Coping With Very High Latencies in Petaflop Computer Systems

Sean Ryan, José N. Amaral, Guang Gao, Zachary Ruiz, Andres Marquez, Kevin Theobald*

Computer Architecture and Parallel Systems Laboratory, University of Delaware, Newark, DE, USA. http://*www.capsl.udel.edu*

The very long and highly variable latencies in the deep memory hierarchy of a petaflop-scale architecture design, such as the Hybrid Technology Multi-Threaded Architecture (HTMT) [13], present a new challenge to its programming and execution model. A solution to coping with such high and variable latencies is to directly and explicitly expose the different memory regions of the machine to the program execution model, allowing better management of communication. In this paper we describe the novel *percolation model* that lies at the heart of the HTMT program execution model [13]. The *Percolation Model* combines multithreading with dynamic prefetching of coarse-grain contexts. In the past, prefetching techniques have concentrated on moving blocks of *data* within the memory hierarchy. Instead of only moving contiguous blocks of data, the thread percolation approach manages contexts that include data, program instructions, and control states.

The main contributions of this paper include the specification of the HTMT runtime execution model based on the concept of percolation, and a discussion of the role of the compiler in a machine that exposes the memory hierarchy to the programming model.

1 Introduction

The Hybrid Technology Multi-Threaded (HTMT) Architecture project [15] has the goal of designing a petaflop scale computer by the year 2007. Such a machine will use a number of unconventional technologies such as: processors and interconnection networks built from super-conducting processing elements (called *SPELLs* [32]), networks based on RSFQ (Rapid Single Flux Quantum) logic devices [11], "Processor In Memory" (PIM) technology [20], high-performance optical packet switched network technology [7], optical holographic storage technology [26], and fine grain multi-threaded computing technology [16].

In this paper we introduce a new program execution model developed for the future HTMT machine. An important characteristic of the HTMT machine is the availability of a large number of very high performance super-conductor processing elements (SPELLs) with a modest amount of single-flux-quantum cryo-memory (CRAM) that can be accessed with a relatively low latency [32]. The latency for the next levels in the memory hierarchy (e.g. SRAM and DRAM) will be several orders of magnitude higher than a CPU cycle time in the SPELLs.

Our analysis shows that hiding the latencies of the deep memory hierarchy in the HTMT architecture is a great challenge; existing multi-threaded execution/programming models may not be able to cope with such latencies, where

* emails: {*ryan, amaral, ggao, ruiz, marquez, theobald*}*@capsl.udel.edu*

even a small percentage of cache misses can have disastrous effects upon performance. As a result, we introduce a new program execution and programming model, the *percolation model*, in order to meet this challenge.

Percolation can be considered to be a combination of multi-threading with dynamic prefetching and reorganization of the data as well as the threads which use the data into coarse-grain contexts. Prefetching in the past has concentrated on moving blocks of *data* within the memory hierarchy. Instead, functions, data, and control states are combined and sent in a single *parcel.* A parcel cannot be percolated to CRAM until all functions and data associated with it are available for transport. The latencies incurred in gathering data for the parcel and its component threads will then be made to overlap. Also, under this model, any data destined to be reused by the same thread is guaranteed to be already stored locally.

The programming model for percolation makes use of program directives to specify what pieces of data will be needed by a portion of the code and how the data should be organized into parcels *before* any code is actually sent to the high speed processors. Processors-in-memory (PIMs) provide the necessary capability to perform such data transformations and to prepare parcels. These parcels, once completed, are percolated to the fast processing units. Any results produced by the computation are percolated back to the memory after the computation in the processing units is complete. These results might then undergo (reverse) data transformations by the PIMs.

The unique memory model of the HTMT and its ramifications are discussed in section 2. Next, in section 3 we introduce, at a conceptual level, the percolation model of program execution as a means of coping with the architectural constraints discussed above. We then divide the conceptual diagram into phases of execution and introduce the runtime system (RTS) that implements the percolation model (section 4). This leads into a discussion of the role of a compiler for a petaflops machine (section 5) and of the next steps in evaluating the percolation model for the HTMT (section 7).

