. We also used DPIM and SPIM node counts that differed from those used in the original analysis.

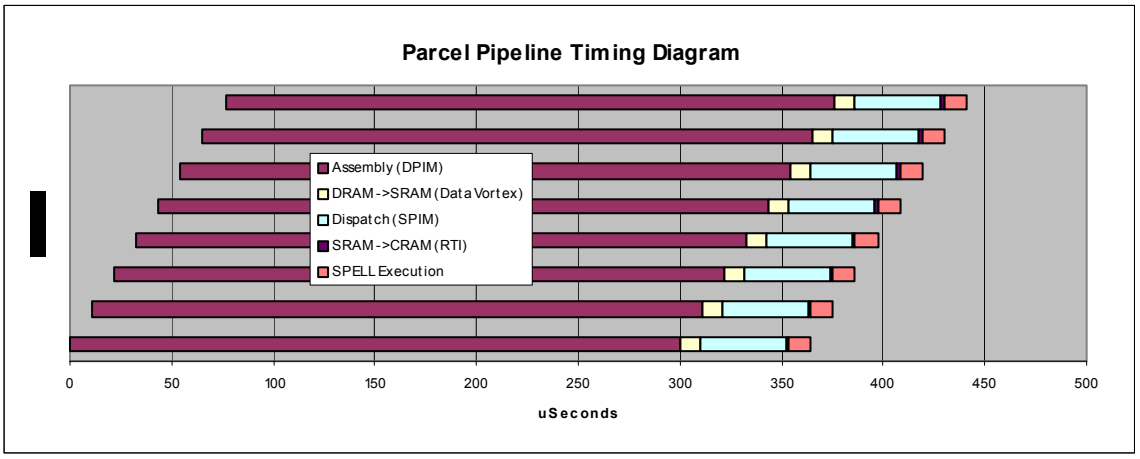

Figure 4 The parcel pipeline timing diagram for the plasma PIC application feeding a single SPELL. The parcel phases are taken in order from Table 6. The SPELL is kept busy by a sufficiently infinite stream of parcels supplied by the DPIM and SPIM layers.

We determined the subscription rates of the various resources in the architecture. First, for each of the five phases in the parcel pipeline we note the additional parallelism available. For example, the 281 μsec required for parcel assembly in the DPIM used a single processor out of the 128 available. These available parallelism factors are included in Table 6.

To compute resource utilization, we normalized the time by the available parallelism factor. Normalized assembly time becomes 2.20 μsec; normalized dispatch time becomes 0.59 μsec. The largest of these overlapable parcel phase times is the normalized SPELL execution time, 14.2 μsec, meaning that the DPIM and SPIM are able to build and feed

parcels fast enough to keep the SPELLs busy. The resource utilization is simply the ratio of the normalized time and this largest normalized time, shown in the table.

We emphasize that this normalization does not imply that we will actually spread the work for a single parcel across the available parallel resources at a given architectural layer. It assumes only that enough parcels will be in the pipeline to keep these resources sufficiently busy.

Table 7 Summary of the plasma PIC problem size estimate. For a problem that must be held in DRAM without paging to and from HRAM, the problem size is limited as shown. To cycle once through all parcels in this dataset will take approximately 72 msec. A typical simulation might run for 10,000 time steps.

	Slice (single SPELL)	**Full HTMT**
Parcel rate	70 K/sec	288 M/sec
Particle rate	1.4 G/sec	5.9 T/sec
DRAM limited:		
Particle Push Parcels	5 K	20 M
Particles	80 M	330 G
Field points	5 M	20 G
Iteration time	72 msec	

Finally, scaling the one-slice analysis up to the full architecture, we can estimate the problem size accessible to this architecture. A single SPELL can compute approximately 70,000 parcels per second. The entire machine, configured with 4096 slices, can therefore compute 288 M parcels per second. We estimate that the total DRAM in a slice can manage of order 5,000 parcels for this problem. Each parcel represents 16,000 particles and 1,000 grid points.

Summarizing salient results from Plasma PIC analysis:

- SPELLs will be 100% busy. For this code, with 88% of the inner loop devoted to floating-point operations, this means that we will sustain 88% of peak FLOP rate.
- A simulation of 330 billion particles on a grid of 20 billion field values could be be run using available DRAM. A 10,000-step run would take approximately 720 seconds.
- The assumptions in the analysis presented in this report result in a sustained performance of approximately 0.9 PFLOP per second for this application. The authors of the original application report came to the more conservative final performance estimate of 0.3 PFLOP per second. This discrepancy owes to our use of increased final DPIM and SPIM node counts per slice.

