[119] Kevin B. Theobald, "Panel sessions of the 1991 workshop on multithreaded computers," *Computer Architecture News*, 22(1):2–33, March 1994.

[120] Kevin B. Theobald, Guang R. Gao, and Laurie J. Hendren, "On the limits of program parallelism and its smoothability," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, Portland, Oregon, pp. 10–19, December 1992.

[121] R. H. Thomas, R. Gurwitz, J. Goodhue, and D. Allen, "Butterfly parallel processor overview," technical report, Bolt Beranek and Newman, Cambridge, Massachusetts, December 1985.

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

[123] Kenneth R. Traub, "Compilation as partitioning: A new approach to compiling non-strict functional languages," CSG Memo 291, Computation Structures Group, MIT Laboratory for Computer Science, 1988.

[124] Kenneth R. Traub, *Sequential Implementation of Lenient Programming Languages.* PhD thesis, Massachusetts Institute of Technology, September 1988. Also published as Technical Report MIT/LCS/TR-417, MIT Laboratory for Computer Science.

[125] D. A. Turner, "The semantic elegance of applicative languages," in *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, New Hampshire, pp. 85–97, October 1981.

[126] David W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, pp. 176–188, April 1991.

[127] Ian Watson and John Gurd, "A practical data flow computer," *Computer*, 15(2):51–57, February 1982.

[128] Wolf-Dietrich Weber and Anoop Gupta, "Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, pp. 273–280, May–June 1989.

[129] X3J3, The FORTRAN Technical Committee of ANSI, "FORTRAN 90, draft of the international standard," June 1990.

[130] Toshitsugu Yuba, Toshio Shimada, Kei Hiraki, and Hiroshi Kashiwagi, "SIGMA-1: a dataflow computer for scientific computations," *Computer Physics Communications*, 37:141–148, July 1985.

[131] Hans P. Zima, Heinz-J. Bast, and Michael Gerndt, "SUPERB: a tool for semi-automatic MIMD/SIMD parallelization," *Parallel Computing*, 6:1–18, January 1988.

[105] A. R. Pleszkun, J. R. Goodman, W.-C. Hsu, R. T. Joersz, G. Bier, P. Woest, and P. Schechter, "WISQ: A restartable architecture using queues," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, June 1987.

[106] Steven A. Przybylski, *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. Morgan Kaufmann, 1990.

[107] George Radin, "The 801 minicomputer," in *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, pp. 39–47, March 1982.

[108] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towle, "The Cydra 5 departmental supercomputer," *Computer*, 22(1):12–35, January 1989.

[109] Gary Sabot, "A compiler for a massively parallel distributed memory MIMD computer," in *Proceedings of Frontiers '92: The Fourth Symposium on the Frontiers of Massively Parallel Computation*, McLean, Virginia, pp. 12–20, October 1992.

[110] Shuichi Sakai, Yoshinori Yamaguchi, Kei Hiraki, Yuetsu Kodama, and Toshitsugu Yuba, "An architecture of a dataflow single chip processor," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, pp. 46–53, May–June 1989.

[111] Charles L. Seitz, "The Cosmic Cube," *Communications of the ACM*, 28(1):22–33, January 1985.

[112] Jong-Jiann Shieh and Christos A. Papachristou, "On reordering instruction streams for pipelined computers," in *Proceedings of the 22th Annual International Workshop on Microprogramming and Microarchitecture*, Dublin, Ireland, pp. 199–206, August 1989. *SIGMICRO Newsletter,* 20(3), September 1989.

[113] Alan Jay Smith, "Cache memories," *Computing Surveys*, 14(3):473–530, September 1982.

[114] Burton Smith, "The architecture of HEP," in *Parallel MIMD Computation: The HEP Supercomputer and its Applications* (Janusz S. Kowalik, ed.), pp. 41–55, MIT Press, 1985.

[115] Burton J. Smith, "Architecture and applications of the HEP multiprocessor computer system," in *Proceedings of SPIE – Real-Time Signal Processing IV*, vol. 298, San Diego, California, pp. 241–248, August 25–28, 1981.

[116] Harold S. Stone and John Cocke, "Computer architecture in the 1990s," *Computer*, 24(9):30–38, September 1991.

[117] C. K Tang, "Cache system design in the tightly coupled multiprocessor system," in *AFIPS Conference Proceedings, 1976 National Computer Conference*, New York City, New York, pp. 749–753, June 7–10, 1976.

[118] Charles P. Thacker and Lawrence C. Stewart, "Firefly: a multiprocessor workstation," in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, pp. 164–172, October 1987.

[91] Kai Li and Paul Hudak, "Memory coherence in shared virtual memory systems," in *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, Calgary, Alberta, pp. 229–239, August 11–13, 1986.

[92] J. R. McGraw *et al.*, "SISAL: Streams and iteration in a single assignment language—language reference manual version 1.2," Technical Report M-146, Lawrence Livermore National Laboratory, 1985.

[93] Charles Melear, "The design of the 88000 RISC family," *IEEE Micro*, 13(2):26–38, April 1989.

[94] Edward F. Miller, Jr., "A multiple-stream registerless shared-resource processor," *IEEE Transactions on Computers*, 23(3):277–285, March 1974.

[95] Mamata Misra, ed., *IBM RISC System/6000 Technology*. International Business Machines Corporation, 1990. Order No. SA23-2619.

[96] Rishiyur S. Nikhil, "The parallel programming language Id and its compilation for parallel machines," CSG Memo 313, Computation Structures Group, MIT Laboratory for Computer Science, July 1990.

[97] Rishiyur S. Nikhil and Arvind, "Can dataflow subsume von Neumann computing?" in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, pp. 262–272, May–June 1989.

[98] Rishiyur S. Nikhil and Arvind, "Id: a language with implicit parallelism," CSG Memo 305, Computation Structures Group, MIT Laboratory for Computer Science, 1990.

[99] 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*, San Diego, California, pp. 224–235, May 1993.

[100] Amos R. Omondi, "Design of a high performance instruction pipeline," *Computer Systems Science and Engineering*, 6(1):13–29, January 1991.

[101] Gregory M. Papadopoulos, *Implementation of a General Purpose Dataflow Multiprocessor*. MIT Press, 1991. Revised version of the author's Ph.D. dissertation, MIT, 1988.

[102] Gregory M. Papadopoulos and David E. Culler, "Monsoon: an explicit token-store architecture," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, Washington, pp. 82–91, May 1990.

[103] Gregory M. Papadopoulos and Kenneth R. Traub, "Multithreading: A revisionist view of dataflow architectures," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, Toronto, Ontario, pp. 342–351, May 1991.

[104] David A. Patterson and Carlo H. Sequin, "RISC I: A reduced instruction set VLSI computer," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, Minneapolis, Minnesota, pp. 443–450, May 1981.

[77] John B. Johnston, "The contour model of block structured processes," in *Proceedings of a Symposium on Data Structures in Programming Languages*, pp. 55–82, ACM, 1971. *SIGPLAN Notices,* 6(2), February 1971.

[78] Norman P. Jouppi and David W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, pp. 272–282, April 1989.

[79] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G Sheldon, "Implementing a cache consistency protocol," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, Boston, Massachusetts, pp. 276–283, June 1985.

[80] Robert M. Keller, "Look-ahead processors," *Computing Surveys*, 7(4):177–195, December 1975.

[81] Parviz Kermani and Leonard Kleinrock, "Virtual cut-through: A new computer communication switching technique," *Computer Networks*, 3(4):267–286, September 1979.

[82] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr., "Data optimization: Allocation of arrays to reduce communication on SIMD machines," *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[83] Yuetsu Kodama, Yasuhito Koumura, Mitsuhisa Sato, Hirohumi Sakane, Shuichi Sakai, and Yoshinori Yamaguchi, "EMC-Y: Parallel processing element optimizing communication and computation," in *Conference Proceedings, 1993 International Conference on Supercomputing*, Tokyo, Japan, pp. 167–174, July 1993.

[84] Yuetsu Kodama, Shuichi Sakai, and Yoshinori Yamaguchi, "A prototype of a highly parallel dataflow machine EM-4 and its preliminary evaluation," in *Proceedings of InfoJapan 90*, pp. 291–298, October 1990.