2 Memory Model

Conventional architectures present to the programmer the appearance of a uniform address space below the registers, with caching and paging hiding the real details of the hierarchy from the programmer. This luxury is not available in the HTMT model. Instead, each level of memory is considered to be a "buffer" of the next level, e.g., super-conducting memory is a buffer of the SRAM, which itself is a buffer of DRAM, and so on. Unlike in traditional cache organization, these buffers are directly addressable. Memory allocation and data movement at each level may be explicitly controlled if necessary [13]. [1]

We assume that the entire HTMT memory address space is explicitly divided into regions: a CRAM region, an SRAM region, a DRAM region, and an Optical

[1] Remote transactions such as data movement are handled asynchronously. Sync slots can be used to signal their completion.

memory region. The actual size of each region is initialized at system configuration time. A memory location can then be addressed by giving the region name and the corresponding offset. In this article we concentrate on the relation between the SRAM region and DRAM region.

Each memory location within the HTMT machine has a unique global address. However, data and instructions used by a processor for computation should be local before a processor can use them for computation. Currently, consistency of an object among regions of memory is the responsibility of the programmer. [2]

If a program moves a memory object from region X to region Y explicitly, we expect that the rest of the program, and any other programs that interact with it, is prepared for this movement and correct code is in place to access the moved memory object in region Y. Our percolation model, to be illustrated in the rest of this paper, assumes explicit control of such data movements between different memory regions.

3 The Percolation Model

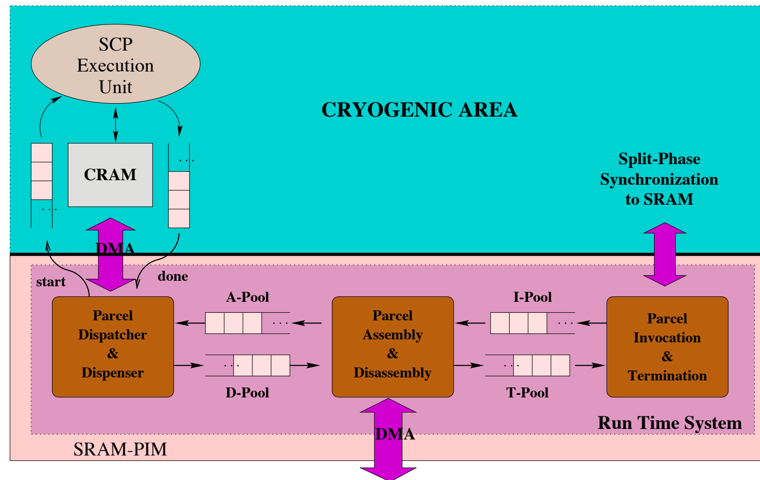

Fig. 1. HTMT Runtime Percolation Model.

This section describes the HTMT percolation model implemented by the HTMT runtime system running in SRAM-PIM. The runtime system consists of three major components: the Parcel Invocation and Termination Module (PIT), the Parcel Assembly and Disassembly Module (PAD), and the Parcel Dispatcher and

[2] In a future implementation of the HTMT system, the compiler and runtime system might, perhaps in conjunction with some "hints" provided by the user for efficiency, provide memory consistency management.

Dispenser Module (PDD). Various concurrent data structures are used to connect these major components; I-pool, T-pool, A-pool, and D-pool. The relation among these modules and data structures are shown in Figure 1.

The underlying purpose of the modules described below is to provide a parallel mechanism for preparing and retiring parcels of code and data, allowing the high speed processors to remain usefully busy at all times.

The main role of the PIT invocation manager is to detect if a parcel function (a function to be sent in a parcel to a SPELL) has become enabled, i.e. whether all its dependencies have been satisfied. The invocation manager will place enabled parcel functions in the I-pool. From here on we will refer to parcel threaded functions simply as parcels.