3.2.4 Volume Rendering

The term volume rendering refers to a class of visualization techniques that present 3-D datasets in ways that illuminate the internal structure of the data (see, for example, Lichtenbelt, Crane, and Naqui 1998). These often involve some form of transparency. In the algorithm described here, the data consist of values on a regular 3-D grid of points. These may represent material density given by an MRI of a human head, the temperature of the fluid in a simulation of the interior of a star, the pressure of the atmosphere in a simulation of global weather, or any such scalar field in any number of measurement or simulation domains. Volume rendering in general need not be limited to scalar fields, but for this algorithm we consider only these sorts of data set.

```
# ------------------------------------------------------------------------------------------------------------------------
# volume rendering pseudocode
# ------------------------------------------------------------------------------------------------------------------------

foreach $pixel in ($plane)                          % 2-D pixel loop
      $frustum = f ($pixel, $plane, $viewpoint)     % compute pixel frustum from viewpoint
      $value = 0                                    % initialize pixel value to zero
      $opacity = 1                                  % initialize opacity
      foreach $voxel along ($frustum)               % march outward along frustum
            $local = interpolate($data, $voxel)     % interpolate value of data at center of voxel
            $value = $value + $local/$opacity       % add to accumulating pixel value
            $opacity = darken($opacity)             % increase opacity factor
            if ( $opacity > $threshold ) exit       % compare opacity factor to threshold and exit
      end                                           % end voxel loop
      image_plane($pixel) = $value                  % save accumulated value in current pixel
end                                                 % end pixel loop
```

Figure 5 Summary of the serial rendering algorithm in pseudocode.

3.2.4.1 Serial Algorithm for Volume Rendering

The approach taken in the serial volume-rendering algorithm is to cast rays through the data volume, one for each pixel to be rendered. With each ray is associated a frustum of included volume to be projected in some way onto the corresponding pixel in the image plane. In the case of orthonormal projection, these frusta become square tubes. Intensity is accumulated as the ray is traversed outward from the viewpoint and through the data, enabling a number of effects in which contributions from data within the volume are manifested visually in the final 2-D rendering. The algorithm accounts for arbitrary viewpoint. It is parameterized by an opacity factor that weights contributions in the foreground more heavily than those in the distance. A threshold test is included to allow early termination of the ray traversal if a test of the opacity determines that the data beyond will make no perceivable contribution or if the accumulated intensity is already at the maximum.

Here is a quick summary of the algorithm as it might be implemented on a serial architecture (pseudocode in Figure 5). The outer loop of the algorithm iterates over the 2-D array of pixel positions in the image being rendered. For each of these pixels a frustum is cast from the viewpoint through the image plane. The data in the frustum can

contribute to the computed pixel value. The inner loop walks outward from the image plane along the frustum centerline. At each step of this loop the data is interpolated and scaled to the voxel volume to compute the local contribution to the pixel value. This local contribution is adjusted by the present value of opacity before being accumulated into the computed pixel value. If the sum reaches the allowed maximum, then the inner loop is terminated. The opacity factor is increased at each step. If the opacity rises above a threshold parameter the inner loop is terminated early.

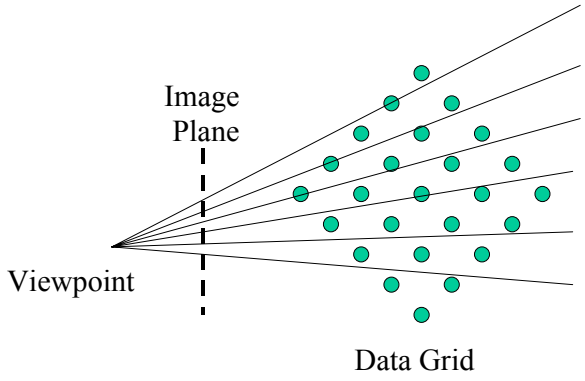

Figure 6 Schematic showing rays cast through data.

Of interest in the algorithm as presented here are bilinear interpolation, inclusion of the cutoff threshold, orientation of the regular data grid, the scale change as a result expanding volume of the frustum, and ray fragment management.

Of particular interest is the fact that in this application the execution time (or time to render a frame) depends on the data, the viewpoint, and the rendering parameters (such as opacity). Consequently an estimate of the performance (for example, in terms of resource utilization) depends strongly on these conditions of the computation.

3.2.4.2 Performance Analysis of Volume Rendering on HTMT

In a classical MPP algorithm one might assign a thread to each ray, particularly if the algorithm were implemented on a shared-memory machine. This approach is not possible on the HTMT, if only because of the space limitations within the CRAM.