[85] Charles Koelbel, "Compile-time generation of communications for scientific programs," Technical Report CRPC-TR91089, Center for Resarch on Parallel Computation, Rice University, January 1991.

[86] Peter M. Kogge, *The Architecture of Pipelined Computers.* New York: McGraw-Hill Book Company, 1981.

[87] Les Kohn and Neal Margulis, "Introducing the Intel i860 64-bit microprocessor," *IEEE Micro*, 13(4):15–30, August 1989.

[88] Janusz S. Kowalik, ed., *Parallel MIMD Computation: The HEP Supercomputer and its Applications.* MIT Press, 1985.

[89] James T. Kuehn and Burton J. Smith, "The Horizon supercomputing system: Architecture and software," in *Proceedings of Supercomputing '88*, Orlando, Florida, pp. 28–34, November 1988.

[90] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy, "The DASH prototype: Implementation and performance," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, pp. 92–103, May 1992.

[63] Robert H. Halstead, Jr., "Multilisp: A language for concurrent symbolic computation," *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[64] Robert H. Halstead, Jr. and Tetsuya Fujita, "MASA: A multithreaded processor architecture for parallel symbolic computing," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, pp. 443–451, May–June 1988.

[65] John L. Hennessy and Norman P. Jouppi, "Computer technology and architecture: An evolving interaction," *Computer*, 24(9):18–29, September 1991.

[66] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[67] James Hicks, Derek Chiou, Boon Seong Ang, and Arvind, "Performance studies of Id on the Monsoon dataflow system," *Journal of Parallel and Distributed Computing*, 18(3):273–300, July 1993.

[68] High Performance Fortran Forum, "High-performance fortran language specification," technical report, Rice University, May 1993.

[69] Mark D. Hill, "A case for direct-mapped caches," *Computer*, 21(12):25–40, December 1988.

[70] Kei Hiraki, Satoshi Sekiguchi, and Toshio Shimada, "Status report of SIGMA-1: A dataflow supercomputer," in *Advanced Topics in Data-Flow Computing* (Jean-Luc Gaudiot and Lubomir Bic, eds.), ch. 7, Prentice-Hall, 1991. Book contains papers presented at the First Workshop on Data-Flow Computing, held in conjunction with the 16th Annual International Symposium on Computer Architecture in Eilat, Israel, May 1989.

[71] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng, "Compiler optimizations for Fortran D on MIMD distributed-memory machines," in *Proceedings of Supercomputing '91*, Albuquerque, New Mexico, pp. 86–100, November 1991.

[72] Herbert H. J. Hum and Guang R. Gao, "Efficient support of concurrent threads in a hybrid dataflow/von Neumann architecture," in *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, pp. 190–193, ACM SIGARCH and IEEE Computer Society, December 2–5, 1991.

[73] Herbert H. J. Hum and Guang R. Gao, "A novel high-speed memory organization for fine-grain multi-thread computing," in *Proceedings of PARLE '91 – Parallel Architectures and Languages Europe*, vol. I, Eindhoven, The Netherlands, pp. 34–51, June 1991.

[74] Robert A. Iannucci, *A Dataflow/von Neumann Hybrid Architecture*. PhD thesis, Massachusetts Institute of Technology, July 1988. Also published as Technical Report MIT/LCS/TR-418, MIT Laboratory for Computer Science.

[75] Robert A. Iannucci, "Toward a dataflow/von Neumann hybrid architecture," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, pp. 131–140, May–June 1988.

[76] Mike Johnson, *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[50] Digital Equipment Corporation, *Alpha Architecture Manual*. Burlington, Vermont, 1992.

[51] E. W. Dijkstra, "Co-operating sequential processes," in *Programming Languages* (F. Genuys, ed.), pp. 43–112, New York: Academic Press, 1968.

[52] John R. Ellis, *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, Yale University, May 1984. ACM Doctoral Dissertation Award, 1985; published in 1986.

[53] Joseph A. Fisher, "Very long instruction word architectures and the ELI-512," in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, Stockholm, Sweden, pp. 140–150, June 1983.

[54] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau, "Parallel processing: A smart compiler and a dumb machine," in *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montréal, Québec, pp. 37–47, June 1984.

[55] G. Gao and Q. Ning, "Loop storage optimization for dataflow machines," in *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, no. 589 in Lecture Notes in Computer Science, Santa Clara, California, pp. 359–373, Springer-Verlag, August 1991. Published in 1992.

[56] Guang R. Gao, Herbert H. J. Hum, and Yue-Bong Wong, "Limited balancing—an efficient method for dataflow software pipelining," in *Proceedings of the ISMM International Conference on Parallel and Distributed Computing and Systems*, New York, New York, International Society for Mini and Microcomputers, October 10–12, 1990.

[57] Guang R. Gao, Herbert H. J. Hum, and Yue-Bong Wong, "Parallel function invocation in a dynamic argument-fetching dataflow architecture," in *Proceedings of PARBASE '90–International Conference on Databases, Parallel Architectures, and Their Applications*, Miami Beach, Florida, pp. 112–116, IEEE Computer Society, March 7–9, 1990.

[58] Guang R. Gao, Yue-Bong Wong, and Qi Ning, "A timed Petri-Net model for fine-grain loop scheduling," in *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, pp. 204–218, June 1991.

[59] James R. Goodman, "Using cache memory to reduce processor-memory traffic," in *Proceedings of the 10th Annual International Symposium on Computer Architecture*, Stockholm, Sweden, pp. 124–131, June 1983.

[60] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes, "The εpsilon dataflow processor," in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, pp. 36–45, May–June 1989.

[61] V. G. Grafe and J. E. Hoch, "The Epsilon-2 hybrid dataflow architecture," in *Digest of Papers, 35th IEEE Computer Society International Conference, COMPCON Spring '90*, San Francisco, California, pp. 88–93, February–March 1990.

[62] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber, "Comparative evaluation of latency reducing and tolerating techniques," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, Toronto, Ontario, pp. 254–263, May 1991.

[35] William J. Dally, Linda Chao, Andrew Chien, Soha Hassoun, Waldemar Horwat, Jon Kaplan, Paul Song, Brian Totty, and Scott Wills, "Architecture of a message-driven processor," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, pp. 189–196, June 1987.

[36] William J. Dally and Charles L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Transactions on Computers*, 36(5):547–553, May 1987.

[37] William J. Dally and D. Scott Wills, "Universal mechanisms for concurrency," in *Proceedings of PARLE '89 – Parallel Architectures and Languages Europe*, vol. I, Veldhoven, The Netherlands, pp. 19–33, June 1989.

[38] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for EPEX/FORTRAN," *Parallel Computing*, 7:11–24, April 1988.

[39] Alan L. Davis and Robert M. Keller, "Data flow program graphs," *Computer*, 15(2):26–41, February 1982.

[40] Denelcor, Inc., "HEP principles of operation," 1979.

[41] Jack B. Dennis, "First version of a data-flow procedure language," in *Proceedings of the Colloque sur la Programmation*, no. 19 in Lecture Notes in Computer Science, Paris, France, pp. 362–376, Springer-Verlag, April 9–11, 1974.

[42] Jack B. Dennis, "Data flow supercomputers," *Computer*, 13(11):48–56, November 1980.

[43] Jack B. Dennis, "The Paradigm Compiler: Mapping a functional language for the Connection Machine," in *Scientific Applications of the Connection Machine* (Horst Simon, ed.), pp. 301–315, Singapore: World Scientific, 1989.

[44] Jack B. Dennis, "The evolution of "static" data-flow architecture," in *Advanced Topics in Data-Flow Computing* (Jean-Luc Gaudiot and Lubomir Bic, eds.), ch. 2, Prentice-Hall, 1991.

[45] Jack B. Dennis and Guang R. Gao, "An efficient pipelined dataflow processor architecture," in *Proceedings of Supercomputing '88*, Orlando, Florida, pp. 368–373, November 1988.