The PAD module will take enabled parcels from the I-pool and prepare them for further percolation. The role of the PAD assembly manager is to move the required code and data into local SRAM locations. At the same time, the data should be organized into its desired structure (via gather/scatter, permutation, pointer swizzling, etc.). This usually involves data movements from DRAM into SRAM through special PIM operations via the DMA (Direct Memory Access) channel between DRAM and SRAM as shown in Figure 1. Once the assembly process is completed, a parcel is passed on to the next stage by entering it into the A-pool.

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

After a parcel finishes its execution in the cryostatic region, it needs to be retired from it. This is begun by the PDD dispenser manager. A completed parcel has its return data (if any) structured and sent to space allocated in SRAM for post-processing. Aside from this processing (if any), the dispenser deallocates the CRAM resources reserved by the parcel. It then enters the parcel into the D-pool.

The PAD disassembly manager processes the parcels from the D-pool, disassembling and distributing output data into its proper places. It may then release the SRAM space used for assembly and disassembly, unless another thread will use the same space. When the disassembly process is finished, the parcel is entered into the T- pool for the final termination service.

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

3.1 Extensions to the Base Model

There is a short-cut path between the D-pool and the A-pool within the PAD module, provided so that parcels may be immediately re-enabled with minimum overhead.

Another extension is the connection between the cryostatic area and the PIM-resident PIT module. This will allow the SPELLs to initiate in an SRAM PIM an action or synchronization without retiring an entire parcel. Such actions are non-blocking and may be performed in parallel with computation.

3.2 Managing Concurrent Data Structures

Note that the runtime system, as described, is highly concurrent in itself. All the three modules can process different parcels in a pipelined fashion. We anticipate that a family of scheduling policies may be provided to manage concurrent data structures in a way to maximize code and data reuse.

4 An Overview of HTMT-Threaded-C

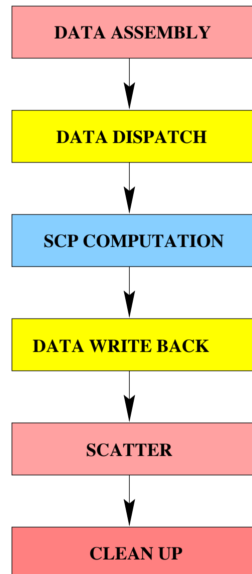

Fig. 2. Phases of the Percolation for HTMT-Threaded-C.

The first version of HTMT-Threaded-C considers the percolation process in the HTMT runtime system as described by a set of phases. These phases can overlap to implement computation in a pipelined fashion with data sharing (as among

iterations of an algorithm). In this section we outline the basic functionality added to the original Threaded-C language to define HTMT-Threaded-C. Primitive functionality in each phase is as follows:

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

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

PARCEL DISPATCH Assembles a parcel with the code of the parcel threaded function(s) to be executed in the SPELL. Percolates the parcel to the designated SPELL.

SPELL COMPUTATION Performs the specified computation in the cryogenic processor. The data used during this phase is referenced by its address in the SRAM region. In HTMT-Threaded-C this computation is specified by a PARCEL THREADED function.

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

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

CLEAN UP Release memory that has been reserved in the CRAM region to allow the start of the next percolation. An equivalent process occurs in other memory regions.

The phases of computation are presented in Figure 2. Although this figure indicates a sequential execution of the phases, the actual program execution model allows for the overlapping of the percolation phases. For instance, after the data for one iteration of the computation has been assembled and dispatched, the assembly of the data needed in the next iteration can start while the SPELL computation of the first iteration is performed. To allow the overlapping of computation phases, the runtime system implements a synchronization mechanism based on synchronization slots that allows for the specification of multi-threaded programs in the SRAM-PIM level. Such a synchronization mechanism was previously implemented in the EARTH system [31].