In this algorithm, the main parcel for the computation will be composed of a ray list and a 3-D subcube of the data. The DPIM will manage the data partitioning. The SPIM will manage ray accounting, orchestrate ordered gather of data from DPIM, dispatch parcels to SPELL, and scatter image plane results. Here is an outline of that process:

1. *Viewpoint Broadcast*. Information is broadcast to DPIMs, including the image plane information necessary to generate ray data.

2. *Assemble.* Each DPIM calculates for the voxels that it owns: the subcubes it needs to contribute, front-to-back ordering information based on viewpoint, and the rays passing through the subcube.

3. *Gather.* Each SPIM gathers from the DPIMs the subcubes oriented in line with the viewpoint.

4. *Dispatch & Inject.* SPIM dispatches to the SPELLs the subcubes in front-to-back order.

5. *SPELL Execution.* SPELL calculates the ray casting through the subcube: lighting, voxel properties, interpolation, and accumulate.

6. *Eject.* SPELL returns intermediate calculation of ray value accumulation and termination information.

7. *Retire.* SPIM holds the ray accumulation buffer for next subcube and adjusts ray tables to steer subcube dispatches. When all rays in the bundle have terminated, front-to-back subcube dispatch along this bundle terminates.

8. *Scatter.* SPIM composites and outputs the image to the Data Vortex. Because image patch is not ready until parcel processing along a ray bundle has terminated, this operation does not happen for every parcel.

Our estimate of the time it takes to execute the principal volume rendering parcel reduces to the following model:

$$T_{EXEC} = \alpha * N_{CYCLES} * N_G^3 * t_{SPELL}.$$

From a bilinear interpolation kernel we find that N_{CYCLES} is at least 22 – this is a lower bound on the number of cycles required to compute the contribution of a voxel to the accumulating ray value. We are using a value of 44. The factor α accounts for three effects: the number of rays being traversed, the effects of ray termination within the subcube, and scale changes in expanding arrays. It is a function of the data being visualized, rendering viewpoint, and opacity; we estimate it to be typically in the range 0.1 to 0.5.

Using the following estimation of the time needed to move the parcel into the CRAM, one can find the portion of parameter space (if any) for which it is cost effective to execute the parcel:

$$T_{MOVE} = (W_M + 6 * N_R + 0.5 * N_G^3) * t_{S\ to\ C}.$$

The first term is the number of words of executable code in the parcel, the second is the space taken by ray data, and the third is the space taken by the subcube itself. It does not take a very large value of N_G for the third term to dominate: probably between 10 and 100.

Table 8 Summary of parcel processing times for the volume rendering application.

Parcel Phase	Principal Resource	Work / Volume		Time	Avail. Parallel	Utilization
Assemble	DPIM	99,500	cycles	191 μsec	128	17 %
Gather	DV (D → S)	80,500	W	8 μsec	1	94 %
Dispatch	SPIM	25,000	cycles	25 μsec	64	4.5 %
Inject	RTI (S → C)	80,500	W	1.3 μsec	1	15 %
SPELL Execution	SPELL	2,200,000	cycles	8.8 μsec	1	100 %
Eject	RTI (C → S)	15,000	W	0.2 μsec	1	2.7 %
Retire	SPIM	25,000	cycles	25 μsec	64	4.5 %
Scatter	DV (S → D)	75	W	0.008 μsec	1	0.1 %

One concern is whether the cost to move a parcel into the SPELL is less than the time it takes for the SPELL to execute that parcel. In other words, is there sufficient reuse? We hope that $T_{MOVE} < T_{EXEC}$. For parcel size dominated by subcube data, this condition reduces to the following:

$$\alpha * N_{CYCLES} > 0.5 * t_{S\ to\ C} / t_{SPELL}.$$

For the parameters of our HTMT model, the right-hand side is approximately 2. With $N_{CYCLES} > 22$, and $\alpha > 0.1$, we conclude that we can expect the volume rendering parcels will use the computing resource of the SPELL effectively. On the other hand, the same parcel must be passed from the DPIM to the SPIM through the data vortex. For the analogous relation (replacing $t_{S\ to\ C}$ with $t_{D\ to\ S}$) the right-hand side becomes approximately 13, which places the burden on the DPIM to supply parcels fast enough to keep the SPELL busy.

For nominal values of the model parameters, the principal parcel of this application can be summarized by the time spent at the various layers in the architecture (Table 8).