[46] Jack B. Dennis and Guang Rong Gao, "Maximum pipelining of array operations on static data flow machine," in *Proceedings of the 1983 International Conference on Parallel Processing*, Bellaire, Michigan, pp. 331–334, August 1983.

[47] Jack B. Dennis, Guang-Rong Gao, and Kenneth W. Todd, "Modeling the weather with a data flow supercomputer," *IEEE Transactions on Computers*, 33(7):592–603, July 1984.

[48] Jack B. Dennis and David P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proceedings of the 2nd Annual Symposium on Computer Architecture*, Houston, Texas, pp. 126–132, January 1975.

[49] Keith Diefendorff and Michael Allen, "Organization of the Motorola 88110 superscalar RISC microprocessor," *IEEE Micro*, 12(2):40–63, April 1992.

[22] Lubomir Bic, "A process-oriented model for efficient execution of dataflow programs," in *Proceedings of the 7th International Conference on Distributed Computing Systems*, Berlin, West Germany, pp. 332–336, IEEE Computer Society, September 1987.

[23] Philip Bitar, "The weakest memory-access order," *Journal of Parallel and Distributed Computing*, 15(4):305–331, August 1992.

[24] David L. Black, Anoop Gupta, and Wolf-Dietrich Weber, "Competitive management of distributed shared memory," in *Digest of Papers, 34th IEEE Computer Society International Conference, COMPCON Spring '89*, San Francisco, California, pp. 184–190, February–March 1989.

[25] D. Allan Bromley, ed., *Grand Challenges: High Performance Computing and Communications. The FY 1992 U.S. Research and Development Program.* Committee on Physical, Mathematical, and Engineering Sciences; Federal Coordinating Council for Science, Engineering, and Technology; Office of Science and Technology Policy, 1992.

[26] E. Brooks, "The attack of the killer micros." Presentation in the Teraflop Computing Panel Discussion at Supercomputing '89, Reno, Nevada, November 1989.

[27] Richard Buerher and Kattamuri Ekanadham, "Incorporating data flow ideas into von Neumann processors for parallel execution," *IEEE Transactions on Computers*, 36(12):1515–1522, December 1987.

[28] Lucien M. Censier and Paul Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Transactions on Computers*, 27(12):1112–1118, December 1978.

[29] David Chaiken, John Kubiatowicz, and Anant Agarwal, "LimitLESS directories: A scalable cache coherence scheme," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, pp. 224–234, April 1991.

[30] Merrill Cornish, "The TI dataflow architecture: The power of concurrency for avionics," in *Proceedings of the 3rd Digital Avionics Systems Conference*, pp. 19–25, IEEE and AIAA, November 6–8, 1979.

[31] David E. Culler, Anurag Sah, Klaus Eric 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*, Santa Clara, California, pp. 164–175, April 1991.

[32] Robert C. Daley and Jack B. Dennis, "Virtual memory, processes and sharing in MULTICS," *Communications of the ACM*, 11(5):306–312, May 1968.

[33] William J. Dally, "Wire efficient VLSI multiprocessor communication networks," in *Proceedings of the Stanford Conference on Advanced Research in VLSI*, pp. 391–415, March 1987.

[34] William J. Dally, "Fine-grain message-passing concurrent computers," in *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, vol. I, Pasadena, California, pp. 2–12, ACM, January 1988.

[9] Arvind, Kim P. Gostelow, and Wil Plouffe, "An asynchronous programming language and computing machine," Technical Report 114a, Department of Information and Computer Science, University of California at Irvine, December 1978.

[10] Arvind and Robert A. Iannucci, "Two fundamental issues in multiprocessing," in *Parallel Computing in Science and Engineering*, no. 295 in Lecture Notes in Computer Science, pp. 61–88, Springer-Verlag, 1987. Proceedings of the 4th International DFVLR Seminar on Foundations of Engineering Sciences, Bonn, West Germany, June 25–26, 1987.

[11] Arvind, Vinod Kathail, and Keshav Pingali, "A dataflow architecture with tagged tokens," Technical Memo MIT/LCS/TM-174, MIT Laboratory for Computer Science, September 1980.

[12] Arvind and Rishiyur S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[13] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali, "I-structures: Data structures for parallel computing," *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[14] Arvind and Robert E. Thomas, "I-structures: An efficient data type for functional languages," Technical Memo MIT/LCS/TM-178, MIT Laboratory for Computer Science, September 1980.

[15] H. B. Bakoglu and T. Whitside, "RISC System/6000 hardware overview," in *IBM RISC System/6000 Technology* (Mamata Misra, ed.), pp. 8–15, International Business Machines Corporation, 1990. Order No. SA23-2619.

[16] BBN Advanced Computers Inc., "The Butterfly GP1000 parallel processor," in *Butterfly Parallel Processing — General Information, Tests, and Modifications: Articles on the Butterfly Computer*, pp. 3–22, Cambridge, Massachusetts: Bolt Beranek and Newman, 1986.

[17] Micah Beck, Keshav K. Pingali, and Alex Nicolau, "Static scheduling for dynamic dataflow machines," Technical Report TR 90-1076, Department of Computer Science, Cornell University, Ithaca, New York, January 1990.

[18] Michael J. Beckerle, "Overview of the START(*T) multithreaded computer," in *Digest of Papers, 38th IEEE Computer Society International Conference, COMPCON Spring '93*, San Francisco, California, pp. 148–156, February 1993.

[19] David Bernstein, "PREFACE-2: Supporting nested parallelism in FORTRAN," Technical Report RC14160, IBM Research, 1988.

[20] David Bernstein and Izidor Gertner, "Scheduling expressions on a pipelined processor with a maximal delay of one cycle," *ACM Transactions on Programming Languages and Systems*, 11(1):57–66, January 1989.

[21] Daniel M. Berry, "Introduction to Oregano," in *Proceedings of a Symposium on Data Structures in Programming Languages*, pp. 171–190, ACM, 1971. *SIGPLAN Notices,* 6(2), February 1971.

checking the literature references. Vugranam C. Sreedhar, Shashank Nemawarkar and Chandrika Mukerji have read the draft of the paper and provided suggestions for its improvement.

We thank Burton Smith for giving us access to design documents of the HEP machine. Arvind, David Culler, Rishiyur Nikhil, Burton Smith, Robert Iannucci, Robert Halstead, Greg Papadopoulos, and Shuichi Sakai have visited McGill University, and this paper benefited from discussions about their architectures. Anoop Gupta has helped us in the understanding of the DASH architecture and its memory hierarchy.

Finally, the second author would like to thank the National Science and Engineering Research Council (NSERC) of Canada for their support.

References

[1] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin, "Sparcle: An evolutionary processor design for multiprocessors," *IEEE Micro*, 13(3):48–61, June 1993.

[2] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz, "APRIL: A processor architecture for multiprocessing," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, Washington, pp. 104–114, May 1990.

[3] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz, "An evaluation of directory schemes for cache coherence," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, pp. 280–289, May–June 1988.

[4] Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr., "Compiling Fortran 8x array features for the Connection Machine computer system," in *Proceedings of the ACM/SIGPLAN PPEALS 1988 — Parallel Programming: Experience with Applications, Languages and Systems*, New Haven, Connecticut, pp. 42–56, July 1988. *SIGPLAN Notices,* 23(9), September 1988.

[5] Stephen J. Allan and R. R. Oldehoeft, "HEP SISAL: Parallel functional programming," in *Parallel MIMD Computation: The HEP Supercomputer and its Applications* (Janusz S. Kowalik, ed.), pp. 123–150, MIT Press, 1985.

[6] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith, "The Tera computer system," in *Conference Proceedings, 1990 International Conference on Supercomputing*, Amsterdam, The Netherlands, pp. 1–6, June 1990.

[7] Marco Annaratone, Emmanuel Arnould, Thomas Gross, H. T. Kung, Monica Lam, Onat Menzilicioglu, and Jon A. Webb, "The Warp computer: Architecture, implementation and performance," *IEEE Transactions on Computers*, 36(12):1523–1538, December 1987.