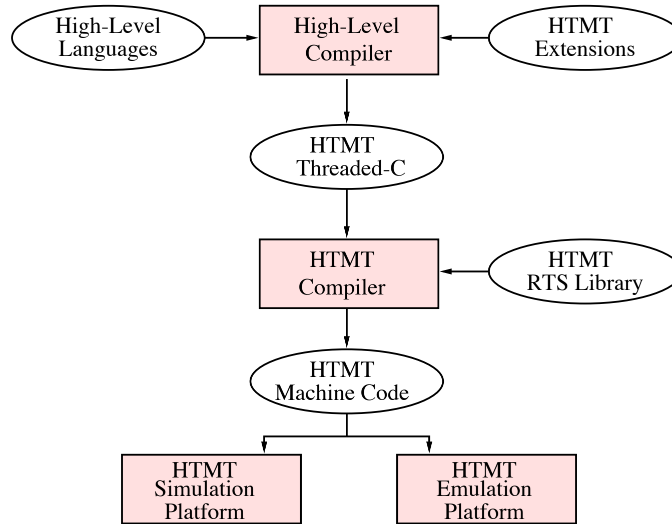

Fig. 3. Relation between a Future HTMT high-level compiler and the HTMT-C language.

5 The Role of the Compiler and of the Runtime System

Providing support for multi-threading at the architecture level is not enough, we must also be able to effectively program such architectures.

HTMT-C is the language introduced to allow the implementation of the program execution model described in Section 3. HTMT-C is currently implemented as a library of functions that extends Threaded-C, a thread-aware multi-threading language originally proposed at McGill University in Canada and that is undergoing incremental improvement by the CAPSL group at the University of Delaware.

Figure 3 illustrates the relationship between an HTMT high-level compiler and the HTMT-C Language. It is expected that in the future a compiler for HTMT will support high-level parallel languages with only a few HTMT extensions. Such a language would include a library of functions to perform data transformations and means to automatically generate threads and allow for run-time partitioning of threads into strands.

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

The HTMT-C language is explicitly multi-threaded and requires a set of primitives to implement the percolation model. The functionality described by the semantics of these primitives is implemented by the HTMT Runtime System (RTS). For example, a request to dispatch a parcel to CRAM, in conjunction with

the appropriate synchronization, must be handled by the RTS. [3]. In general, the RTS is required to:

- Implement the thread synchronization mechanism through synchronization slots;
- Implement the percolation model providing primitives for the percolation of data and code;
- Provide the automatic translation of addresses from the SRAM region to the CRAM region.

In this document, we describe a number of actions as the responsibility of the "programmer". In the future, many responsibilities inherent to the percolation model, including data assembly, synchronization primitives, etc., may be undertaken by library functions or assumed entirely by the compiler.

6 Perfomance Evaluation

Although HTMT-C is currently implemented at University of Delaware, profiling-based performance analysis is not available at the time of publication. Emulation is not the only means of determining performance, however. Some analytical studies of the HTMT architecture's performance, using the percolation model, have been performed [6]. [4] Working HTMT-C code examples can be made available upon request.

7 Future Work

During the current phase of the HTMT project, that will end in July 1999, the Delaware team will deliver the program execution model emulator for the machine. This emulator will enable application scientists from some national laboratories to develop irregular applications in HTMT-Threaded-C and to obtain measurements for runtime parameters in these applications. In collaboration with these scientists we will develop an analytical performance model that takes as input the runtime parameters measured by the emulator and estimations for architectural parameters such as processing speed, communication latency, communication bandwidth and storage capacities in the different levels of the machine.

If financing is in place for the next phase of the project, the HTMT-Threaded-C language will be revised and a compiler framework will be developed for the construction of a compiler that enables the development of applications for HTMT in a higher level language.

---

[3] It should be noted thata fully functional RTS is required for any program in HTMT-C to be exectured

[4] The algorithm analyzed in this report should be implemented in HTMT-C by the time this article is published.

Although the percolation model of execution was originally proposed for the HTMT project, it is suitable to other architectures that also have to cope with high latencies but that do not necessarily have as many levels of processing. We are currently working in a split phase percolation model that might extend the model described in this paper.

8 Related Work

The percolation model presented in this paper is an extension of EARTH, a fine grain multi-threaded model developed by Prof. Gao and many of his students and research associates [16]. Many other architectures have been proposed to address the problem of tolerating inherent communication and synchronization latencies by switching to a new ready thread of control whenever a long-latency operation is encountered [4, 5, 8, 10, 12, 18, 19, 23–25, 29].