We can estimate the approximate size of the largest rendering job that the HTMT could support. At 8.8 msec per parcel, the 4096 SPELLs can process almost 500 million parcels per second. And at 24 frames per second, this corresponds to about 2.4 teravoxels per frame, or 180 megapixels per frame. If the rendered pixels fed an 8 foot by 7 foot display with 150 dpi resolution, we could interactively navigate through a volume-rendered dataset that was approximately 1.2 terawords (3 bytes per grid point).

3.3 Summary Evaluation of Application Performance

Table 9 summarizes the equilibrium resource utilization for our four applications. As hoped for, the applications chosen exhibit a range of demands on the available resources.

The Cannon algorithm and the SAR application involve many parcels. The values tabulated in Table 9 represent a suitable average of the individual parcels. In the case of

the SAR application, for example, we computed the aggregate utilization by weighting the individual parcel resource values with the time spent as a fraction of the total SPELL time. Note that the resource usage for the *Read Patch* and *Write Patch* parcels is almost entirely nonoverlapping with other parcel resource usage, and so these can be executed in parallel with the parcels that use SPELL time. For each *Read Patch* parcel executed, we are using 4096 *Range Line* and 11812 *Azimuth Line* parcel, the typical patch size.

To summarize our impressions based on the applications analyzed, we revisit the questions posed in introducing the evaluation framework:

Will the application perform well? All of the applications summarized here come close to extracting petaops performance out of the SPELL layer of the architecture. In this sense, the applications perform well, within the limitations of the present analysis. The volume-rendering application stands out here as an example of an application on the edge. Its performance depends not only on the details of the data being visualizes but also on the particular viewpoint and visualization parameters selected by the user. As an interactive application (assuming the appropriate interfaces to a display device), the performance of the application can change.

Table 9 Equilibrium HTMT resource utilization for applications in our suite.

Application Summary		Matrix Multiply	SAR	Plasma PIC	Volume Rendering
	Parcel Time	750 μsec	60 msec	55 msec	8.6 μsec
	HTMT Rate		15 μsec		24 fps
Execution Time	Total Time	16 sec	82 sec		forever
Processor Utilization	**SPELL**	83 %	100%	100%	100%
	SPIM			6%	9%
	DPIM	17 %	6%	22%	17%
Comm. Utilization	**S => C**	1 %	6%	14%	15%
	S <= C		6%		2.7%
	S => S				
	D => S		31%	92%	94%
	D <= S		37%		0.1%
	D => D				
	H => D		2%		
	H <= D		2%		
Memory Utilization	**CRAM**	49 %	19%	78%	59%
	SRAM	99 %		100%	4%
	DRAM	8 %	1%	100%	75%
	HRAM		50%		

What system parameters are responsible for the performance? The SPELL cycle time and the bandwidth of the data vortex are the highest-level parameters that shape or limit the performance of these applications. Details of the performance of the SPELL pipeline

were not tested here and quite probably will further restrict the performance of our applications as more realistic account is taken of instruction mix and pipeline resource contention. We note that for all of the applications the problem decomposition began by scaling to available CRAM, usually imposing an uncomfortably tight constraint. This implies to us that increasing CRAM should be considered.

Is the system balanced from an applications viewpoint? In several ways, the answer to this question seems to be no. For at least some applications (plasma PIC and volume rendering), the data vortex bandwidth is uncomfortably close to limiting our ability to keep the SPELLs profitably occupied. On the other hand, half of the applications did not make much use of the available memory in the DPIMs and particularly in the SPIMs – though we note that this may reflect economies of implementation in these hand-worked analyses. Perhaps the most striking imbalance is the underutilization of the processing capacity in the PIM layers. This is not surprising and may simply reflect a limitation in tools to help application programmers first explore and then use this resource. We note that the four fibers providing input to the SPIM unit might not provide much relief unless matching capacity from the DPIM unit is added, since they cannot be used in an equilibrium sense.

Do we understand the execution model and its performance? No, probably not. Even the small suite reported on here exhibits a wide range of interesting details in their use of parcels to express algorithm decomposition and data motion. We note that two of our applications (SAR and Cannon) injected parcels into the SPELLs that did not in themselves include all of the data needed to execute to completion – striking departures from the design philosophy of the HTMT parcel percolation model.

What tools are needed to improve performance analysis and prediction? Several tools will help us improve on this analysis. First, detailed simulators of the individual components with cycle-level accuracy will enable us to understand how to model instruction mix effects on performance. This capability exists (at some level) for all or part of the SPELL but was not in wide use for these studies, nor were simulators available for the PIM parts. These will enable microkernel performance characterizations that can be used as a basis for refined models of application components. Second, a compiler or simulation tool that will allow us to evaluate more detailed models of our applications would be of immense utility.