[8] James Archibald and Jean-Loup Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

9 Conclusions

We have surveyed multithreaded computer architecture with two domains of application in mind: workstations and servers to support general purpose computation; and massively parallel computers for scientific computation. Two forms of multithreaded architecture were covered—those that interleave the execution of instructions, and those that exploit the advantages of fast context switching. Both forms of multithreaded architecture offer the ability to support fine-grain interprocessor communication, the ability to switch activities to tolerate remote memory access latency, and the possibility of exploiting parallelism in conventional programs. However, the prospects for effective use of multithreaded architectures in support of conventional languages for general purpose computing do not appear great due to the difficulty of partitioning programs written in imperative languages into concurrent threads.

The use of processing elements with interleaved thread execution is an exciting possibility for massively parallel computers for scientific computation. Together with new compilers supporting the SPMD model, supercomputers built on this technology should yield a significant advance in the efficiency of high performance computation.

On the other hand, the SPMD model will not satisfy scientists and engineers for long. The demand for greater generality in parallel computation—unstructured grids, search problems, symbolic computations, and greater flexibility in sharing the physical resources of large computers, will call for parallel computer systems with programming environments of quality at least as good as those provided for conventional machines.

Will it be possible to support high performance scientific computing and general purpose computing within the same system architecture? At present the requirements of the two domains are different and seem to be diverging further. To achieve the merger of these two domains, two problems must be addressed. Firstly, it must be possible to meet the needs of scientific computation for floating point performance within a programming model suitable for general purpose computation. Secondly, a solution to the multiprocessor virtual memory problem is needed that provides an enormous address space within which multiple users may effectively share access to programs and data. The achievement of these goals is a grand challenge, but the payoff could be tremendous. It remains to be seen how soon the issues will be resolved; processor architectures significantly different from those of today may be required.

10 Acknowledgements

The authors would like to thank the MIT Laboratory of Computer Science and the School of Computer Science of McGill University for facilitating this effort. The members of the Advanced Architecture and Program Structures Group of the School of Computer Science at McGill have provided direct assistance. In particular, we owe thanks to Chandrika Mukerji for her assistance in the preparation of the manuscript, and to Kevin Theobald for his help in preparing and

the physical main memory of a computer, and permits running computations that access large amounts of data and program without running out of address space. By separating the names of objects from their locations in physical memory, management of memory allocation is made simpler and more efficient. Also, a portion of the virtual memories of several user processes may be mapped into the same physical memory so that one copy of certain software will serve all users. These motivations for virtual memory carry over to multiprocessor computer systems.

In a parallel computer it is natural to expect that reference to any "main memory object" should be effective regardless of in what node the object is located and regardless of whether the object is in main memory or on disk. The "system" should be responsible for moving objects to facilitate access. In this process, copies of objects would be made and stored at multiple nodes of the parallel machine.

An experiment in shared virtual memory for a distributed computer system has been conducted using Apollo workstations [91]. The virtual addressing code of the operating systems was modified so that a collection of processes (running for a particular user) on each of the workstations mapped virtual memory addresses to a common set of *abstract pages*. Each abstract page is either in write status or in read status. The last node to write on the page is considered the *owner* of the page. Copies of pages are made in response to read transactions where the owner of page is a remote node. Several policies for automatically updating copies of abstract pages to ensure coherence were evaluated for test problems executed as multiprocess programs on several nodes: a sparse matrix solver, matrix multiplication, and a sorting procedure. Good speedups were reported. While this approach avoided the need to use explicit communication, the test problems were coded using explicit primitives for process coordination.

The DASH multiprocessor [24, 90] also implements a virtual memory over many processors. By using cache lines instead of pages as the unit of data movement, and using cache misses instead of page faults to indicate need for retrieving blocks from remote nodes, more efficient operation should be obtained. The DASH system also uses a directory protocol to achieve coherent memory in place of the page ownership policies evaluated in the Apollo experiment. Nevertheless, as in the Apollo experiment, the virtual memory address space is limited to that supported by the processor chips being used, and the virtual memory is provided independently for each user of the system.

The ideal virtual memory for a parallel computer would provide a unique virtual address for every object held online, and would support a sound program execution model. However, no contemporary distributed system implements a virtual main memory in which objects have unique addresses (names) over the processor nodes of the computer system. The Multics system [32] achieved this with some limitations, but remains an illuminating example of the powerful programming environments that are possible. The IBM RISC/6000 processor [95] and the newly introduced Alpha chip from DEC [50] include addressing support capable of realizing such a large virtual memory, but it remains to be seen whether a sound execution model will be achieved. In particular, the need for supporting virtual memory for multiple users in a multiprocessing environment has not been carefully considered.

partitioning algorithm must ensure that the resulting threaded program will be deadlock-free, an issue that has been noted and resolved by Iannucci [74].

Once a code module has been partitioned into threads, code generation for each thread must be performed. The problem of register allocation is similar to register allocation in compilers for conventional machines. In machines where register access may be shared by several threads, as in the HEP, the super actor machine [72], and Culler's TAM [31], register allocation poses interesting optimization problems for compiler writers.

The presence of multiple processors introduces the possibility of optimizing remote memory references. To reduce network traffic and reduce the number of long latency remote accesses, it is important to avoid duplicate references to the same remote location (assuming remote data is not cached). Opportunities for moving data between nodes in blocks of two or more words should be exploited. A more ambitious approach requiring global program analysis would be to attempt to automatically determine an optimal distribution of data structures over the nodes of the machine.

Scheduling issues are present at two levels for mutithreaded machines. The scheduling of threads is generally done at execution time and hardware scheduling of threads is characteristic of the multithreaded machines that use interleaved instruction execution. In other multithreaded machines system software schedules threads using hardware support for fast context switching. In TAM [31], the compiler takes control of synchronization, scheduling, and storage management, making use of an instruction set where such operations are explicit. At a lower level of detail, rearranging the order of instructions within a thread may permit a better use of a register set, and could ease the problem of achieving full latency hiding. The problems of instruction order within threads and register allocation interact, and the problem of formulating and solving the combined optimization problem has been attacked by several investigators [55, 58, 56, 17]. Although research efforts on compiling programs for general purpose computing from functional programming languages with implicit parallelism has made impressive strides, it is not clear how these techniques will make their way into the practice of general purpose computing.

8.7 Virtual Memory for Multiprocessors

Virtual memory has become an expected feature of computer systems offering modern programming environments. Processes running in these machines use a large address space divided into pages. As an operating system allocates main memory for application programs, operating system data, and new data structures, pages of information not recently referenced are *paged out* to space on a disk storage unit. Modern RISC processors implement virtual memory by means of page tables held in main memory and a hardware *translation lookaside buffer* that speeds access to the entries of the page table much as a cache speeds up access to words from within pages. Use of a paged virtual memory permits several independent applications to share

target computer depend on the nature of the high-level language and the program execution model.

In massively parallel scientific computing the dominant program execution model is the single-program, multiple-data (SPMD) model. Presently, there are few compilers that support the SPMD model for programs written in a high-level language, the notable commercial product being the Thinking Machines Fortran compiler [4, 109] supporting the Fortran 90 array extensions [129]. Most vendors of commercial massively parallel machines expect application programmers to utilize the SPMD model by means of explicit message-passing commands added to the Fortran or C language. Nevertheless, several projects are developing the capability to automatically derive SPMD code from high-level programs without explicit parallel processing commands. These experimental compilers make use of information about the data structures of the application and how they may be mapped onto the nodes of a parallel computer. The information is provided to the compiler by its analysis of the source program [131, 43], from annotations added to the program text [71], or as advice given interactively [131]. The compiler automatically constructs the mappings of data arrays to the processing element memories and generates appropriate communications code.

Most research on identifying threads in high-level language programs has been done for functional programming languages with implicit parallelism. When the parallelism of a computation is expressed implicitly in the high-level program, the important task of the compiler is forming the threads and arranging their scheduling on the target machine. The kind of program module to be partitioned depends on the program execution model of the target computer. In the case of language-based models, the module to be partitioned is the body of a function or loop. In the case of a multiple process model, the cost of process creation usually demands that partitioning be applied at a more global program level.