Central ideas in the program execution model proposed in this document originate from the extensive experience that the Delaware team has acquired with the multi-threading program execution model and the multi-threaded language developed for EARTH [16, 31]. The initial design of the HTMT-Threaded-C language presented here is a simple extension of the Threaded-C language. Thus the HTMT project will be able to benefit from the joint effort and investment of the McGill/Delaware group in the development of the EARTH architecture and the Threaded-C language. Another benefit of choosing an extension of Threaded-C as a first emulator for the HTMT project is the fact that Threaded-C is currently operational on a number of important parallel platforms.

This document builds on a number of documents, discussions, and research efforts in Delaware and elsewhere. In our July 1997 Technical Memo 09 we presented the concept of a percolation model [13]. In related studies the Delaware group has explored ways to achieve high levels of parallelism at the instruction level without incurring a great penalty in the real estate required for control flow and synchronization mechanisms in the hardware implementation of the machine [21, 22]. The Super-strand Architecture introduces the notion of a *strand* as a block of instructions grouped together to become a scheduling quantum of execution. The first experiments with this architecture indicate that programs can be efficiently partitioned into strands to be executed under a super-strand execution model.

Compiling a program that is not thread aware into a multi-threaded program is a difficult task that includes the need to partition the code into threads. In the area of functional language, research into this problem is more abundant [9, 17, 28, 27]. Hendren *et al.* and Tang et al. have introduced heuristic-based thread partitioning algorithms for imperative languages [14, 30].

Acknowledgments

The model presented in this paper is the result of fruitful discussions with a number of important researchers and scientists in the HTMT community, including

Thomas Sterling, Burton Smith, Peter Kogge, Vince Freeh, Nikos Chrisochoide, Larry Bergman, Rick Stevens, Loring Cramer, John Salmon and Herb Spiegel. Both Threaded-C and HTMT-Threaded-C are continually been improved thanks to feedback from John Salmon, Phil Merkey, Charles Norton, John Lou, Mark Hereld, Ivan R. Judson, Xiaohui Shen, Tom Cwik, and others. Many current and former members of Prof. Gao's research group made Threaded-C possible, we highlight the works of Herbert H. J. Hum, Olivier Maquelin, Haiying Cai, Prasad Kakulavarapu, Cheng Li, Shashansk Nemawarkar, Xinan Tang, Gerd Heber, Ruppa Thulasiram, Thomas Geiger, and Parimala Thulasiraman.

The model described in this paper was developed on a research performed for the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the National Security Agency (NSA) through an agreement with the National Aeronautics and Space Administration.

References

1. *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.
2. ACM SIGARCH and IEEE Computer Society. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, California, May 17–19, 1993. *Computer Architecture News,* 21(2), May 1993.
3. ACM SIGARCH and IEEE Computer Society. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 22–24, 1995. *Computer Architecture News,* 23(2), May 1995.
4. Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* [3], pages 2–13. *Computer Architecture News,* 23(2), May 1995.
5. Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. Presented at the Workshop on Multithreaded Computers, held at *Supercomputing '91*, Albuquerque, New Mexico, November 1991.
6. Jose Nelson Amaral, 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. Technical report, University of Delaware, 1999.
7. Karen Bergman and Coke Reed. Hybrid technology multithreaded architecture program design and development of the data vortex network. Technical report, Princeton University, 1998. Technical Note 2.0.
8. Derek Chiou, Boon S. Ang, Robert Greiner, Arvind, James C. Hoe, Michael J. Beckerle, James E. Hicks, and Andy Boughton. StarT-NG: Delivering seamless parallel computing. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *Proceedings of the First International EURO-PAR Conference*, number 966 in Lecture Notes in Computer Science, pages 101–116, Stockholm, Sweden, August 29–31, 1995. Springer-Verlag.