4 Evaluation of HTMT-C as a Tool

One of our tasks was to evaluate the current efficacy and future potential of HTMT-C as a tool. By HTMT-C we mean a language and compiler probably based on Threaded-C (Theobald et al. 1998; Tremblay et al. 2000), in turn derived from EARTH-C. As yet, no actual HTMT-C compiler exists. Threaded-C is most recently incarnated as a public release version 2.0. Some of its original salient features were that it

- supported fine-grained multithreading,
- offered detailed control of program execution by programmer in this environment,

- integrated with hardware synchronization of the EARTH machine architecture, and
- took advantage of 32-bit global address memory space of that architecture.

The first two on the list are of great interest to HTMT developers, while the others are less relevant, pertaining to particular aspects of another hardware architecture.

The language is based on C, supports threading, and on a finer grain supports what are called fibers. In the parlance of Threaded-C, fibers have several interesting properties: they are a kind of lightweight thread, they support rapid context switching, they are executed only when all required data are available, and they execute to completion. These properties and conditions are related to those placed on parcel execution in the SPELL and are in large part what makes Threaded-C an interesting starting point for HTMT-C.

4.1 Key Questions

At the risk of overlap in our analysis, we have divided some of our observations and concerns into a set of answers to questions one might ask about HTMT-C. We feel that these questions expose features and capabilities that will be important to the further development of the HTMT software and system.

Porting. Is HTMT-C useful in the process of porting codes to the HTMT? An undoubtedly important source of important applications for evaluation of the system will be existing codes; these may well dominate the scene for the early years.

> HTMT-C has several factors working in its favor in this connection. Its C roots make it a strong choice (Fortran likely edging it out) as a porting vector, since many codes and libraries exist that are C-like. The power of C to deliver almost machine-level control may also be of some help here.
>
> On the other hand, the notions of threads and fibers and the communication methods made available by Threaded-C are very foreign to notions captured in any existing code that we have come across. Any additions to the language that express the additional aspects of the HTMT notion of parcel percolation will only increase this distance. Furthermore, there do not appear to be any automatic remedies to these problems on the horizon.

System development. Is it a useful as a system development tool? For example, does it allow sufficient flexibility in a rapidly changing development environment, with aspects of the program execution model evolving constantly and with parameters and topology of the hardware also under constant modification?

> We suspect that as Threaded-C evolves support for HTMT architectural and program execution model features, it could become a powerful tool for system development.

Expressiveness and power. More generally, is it an appropriately expressive language? Does it enable painless access to the deepest aspects of the hardware architecture, control over the program execution? The end goal would be to enable (and even encourage) creation of high-performance applications.

> As has been noted, based on its inheritance from C, we believe that Threaded-C is powerful.
>
> Its expressiveness, a close kin to its power, is possibly another matter. While one can build efficient code that will use the architecture to good advantage, it might not be easy or natural to do so. This will depend in part on the details of the evolution of Threaded-C (or something) into HTMT-C. If suitable constructs are developed to enable the economic expression of computation and communication in terms of the HTMT parcel percolation model, then HTMT-C will have succeeded in part.

Simulation and emulation. Can HTMT-C play a useful role in simulation and/or emulation of the performance and behavior of the hardware and runtime systems while they are still on the drawing board?

> There may be an advantage to developers of simulation and emulation capability to use Threaded-C for its thread/fiber constructs. In particular the sync slot management might be very helpful in building object-oriented components. It is naturally parallel and hence automatically takes advantage of large computing farms to carry out what will undoubtedly be challenging simulations.

Extensibility. Probably a corollary to some or all of the above is the question: Is HTMT-C sufficiently extensible to handle the expected and the unforeseen requirements of the development process?

> This is not an extensible language in any of the modern senses of the word. The critical syntactic features of the language are part of the definition of the language imposed by the compiler. Although there are stylistic approaches to extending C, little about the language or any development environment encourages such extension. It may be as simple as the fact that compiled functional languages are not built to be tweaked.

Ease of use. In addition to the preceding functional questions, one might drive the evaluation of HTMT-C by asking: Is it easy to use, or natural, convenient, clean?

> The experience of the application performance analysis effort suggests that HTMT-C will not be easy to use. The learning curve for Threaded-C was fairly steep. Also, as yet, the mapping of Threaded-C constructs to HTMT constructs is not one-to-one. Programming an application using the parcel percolation model would be greatly simplified if there were objects in the language that naturally expressed concepts in the model – Threaded-C is not there yet.