If the target machine does not interleave the execution of threads, the issues of compiling are essentially those of compiling threads independently for fast execution. It is important to keep the average length of threads long to amortize the cost of thread switching. For machines that interleave the execution of threads, there is a basic tradeoff to be made in partitioning: it is desirable that there always be sufficient active threads to fully utilize processing element resources, and thereby achieve complete latency hiding. On the other hand, creating more threads probably means shorter threads and greater thread creation and termination cost. We believe this problem of trading off latency hiding against thread scheduling is inherently easier than the problem of compiling for high performance by superscalar target machines where the criterion of success is often the fastest execution time for a single thread.

There have been several research reports on identifying threads in programs written in functional or dataflow languages [75, 74, 22, 27, 96, 31, 124, 123]. Nikhil has proposed a *compiler flow graph* formalism [98] in which threads are viewed as the units of code to be scheduled. Based on the experience with P-RISC [97], the approach starts with dataflow graphs generated from a declarative high-level programming language, and compiles them into threads [96]. A

tems is that all memory in a shared-memory system is directly addressable by any processing element, whereas, all interaction in a distributed memory system is accomplished by the sending and receiving of messages. As we have noted, a fine-grain message-passing capability may be used to implement global memory access. With the introduction of multithreaded architecture and fine-grain communication support, it appears that the shared-memory versus distributed memory distinction will no longer be a useful way of classifying multiprocessor computers.

The use of distributed memory in computer systems introduces the issue of cache coherence, which arises from a desire to have all concurrent activities in a multiprocessor see the same memory state. If a cache line is updated at one node, any copies of the line held in caches at other nodes will not reflect the updated value unless special action is taken. A variety of schemes for ensuring cache coherence in multiprocessors have been proposed [8], the most successful being the *snoopy cache* [59, 118, 79] which is used in several commercial bus-based parallel computers. Because the method depends on use of a shared bus, it does not scale effectively for large-scale multiprocessors. *Directory schemes* have been developed to address the problem of coherence in a manner that is claimed to scale to large multiprocessors [117, 28, 3], and the area remains one of active investigation.

Alternative views of the problem are possible. In the first place, most distributed-memory multiprocessors do not have a cache coherence problem because their memories are not treated as forming a common address space and data received from a remote node is not automatically held in a local cache. In a multithreaded architecture, one possibility is not caching any data retrieved via a remote memory access. This is reasonable because the value of cache memory for avoiding latency in a multithreaded processor with interleaved instruction execution is much reduced. If remote memory locations are not cached, there is no coherence problem. If remote memory transactions are implemented using message passing primitives, data management is under programmer control; the programmer (or compiler) may ensure that all use of local copies of data is completed before they could be overwritten. If so, then cache coherence problems do not arise.

The idea that the memory system should reflect precisely the same state to every activity (the goal of coherence protocols) is needlessly restrictive. The requirement is based on the multiprocess model of program execution which is currently the most common basis for reasoning about the correctness of multiprocessor systems. Within this framework, Bitar [23] has surveyed several approaches to "weakening" the requirements for coherence. The choice of a different program execution model, for example a nested function model such as the one that guided the design of Monsoon, will yield a different (and, we believe, simpler) set of requirements for correct memory system operation.

8.6 Compiling Programs for Multithreaded Architectures

A compiler translates programs written in a high-level language into object code according to a program execution model. Thus the problems and issues of compiling for a multithreaded

	Butterfly	HEP	MDP	Monsoon
Global addressing	Yes	Yes	No	Yes
Local access	Yes	Yes	Yes	No
Latency tolerant	No	Yes	Yes	Yes
Interleaved pipe	No	Yes	No	Yes

Butterfly: BBN Butterfly
HEP: Denelcor HEP
MDP: MIT Message-Driven Processor
Monsoon: MIT/Motorola Monsoon

Table 1: Examples of global memory implementations.

in the J-Machine [35, 37]. By using any of these schemes it is possible to reduce the overhead of interprocessor communication to no more than a few memory cycles per transaction plus the cycles needed to access and store the message content.

Computations run on a multiprocessor need to exchange data. Two viewpoints have evolved: For computer systems supporting the SPMD computation model, machine code is constructed such that each processing element sends messages containing data values to other nodes in time for their use. A global address space for data is not necessary. More general program execution models require that a node be able to retrieve data from any global address without anticipation by the remote node. The usual protocol is to send a packet containing the global address to a remote node which responds by returning a packet containing the requested data. The range of possible global memory implementations is illustrated by the examples in Table 1. All of these systems employ multistage interconnection networks. All except Monsoon provide direct local access to part of the physical global memory. These systems all support the global address space directly in hardware. Alternatively, a distributed message-passing machine may support a global address space by using large integers as global addresses that are interpreted by software as node number/word address pairs. A study of various latency-reducing and tolerating techniques based on experience with the Stanford Dash multiprocessor appears in [62] and is included in this volume.

The BBN Butterfly [121, 16] is the system closest to being a classical shared-memory architecture. Once a path through the network is established, it behaves as a circuit for transmitting a request to a memory unit or transmitting a reply. The processors are not latency tolerant and must idle while waiting for a response from a remote memory unit. The other systems achieve latency tolerance of remote memory references by differing schemes: The HEP through the interleaved execution of threads; the MDP by fast process switching; and Monsoon by use of split-phase memory transactions coupled with fast thread activation.

The traditional distinction between distributed and shared-memory multiprocessor sys-

the J-Machine and Tera, each of which may be guaranteed a certain share of processor capacity (instruction issue cycles). External events with high priority, and crucial program segments may be given exclusive access to one queue.

Several of the systems described earlier make use of *futures* to implement lenient evaluation of function arguments, among other uses. A future is a suspended process waiting to be activated by a demand for its result. Futures are usually implemented dynamically, that is, a record (requiring a storage allocation step) is created at the time the suspended process is set up. Futures are advocated for use in delaying evaluation of function arguments that might never be used in the called function activation. They may also be used in the "lazy" construction of data structures and data streams. A future is a synchronizing primitive if the suspended process is permitted to proceed (*eager beaver* evaluation) because use of its result is conditioned on occurrence of a demand for the result and completion of the computation by the future.

Lenient (non-strict) evaluation of arguments has a straightforward implementation in Monsoon using fork and join, and the I-structures in Monsoon support the lenient construction of arrays. Although lenient evaluation is important for implementing stream data types and contributes elegance to programs for certain computations, the importance of lenient evaluation in large-scale scientific computation is an unsettled question.

8.5 Multiprocessor Issues

When processing elements are combined into a parallel computers, new issues arise concerning interprocessor communication and the treatment of distributed memory.

Multithreading and the use of message-passing to implement remote memory access achieve latency-tolerance of remote memory access and make the use of packet-switched, multistage interprocessor networks attractive. The design issues concern network topology, the choice of circuit or packet switching, send/receive protocol, and message format and content. The topology of the network must provide a path from each processor to each other processor. The bisection bandwidth must increase in proportion to the number of processor nodes and the latency of message transmission should increase no faster than logarithm of the number of processors. Circuit switching has seen greater use than packet switching because the software overhead of message send/receive has required long messages to realize high average transmission rates. A consequence is that use of data by the receiving processor is usually delayed until the complete message has been received. Use of multithreaded architecture with fine-grain scheduling allows use of fine-grain message send and receive operations, and may lead to greater use of packet switching, as in the HEP and Monsoon. Fine-grain interprocessor communication protocols can be implemented efficiently by supporting the protocol with special hardware. One example is the scheme sketched in Section 5 for a dataflow multiprocessor. Other schemes that illustrate possible choices are those used in Monsoon (Section 6), Start, and

8.4 Synchronization and Scheduling of Threads

Synchronization is the coordination of activities in computation so that no action takes place earlier than permitted by predecessor actions. Schemes used to implement synchronization include dataflow sequencing, the *fork* and *join* primitives, tags on memory words, and *futures*. Each of these means should be viewed in terms of how well it serves to implement the program execution model used to guide system design.

The *fork* and *join* primitives may be used to express the parallelism within a function body; they are naturally related to dataflow representations of function bodies. In Monsoon fork and join have efficient implementations that consume only a few machine cycles: fork places an activity specifier in the instruction queue; join does a simple count and test.