9. David E. Culler, Seth C. Goldstein, Klaus E. Schauser, and Thorsten von Eicken. TAM – a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.
10. David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society. *Computer Architecture News,* 19(2), April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices,* 26(4), April 1991.
11. Mikhail Dorojevets, Paul Bunyk, Dmitri Zinoviev, and Konstantin Likharev. Petaflops rsfq system design. In *Applied Superconductivity Conference*, Sept 1998.
12. Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine multicomputer. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 146–156, Ann Arbor, Michigan, November 29–December1, 1995. IEEE-CS TC-MICRO and ACM SIGMICRO.
13. Guang R. Gao, Kevin B. Theobald, Andrés Márquez, and Thomas Sterling. The HTMT program execution model. CAPSL Technical Memo 09, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, July 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.
14. Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)* [1], pages 12–23.
15. HTMT. Hybid technology multi-threaded architectures. http://htmt.caltech.edu, 1998.
16. Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.
17. Robert A. Iannucci. A dataflow/von Neumann hybrid architecture. Technical Report MIT/LCS/TR-418, MIT Laboratory for Computer Science, Cambridge, Massachusetts, July 1988. PhD thesis, May 1988.
18. Robert A. Iannucci, Guang R. Gao, Robert H. Halstead, Jr., and Burton Smith, editors. *Multithreaded Computer Architecture: A Summary of the State of the Art.* Kluwer Academic Publishers, Norwell, Massachusetts, 1994. Book contains papers presented at the Workshop on Multithreaded Computers, held in conjunction with Supercomputing '91 in Albuquerque, New Mexico, November 1991.
19. Yuetsu Kodama, Hirohumi Sakane, Mitsuhisa Sato, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi. The EM-X parallel computer: Architecture and basic performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* [3], pages 14–23. *Computer Architecture News,* 23(2), May 1995.
20. Peter M. Kogge, Jay B. Brockman, Thomas Sterling, and Guang Gao. Processing-in-memory: Chips to petaflops. Technical report, International Symposium on Computer Architecture, Denver, Co., June 1997.
21. Andrés Márquez, Kevin B. Theobald, Xinan Tang, and Guang R. Gao. A superstrand architecture. CAPSL Technical Memo 14, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, December 1997. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

22. Andrés Márquez, Kevin B. Theobald, Xinan Tang, Thomas Sterling, and Guang R. Gao. A superstrand architecture and its compilation. CAPSL Technical Memo 18, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 1998.
23. R. S. Nikhil and Arvind. Id: a language with implicit parallelism. In J. Feo, editor, *A Comparative Study of Parallel Programming Languages: The Salishan Problems*. Elsevier Science Publishers, February 1990.
24. Michael D. Noakes, Deborah A. Wallah, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* [2], pages 224–235. *Computer Architecture News,* 21(2), May 1993.
25. Kazuaki Okamoto, Shuichi Sakai, Hiroshi Matsuoka, Takashi Yokota, and Hideo Hirono. Multithread execution mechanisms on RICA-1 for massively parallel computation. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)* [1], pages 116–121.
26. Demetri Psaltis and Geoffrey W. Burr. Holographic data storage. *Computer*, 31(2):52–60, Febuary 1998.
27. Klaus E. Schauser, David E. Culler, and Seth C. Goldstein. Separation constraint partitioning — A new algorithm for partitioning non-strict programs into sequential threads. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 259–271, San Francisco, California, January 22–25, 1995.
28. Klaus Eric Schauser, David E. Culler, and Thorsten von Eiken. Compiler-controlled multithreading for lenient parallel languages. Report No. UCB/CSD 91/640, Computer Science Division, University of California at Berkeley, 1991.
29. Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally. Evaluation of mechanisms for fine-grained parallel programs in the J-Machine and the CM-5. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* [2], pages 302–313. *Computer Architecture News,* 21(2), May 1993.
30. Xinan Tang, Jian Wang, Kevin B. Theobald, and Guang R. Gao. Thread partitioning and scheduling based on cost model. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–281, Newport, Rhode Island, June 22–25, 1997. SIGACT/SIGARCH and EATCS.
31. Kevin B. Theobald, José Nelson Amaral, Gerd Heber, Olivier Maquelin, Xinan Tang, and Guang R. Gao. Overview of the Threaded-C language. CAPSL Technical Memo 19, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 1998. In ftp://ftp.capsl.udel.edu/pub/doc/memos.
32. L. Wittie, 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. In press.