4.2 Summation of HTMT-C Remarks

Strengths. HTMT-C has a partial incarnation already in Threaded-C. Because Threaded-C is under development, the development of HTMT-C could be steered. It does embody aspects of multithreading that are core to the HTMT architecture and programming model. There is power in its C heredity, conferring to the application and system programmers much potential for control.

Weaknesses. As yet, there is a great distance between the programming model expressed by Threaded-C and what is likely to be an efficient expressive engine for HTMT-C. It does not support or reflect any aspects of an HTMT execution model or runtime interface (beyond multi-threading). There are no apparent mechanisms to target the specific processing layers of HTMT: SPELL, SPIM, and DPIM.

Alternatives. No obvious alternatives are under development.

Comments. The developers of HTMT software might consider augmenting the notion of a solitary HTMT-C solution.

- As an alternative to the present HTMT-C, one might create a relatively simple library-based API to the important parcel construction and percolation. This could enable more direct expression of programs in terms of the program execution model than is currently provided – a shortcut to a higher level of interface and experimentation.
- One might incorporate high-level scripting tools from the start. They might provide a useful supplement HTMT-C and enable developers to quickly build applications.
- The compiler could be opened up a bit, with added support for macros and other features through precompiler interface available to developers. This might speed the coevolution of the language, architecture, and programming models.
- One could include the runtime itself as part of the development from the start: a runtime written in the same language of parcel percolation could have many advantages, not the least of which is flexibility (similar to Unix written in C, FORTH written in Forth, and so forth).

5 Final Comments

When we set out to include application performance analysis in the HTMT development process, we knew that this key task would be challenging. To understand with sufficient fidelity the effects of hardware and system software models on the final performance of real applications, we need

- detailed performance models for the individual subsystems under realistic loads,
- means for expressing key algorithmic components of applications, and
- multilevel composition of the application model pieces.

Some of the key elements in our application performance evaluation framework were beyond the scope of the present analysis. Some of these are laid out in the following paragraphs.