Tags in memory are used in the HEP as a natural way of implementing producer/consumer synchronization. However, if a long wait is necessary, the process of testing the tag wastes network bandwidth in repeated polling of the tag. An alternative is software supported scheduling of suspended processes, but this does not have efficient support in the HEP architecture, and code created for the machine must be carefully planned to avoid long waits as much as possible. In the Tera machine, a retry mechanism built into its memory modules provides a more efficient polling scheme.

In Monsoon synchronization is architecturally associated with data transmission and availability. A thread may terminate with a value that serves as the left or right operand of an operation at the beginning of a new thread. The new thread begins execution as a continuation of the thread that supplies the second operand. The cost of this mechanism is the space needed to store an operand value as part of every activity identifier. A benefit is that the second operand may be transmitted to the successor thread without being stored and retrieved from frame memory.

Interrupt mechanisms are built into computers to direct attention to routines that respond to external events. In a multithreaded architecture the usual form of interrupt mechanism is redundant; completion of input/output transfers or the receipt of messages may be handled by hardware mechanisms that place an active thread in the execution pool. If thread durations are short (from creation to termination) they do not have to be interruptible. If, in addition, each thread has no dependence on register contents at its beginning, then no provision for saving register state is needed. In the HEP threads may have arbitrarily long duration, and may become blocked for long intervals. They must be interruptible to give high priority activities a chance to proceed.

Some multithreaded computers implement the scheduling of threads in hardware, and first-come, first-served management of the instruction queue is generally used. It may be argued that this policy would prevent real-time response to external events if the effect of an external signal is to place an activity specifier at the end of the instruction queue. However, real time response may be guaranteed by using two or more queues of enabled threads, as in

2. Resetting (flushing) the cache memory (if present) to avoid false hits for the new process.

3. Changing the main memory map (page table) to the address space of the new process.

4. Setting the owner environment for the new process for accounting functions, protection, and file system operations.

Of these costs, the first is relatively small, and the last may be accomplished as a consequence of changing the memory map. If physical addresses are used as cache tags, no cache resetting is required for context switching and register saving/restoring form the remaining portion of context switching overhead.

Early studies for the DASH multiprocessor [128] have tested the suggestion of providing a small number of register sets to support low-cost switching among a group of processes (block multithreading). Because the number of processes waiting for service will often be larger than the number of register sets, the full advantage of having multiple register sets will be realized only if resources are applied to a few of these processes at a time. An alternative is to provide a *context cache* that holds register bindings for variables from several threads, thereby avoiding the need to explicitly save and restore register contexts. A related use of a cache memory to build an efficient instruction pipeline has been reported [100]. Another possibility is to identify *interrupt points* in the machine code where the contents of the register context would be guaranteed (by the programmer/compiler) to have no effect on the future course of the process. At these points a switch between activities may be performed without the cost of saving and restoring register state.

Designing a processing element to support fast context switching has several attractions: more efficient interprocessor communication; lower process scheduling overhead; and the ability to implement lenient evaluation schemes with low penalty. The work of Culler [31] shows that use of cheap process switching makes it practical to implement a lazy functional language on a conventional architecture without a large penalty in performance in comparison to conventional languages. In contrast to processors supporting interleaved instruction execution, latency of floating point operations and local memory access can be tolerated only by exploiting parallelism within the running thread, that is, through RISC or superscalar processor designs.

A limitation of many fast switching schemes is that the collection of processes must be operating in a common address space and for the same user. Extension of fast context switching to more general programming environments may require architecture changes, such as supporting a common address space for all users.

The use of cache memories will help avoid local memory latency, as in conventional processors, except that the working set of memory locations may be significantly larger because threads sharing the processor may be performing unrelated activities.

long-lived entities, alternating between active and dormant states, and subject to interruption when needed to perform higher priority work. A tag, known in the HEP as a *process tag*, is associated with the thread throughout its life. On one PEM, up to 64 threads may be created by making entries in the Process Queue using the HEP create instruction, but going beyond this involves software overhead. In contrast, Monsoon regards threads as inexpensive entities that may be created and terminated according to the needs of active function instantiations. Threads run to completion once their execution commences, and they are not interruptible. In Monsoon, the hardware assigns a tag when a thread is made active and the tag is released when execution of the thread terminates. The number of user threads that may share the processor without software (operating system) intervention is limited to 64 for each processor of the HEP, but is limited only by the size of the address space in the case of Monsoon.

When instruction interleaving is implemented, the role of cache memory changes because greater latency of memory access may be tolerated. Small instruction and data cache memories that reduce main memory bandwidth for local variables and instruction loops may yield a balanced design. The HEP and Monsoon use relatively fast static memory devices for the memories to which processors have local access and there is no cache memory. As the disparity between logic and memory speeds continues to grow, there will be an even greater performance/cost penalty for not dealing with the latency of local memory access.

In conventional processors cache lines holding multiple words are employed on the assumption that words not yet referenced are likely to be referenced soon. Multithreading makes this less valuable because achieving low average latency becomes less important than minimizing the memory bandwidth needed to support computation. Unless the computation has a precisely regular pattern of addressing memory, use of a line size greater than one word will increase the data memory bandwidth demanded by a computation. Instead of building high memory bandwidth by using a large block size, one may increase the number of independent memory transactions processed concurrently. One way to achieve this is to use several interleaved banks of dynamic random access memory.

8.3 Processor Design for Fast Context Switching

In multithreaded processors without interleaved instruction execution, the instruction execution mechanism is conventional and the novelty lies in the mechanism used to "switch context" among processes awaiting execution. The objective is to reduce the time a process must run to amortize the cost of switching control to a new process, thereby permitting a finer grain of process scheduling without incurring overwhelming overhead costs.

In typical operating systems thousands of instruction executions may be required to switch between processes. This cost of context switching has several components:

1. Saving and restoring an activity specifier and register context.

All of these language-based models meet the requirement of supporting parallel program execution because enabled activities in all existing function activations may proceed concurrently: any allowed sequence of actions is a correct execution of the code. Although these program execution models are attractive choices for designers of multithreaded machines, they require users to accept new programming styles and environments.

8.2 Processor Design to Support Interleaved Threads

Interleaved thread execution is intended to exploit greater amounts of parallelism with less complexity than other processing element designs. This processing element organization also promises less overhead for process switching, synchronization, and interprocessor communication than high performance processors for single thread computation.

Processing elements that support the interleaved execution of threads use instruction execution pipelines designed to achieve high resource utilization by sharing logic resources (floating point functional units, for example) among several threads. Each pipeline stage may perform work for a different thread on each clock tick. In this way, latency of operations in one thread need not lead to low utilization of pipeline stages. The mechanism for implementing this was discussed in Section 5 and illustrated by Figure 10. An active thread is assigned a *pipeline tag* that accompanies each instruction as it passes through the pipeline stages. In most designs a register set is reserved for each distinct distinct tag, and the logic of each stage uses the tag to select the appropriate register set. There should be sufficient parallelism in the computation allocated to the processor that sufficient threads are active to fully utilize the pipeline.

In superscalar architectures it is often advantageous to execute instructions out of order. Although the instructions in one thread of a multithreaded processor might be executed out of order to improve single thread performance, this would seem to negate one of the principal attractions of multithreaded design—the simplicity of creating and terminating active threads and handling exceptional conditions. Nevertheless, the ability to initiate floating point add and multiply operations concurrently in one thread may be desirable so that fewer threads are needed to fully utilize floating point resources for scientific computations.

If some actions performed by pipeline stages are of varying duration, such as memory references unsatisfied by cache probes, it may be desirable to structure the execution pipeline so that the work unit just acted on by one stage does not have to be acted on by the next stage in the next cycle. Also, all work units do not have to follow the same flow pattern through the pipeline—a register load instruction does not require action by an ALU stage, for example. Various techniques may be used to accomplish this. The use of buffers for work units between pipeline stages [94] was discussed in Section 5; the use of *reservation stations* [122] is a similar scheme, and is now being used in superscalar microprocessors such as the Motorola 88110 [49].