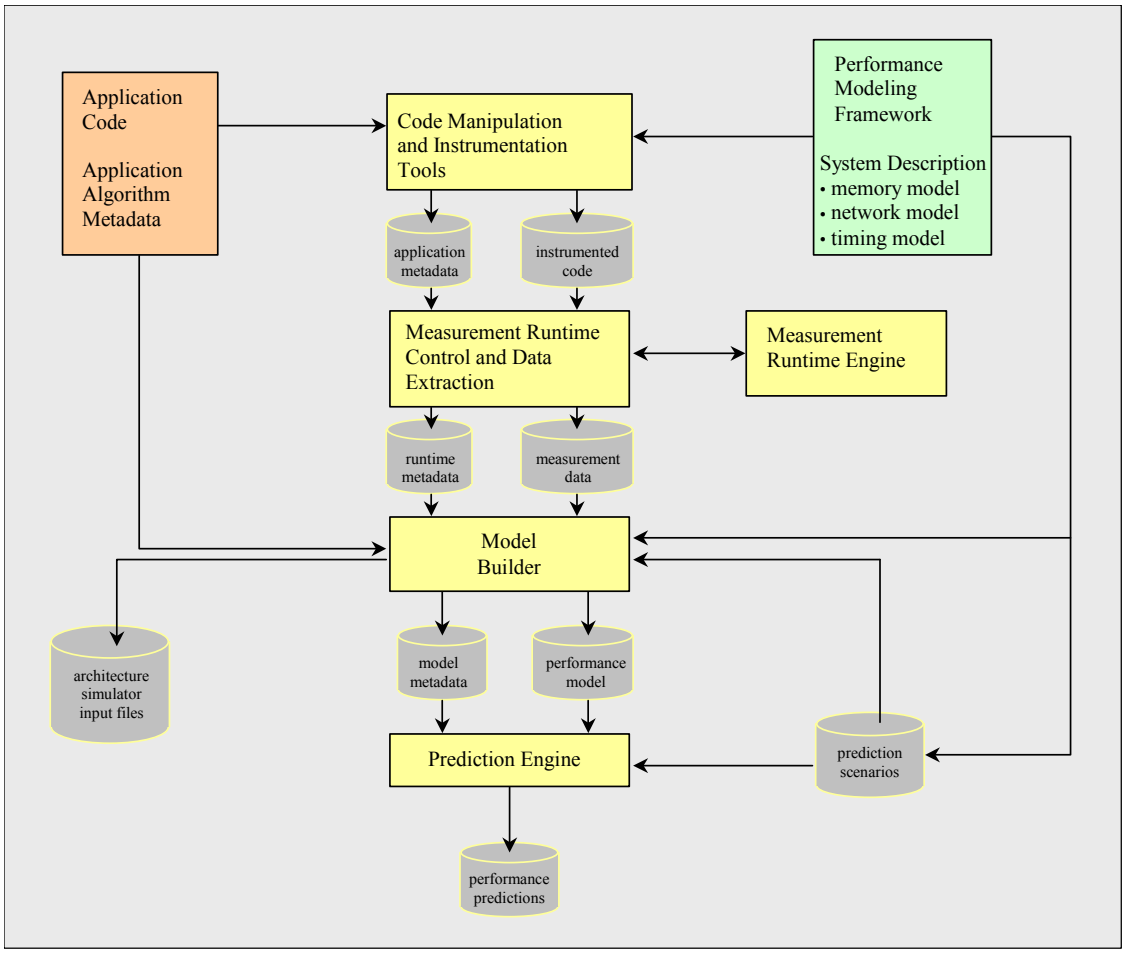

Figure 7 A model for a flexible application performance analysis system. It includes components for measurement of kernel performance, architecture and runtime description, and model construction.

In our view, three aspects of the problem need to be fully expressed and tested. First, the machine is described as subsystems that interact with one another over its connection topology. The program execution model shapes the ways in which the application interacts with the machine. And the application itself is a large object with algorithmic components that will interact with one another, with the application data structures, with the runtime system, and with the characteristics of the machine.

- *HTMT subsystem simulation, modeling, and analysis*. In the end we need applications-level performance models for each subsystem: SPELL, CNET, SPIM, Data Vortex, DPIM, and HRAM. These will be based on detailed simulations of a significant set of microkernels and will be represented by a

parameterized model of the subsystem in terms of, for example, instruction mixes. These will probe the individual subsystems as well as the aspects of the critical interfaces between them.

- *Integrated end-to-end modeling.* Microkernel and subsystem simulation will form the basis for modeling of entire applications via *composition*. An application will be broken down into pieces that map to these system components. An end-to-end model will be constructed by bolting these submodels together.

- *Program execution model performance evaluation*. In addition to aspects of the performance evaluation targeting the hardware subsystems, the program execution model will be represented. Its effects on performance will be manifest in the rules for decomposition of applications onto the PXM and the performance of the supporting components of the runtime.

- *Predictive models*. Each model is actually three models: serial kernel complexity, classical MPP, and multithreaded HTMT-C. Creating predictive models of the performance of the application on serial and classical MPP, which can then be tested on existing systems, is key to model verification on the unrealized HTMT architecture.

One possible structural implementation for these ideas is presented in Figure 7. With this implementation one could perform large-scale optimizations of end-to-end models, iterate, and converge ("mass balance") the models for internal self-consistency (e.g., Taylor et al. 2000; Taylor et al. 2002). With a reliable model of several applications in place, one would perform sensitivity analysis over range of system and applications parameters to help identify weak portions of the architecture or program execution model.

Acknowledgments

We acknowledge considerable contributions to the development and testing of this application analysis framework from the many people involved in the HTMT project, particularly those involved directly with the application analysis and software studies. These include Jay Brockman, Loring Craymer, Jose Nelson Amaral, Guang R. Gao, Bill Gropp, Phillip Merkey, Charles D. Norton, John Salmon, Herb Siegel, Thomas Sterling, and Kevin Theobald.

References

Amaral, Jose Nelson, Guang R. Gao, Phillip Merkey, Thomas Sterling, Zachary Ruiz, and Sean Ryan, "An HTMT Performance Prediction Case Study: Implementing Cannon's Dense Matrix Multiply Algorithm," CAPSL Technical Memo 26, February 17, 1999.

Arend, Mark, Keren Bergman, and Coke Reed, "Data Vortex Network Packaging," technical report, HTMT Tech Note #40, September 1, 1998.

Arend, Mark, Coke Reed, and Keren Bergman, "Physical Design and Specifications for the Data Vortex Network," technical report, HTMT Tech Note #33, 1998.

Bunyk, Paul, Mikhail Dorojevets, Konstantin Likharev, and Dmitry Zinoviev, "RSFQ Subsystem for HTMT PetaFLOPS Computing," technical report, HTMT Tech Note #18 and SUNY Technical Report #3, 1997.