The association of tags with threads may be permanent or temporary, with consequences for software design, performance, and programmability. In the HEP threads are regarded as

features of such languages as Fortran 90 [129], High Performance Fortran [68], and certain functional programming languages (Sisal [92] and Id [98], for example). The SPMD model lacks generality because its high performance potential is only realized when the computation is on regular arrays of data and a good mapping scheme can be found. In addition, use of the model makes the assumption that a reasonable load balance over the nodes will be obtained by use of the chosen mapping. This is not true in general, yet the SPMD model is important because it is at present the only accepted model for massively parallel computation that serves as a target for compilers of high level languages with implicit parallelism.

Multithreaded processors should be attractive as nodes for a massively parallel machine programmed using the SPMD model. The computation performed by each node often computes many elements of a data structure and these may be divided up among enough threads to fully utilize the processor using interleaved execution. This is in addition to any concurrency in the computation for each data structure element that might be exploited through partitioning into parallel threads. Moreover, the ability of a multithreaded processor to perform fine-grain scheduling can support efficient fine-grain interprocessor communication.

Some of the multithreaded architectures we have described use program execution models based on the semantic requirements of a high-level programming language. The high-level languages used are functional programming languages in which the expression of parallelism is implicit, a choice that permits source programs to be concise and elegant [125]. The language-based execution models all embody a tree of module activations as a fundamental operational structure, corresponding to a nested set of functions in the source program. Two models of this type are the *graph/heap model* [41], and the *unraveling interpreter* [9]. Both models use dataflow program graphs to represent the flow of data and control. They differ in that the graph/heap model defines instruction enabling in terms of the presence of simple tokens on the arcs of copies of the dataflow graph, whereas the unraveling interpreter enables instructions by matching *tags* that identify the graph instance to which a token belongs. These models have guided the design of several multithreaded computers. The Monsoon multiprocessor evolved from work on implementing the unraveling interpreter, but is more naturally viewed as an elaboration of the graph/heap model. The dataflow processor of Section 5, when equipped with base addressing for program and data segments, may also be viewed as an implementation of the graph/heap model. In general, many instruction executions are possible from a typical state in these models, allowing many possibilities for realizing concurrency in an implementation.

The Multilisp model [63] and the P-RISC model [97] are designed to support nested-function languages using sequential threads of instructions. These models differ from the graph/heap models in the expression of synchronization. In the dataflow models synchronization is expressed by a requirement that two (or more) specified events occur before an instruction is eligible for execution. In the models that use sequential threads, synchronization takes place only at specific instructions provided for the purpose. Multilisp relies on *futures* which condition use of a value on the conjunction of a demand for the value and completion of the activity that computes it. The P-RISC model uses explicit fork and join commands.

systems. For a typical user running common service software (editors, compilers, etc.) in the von Neumann model, the amount of concurrency available appears to be small [78, 126]. Other studies, however, have shown that a much higher level of parallelism potentially exists if different programming and execution models are assumed [120].

The success of multithreaded machines employing fast context switching in supporting the von Neumann model seems to depend on whether the fast switching can be used to multiplex the processor among several user jobs. If the switching is only fast between processes running in the same user address space, then achieving high performance depends on the presence of sufficient parallelism within the user job. This appears unlikely without significant changes in programming languages and methodology.

In systems having two or more processors, different program execution models have been used according to the taste of the designers and the architecture of the multiprocessor. Many involve the concept of *cooperating sequential processes* [51]. In some cases the processes run in the same address space and may share variables and data structures. In this case synchronization operations, such as the semaphore commands P and V, are used to coordinate actions and to enforce mutual exclusion in critical sections of programs. In other systems the processes operate in separate address spaces and user-level interprocess communication is supported by message-passing facilities. In these systems, users normally express applications in conventional high-level languages extended with appropriate operating system calls for explicit coordination of the processes. Whether common or separate address spaces are used, these parallel programs are larger than their conventional counterparts, and are more difficult to debug and prove correct.

The HEP uses a program execution model close to a multiprocessor von Neumann model. All processes in the HEP run in the same shared address space, and there are no hierarchical relationships among processes except that each user process has an associated supervisor process. Most of the other systems we have described use models that are *language based*, as discussed below.

The *single-program multiple-data* (SPMD) model [38] is currently gaining wide acceptance for massively parallel scientific computation. The basis of this model is a set of processors (nodes) and a mapping of elements of the data structures of a computation into the processor set. The same machine level program is run at each node with its behavior modified as necessary according to the node's identity. A computation executed according to the SPMD model advances by performing alternate computation and communication phases as if in lock-step synchrony. Often each node performs the computation associated with defining (initial or updated) values of the set of array elements assigned to it by the mapping. In general these computations require input values that have been computed at other nodes. The communication phase serves to move values from the nodes where they are computed to the nodes where there are needed for subsequent computation. The model is implemented by mapping it onto an MIMD multiprocessor, either manually or by using a *data parallel* compilation strategy [71, 131, 82, 85, 43]. The SPMD model provides a good target language for the array computation

In an Alewife machine, each memory word has an associated full/empty bit used for thread synchronization. As in MASA, this synchronizing capability is designed to provide efficient support for futures. A blocked memory access causes a trap. The trap handler may then retry the trapped instruction, or may give control to another active thread, with or without unloading the trapped thread. Futures are supported by a similar mechanism. These aspects of the Alewife multiprocessor are discussed elsewhere in this volume. An Alewife multiprocessor is under construction at MIT, and a run-time software system has been tested using simulation tools.

8 Issues in Multithreaded Architecture

The examples presented above illustrate broad the range of possibilities in multithreaded computer architecture. Here we examine the design space and consider the choices made in actual machines and in recent design proposals, keeping in mind that the overall goal is to implement a chosen program execution model in a way that offers high performance for the least cost. The choice of program execution model is crucial to meeting the needs of applications and is discussed first. Then the architecture of a processing element is considered, where the principal design distinction is whether or not instruction interleaving is implemented. Because this difference affects most other aspects of processing element design, we consider issues of resource multiplexing in separate sections for interleaved and non-interleaved designs. These discussions are followed by discussion of how the scheduling and synchronization of concurrent activities are implemented. Next we consider issues arising in multiprocessor configurations of multithreaded processing elements: interprocessor communication, global memory implementation, and multiprocessor cache coherence. We conclude with observations about compiling programs for multithreaded architectures and about providing virtual memory in parallel computer systems.

8.1 Program Execution Models

A program execution model serves as the interface between the compiler writer and the system implementor. The role of the computer system architect is to design an efficient implementation of the program execution model. The role of the compiler writer is to translate high level programs into the machine language of the program execution model. When the target machine is built of multithreaded processing elements, new questions arise: How well can multithreaded architecture support important program execution models? Are there other program execution models better suited for multithreaded architectures, and will these meet the test of user acceptance?

The principal program execution model for sequential computers is the *von Neumann model*. It consists of a sequential process running in a linear address space. It can serve as the target language for compilers of conventional languages and support their run-time software

second register set is provided so the processor can handle two priority levels without the cost of a full context switch.

In contrast to early message-passing multiprocessors, the J-machine can exploit parallelism at a much finer grain. For example, delivering a message and dispatching a task on the J-machine is three orders of magnitude faster than on the Intel iPSC.

The grid network used for the J-machine uses the ideas of *wormhole routing* and *virtual channels* developed in connection with the Caltech Cosmic Cube [81, 111, 36, 33, 34]. A 512 node J-machine is operational and preliminary results of its evaluation have been presented in [99].

The Alewife Project. Alewife is a large-scale multiprocessor project led by Anant Agarwal at MIT [2], and is intended for symbolic computing using Mul-T, an extended version of the Scheme programming paradigm. The processing nodes are organized in a two-dimensional grid, each node consisting of a processor chip, cache memory with a specially designed controller, local data memory, a floating point unit, and a network switch unit that implements a packet protocol for routing messages over the grid.

Cache consistency is implemented by a directory scheme that permits each cache controller to access up to five pointers to nodes that hold copies of a cache line. To provide for cases that require more than five copies of a cache line, a trap mechanism is provided that allows overflows to be handled by software [29].