Dorojevets, M., P. Bunyk, D. Zinoviev, and K. Likharev, "COOL-0: an RSFQ Subsystem Design for Petaflops Computing," IEEE Trans. on Appl. Supercond., June 1999, vol. 9(2), pp. 3606-3614.

Gao, Guang, Jose Nelson Amaral, Andres Marquez and Kevin Theobald, "A Refinement of the HTMT Program Execution Model," CAPSL Technical Memo 22, July 13, 1998.

Gao, G., K. Theobald, A. Marquez, and T. Sterling, "The HTMT Program Execution Model," University of Delaware, Department of Electrical and Computer Engineering, Computer Architecture and Parallel Systems Laboratory, CAPSL Technical Memo No. 9, July 18, 1997.

Kogge, Peter, Notre Dame University, "PIM Technology Projections for the HTMT Project," technical report, HTMT Tech Note #41, 1999.

Kogge, P., S. Bass, J. Brockman, D. Chen, and E. Sha, "Pursuing a Petaflop: Point Designs for 100TF Computers Using PIM Technologies," Frontiers of Massively Parallel Computation, Oct. 1996, also technical report, HTMT Tech Note #8, 1996.

Lichtenbelt, Barthold, Randy Crane, and Shaz Naqui, *Introduction to Volume Rendering*, Prentice Hall, 1998.

Liu, Wenhai, Ernest Chuang, and Dmitri Psaltis, "Holographic Memory Design for Petaflop Computing," technical report, HTMT Tech Note #22, July, 1998.

Norton, Charles D., Viktor K. Decyk, and Thomas A. Cwik, "Performance Estimation of a Plasma PIC Code on the HTMT Architecture," internal report, 1999.

PetaFLOPS Workshop Series 1994-1999. *Workshop on Enabling Technologies for PetaFLOPS Computing Systems*, Pasedena, CA, February 22-24, 1994; *Preparation for PetaFLOPS Summer Workshop,* Bodega Bay, CA, July 22, 1995; *PetaFLOPS Workshop,* Bodega Bay, CA, August 14-23, 1995; *PetaFLOPS Architecture Workshop* (PAWS 96), Oxnard, CA, April 22, 1996; *PetaFLOPS Meeting*, Washington, DC, August 5, 1996; *PetaFLOPS Briefing*, Arglington, VA, August 28, 1996; *PetaFLOPS Workshop*, La Jolla, CA, January 28-29, 1997; *PETAFLOPS II Conference*, Santa Barbara, CA, February 19, 1999.

Siegel, Herb, and Loring Craymer, "The HTMT SAR Benchmark Results," internal report, 1999.

Sterling, Thomas, and Larry Bergman, "A Design Analysis of a Hybrid Technology Multithreaded Architecture for Petaflops Scale Computation," technical report, HTMT Tech Note #39, 1998.

Sterling, T., P. Messina, and P. H. Smith, *Enabling Technologies for Petaflops Computing*. MIT Press, Cambridge, MA, 1995.

Stevens, R., and V. E. Taylor, "Strategic Applications for Peta(FL)OPS Computational Systems," in *PetaFLOPS Workshop at the Frontiers Conference,* 1995.

Valerie Taylor, Xingfu Wu, Jonathan Geisler, Xin Li, Zhiling Lan, Rick Stevens, Mark Hereld, and Ivan R. Judson, "Prophesy: An Infrastructure for Analyzing and Modeling the Performance of Parallel and Distributed Applications," in *Proc. HPDC 2000*.

Valerie Taylor, Xingfu Wu, Jonathan Geisler, and Rick Stevens, "Using Kernel Couplings to Predict Parallel Application Performance," in *Proc. 11th IEEE International Symposium on High-Performance Distributed Computing (HPDC 2002*), Edinburgh, Scotland, July 24-26, 2002.

Theobald, Kevin B., Jose Nelson Amaral, Gerd Herber, Oliver Maquelin, Xinan Tang, and Guang R. Gao, "Overview of the Threaded-C Language," CAPSL Technical Memo 19, March 16, 1998.

Tremblay, Guy, Kevin B.Theobald, Christopher J.Morrone, Mark D.Butala, Jose Nelson Amaral, and Guang R. Gao, "Threaded-C Language Reference Manual (Release 2.0)," CAPSL Technical Memo 39, September 2000.

Wittie, L., D. Zinoviev, G. Sazaklis, and K. Likharev, "CNET: Design of an RSFQ Switching Network for Petaflops-Scale Computing," IEEE Trans. on Appl. Supercond., June 1999, vol. 9(2), pp. 4034-4039.