A major feature of the Alewife multiprocessor is its use of *block multithreading* to reduce context switching overhead. This idea was evaluated by the DASH project at Stanford and found to be a promising approach to efficient tolerance of memory latency [128]. In block multithreading each processor is equipped to hold context information for several sequential threads that make up an *active* set. Switching between active threads is then very fast compared with switching to a thread not in the active set. By minimizing the occurrence of events that move a thread in or out of the active set, the average cost of context switching can be made low.

The node architecture for the Alewife multiprocessor was chosen to utilize the fast single-thread operation of a commercial microprocessor chip. The processor developed for Alewife is Sparcle [1], a Sparc microprocessor modified to support block multithreading with four program counters and four register sets. A thread is said to be *loaded* when it is a member of the active set. Thus Sparcle can support up to four loaded threads. Switching context between two loaded threads takes only eleven machine cycles.

This fast switching capability is used mainly to provide latency-tolerant access to the shared global memory. Instructions from one thread are executed until a memory operation incurring a cache miss is encountered. When this happens, the processor switches context and executes another active thread while the first thread awaits a response from memory.

Because the number of task frames is fixed, only a limited number of threads may have associated task frames. System software is intended to unload threads to make room for new threads when the supply of task frames is exhausted. Similarly, saved frames are restored as task frames become free.

The design of MASA includes memory tags and hardware support for generational garbage collection in support of the automatic memory management expected by Lisp-like and object-oriented languages.

In MASA, an instruction is subject to a number of trap conditions, including some that are raised when a memory access is blocked by tag bits. When a trap condition is detected, a *trap token* is generated which contains the entry address of its handler. A task frame is automatically assigned for the handler if the trap is effective. A consequence of the availability of a fine-grain trap handling mechanism is that it is possible to construct machine code that offers more parallel threads without incurring excessive overhead. One application of this mechanism is the implementation of futures for the Multilisp programming language [63].

The MASA design has several significant differences from the HEP. The parent/child relationship of register contexts supports efficient argument/result passing and trap handling. The support for synchronization is better matched to the requirements of lazy language semantics; and memory tags are provided in support of run time type management and automatic memory management. These differences are related to the higher level view of programming language support taken by the MASA designers.

The J-Machine. The Message Driven Processor (MDP) [35, 34] has been designed as a processor node for the MIT J-machine—a massively parallel, VLSI multiprocessor architecture intended to support a fine-grain, message-passing style of parallel computation. The principal mechanisms for achieving high performance are the direct activation of message handlers by arriving messages, and a processor architecture that permits fast context-switching.

The MDP is intended to support a fine-grain, object-oriented, concurrent programming paradigm with message passing as the principal mechanism for inter-object communication. In such a system, execution progresses by the sending of messages, each of which specifies a *method* to be performed using message data. Getting messages coupled with and processed by their appropriate handlers at low cost is crucial for efficient fine-grain performance. The overhead of passing messages is reduced by using a dedicated Message Unit (MU). Without interrupting instruction processing by the node processor, the MU can determine whether an incoming message should be executed immediately, or should be queued for later processing.

A novel memory organization includes an associative access mechanism for fast address translation from object names to object locations, and to locate the method to be executed to process a message.

Fast context switching is achieved by using a small register set—four general-purpose registers. The entire state of a context can be saved or restored in less than 10 clock cycles. A

form of *lenient* or *non-strict* interpretation [63]. The combination of the linguistic features of futures and synchronization variables creates a basis on which a variety of user-level parallel programming disciplines may be offered.

In many ways the Tera supercomputer is a commercial version of the Horizon architecture. However, it goes beyond Horizon in its architectural support for virtual memory. The processor design implements a paged address space for programs, and data memory accesses go by way of a segment table. The page and segment tables are separate for each of as many as 16 user protection domains in each processor, but may allow programs and data to be shared among users, yielding a potentially large saving in memory requirements.

The Tera system is being developed as a commercial supercomputer system for a large-scale, multi-user scientific computing environment. A full machine is to have 256 processors with a total of 32,000 threads available.

MASA: Multithreaded Architecture for Symbolic Computing. MASA is a design for a shared-memory multiprocessor intended for efficient, parallel processing of symbolic computations expressed in a parallel Lisp dialect such as Multilisp [64]. The design builds on ideas in the HEP architecture, specifically, the interleaving of instructions from several concurrent threads to tolerate the latency of global memory access.

In MASA a thread is known as a *task*. An active or *loaded* thread is represented in a hardware register set called a *task frame*. A MASA processor has a fixed number of task frames, each having an associated *task frame number* that plays the role of the HEP process tag in binding a thread to a register set. As in the HEP, threads are created and terminated by specific machine instructions.

The proposed processor design for MASA uses a RISC-like load/store instruction set, a four-stage execution pipeline, and includes instruction and data caches. It includes a load/store unit intended to handle multiple memory references to local and remote memory units, and to process memory access requests from remote processor nodes.

The processor is able to issue an instruction from a different thread on each machine cycle. As in the HEP, successive instructions in the pipeline are not normally from the same thread, and the problem of filling pipeline delay slots caused by data and control dependences is not present.

A task frame includes the general registers for the thread, its current instruction address (program counter), and fields containing the frame numbers of a parent and a child thread. Normally, when a procedure call occurs, the caller and callee frames are in a parent/child relationship. Thus a chain of task frames reflects the state of a linear nest of procedure activations. The parent/child relationship also applies to a thread and a trap handler invoked during execution of the thread. The parent and child references in task frames permit procedures to access the register context of their caller or of a child, yielding an efficient argument/result passing mechanism analogous to register windows.

The Monsoon project itself has a successor in a continuing collaboration of MIT and Motorola in the *T (pronounced "start") multithreaded multiprocessor. The new project has defined a multiprocessor architecture using the Motorola 88110 superscalar microprocessor with added on-chip features to support fine-grain communication and scheduling of user microthreads [18]. The architecture is intended to retain the latency-hiding feature of the Monsoon split-phase global memory operations. However, this is to be achieved within the framework of a commercial superscalar processor design instead of the unique instruction-interleaving design of Monsoon.

7.2 Architectures not Derived from Dataflow Models

Here we survey some other projects in multithreaded computer architecture which do not derive from dataflow program models. Some of these projects are exploring architectures that have evolved from shared-memory designs (Horizon, Tera, and MASA), whereas others (the J-Machine and Alewife) are derived from distributed-memory multiprocessor organizations. The close relationship of the program execution models supported by this variety of architectures further illustrates the disappearing distinction between shared-memory and distributed-memory multiprocessor architectures.

Horizon and Tera. The Horizon research project [89] and the Tera commercial supercomputer venture [6] have both followed from Burton Smith's ideas and experiences with the HEP multiprocessor.

The Horizon and Tera both resemble the architecture of the HEP in their use of interleaved execution of instructions from several threads as a means of tolerating latency of access to global memory. In contrast to the HEP, both later designs employ a somewhat horizontal instruction format: Each instruction can issue a memory operation, an arithmetic operation, together with a control operation or second arithmetic operation. Also, each thread has its own register set which is not accessible to other threads. A lookahead field is provided in each instruction to indicate the number of subsequent instructions in the thread that may be issued without waiting for the completion of this instruction. These features of the Horizon and Tera architectures support instruction-level parallelism within one thread, reducing the number of parallel threads a program must offer to fully utilize the processor. To avoid excess network traffic from repeated memory operations blocked by full/empty tags, a delayed retry mechanism is provided in both architectures.

The Tera architecture includes four tag bits in each memory word to support thread synchronization. In addition to the "synchronization variables" of HEP Fortran, the programming concept of *futures* is offered. A future is an expression whose value is being computed. Any activity (thread) that needs the value of the future must be delayed until computation of the value is complete. The most common application of futures is to permit the body of a procedure to begin execution without waiting for argument expressions to be fully evaluated—a

7.1 Multithreaded Architectures Based on Dataflow Models

Advanced design proposals for dataflow multiprocessors inspired from the static architecture described in Section 5 have been developed by a research group at McGill University. The *argument-fetching* dataflow architecture [45] has been refined into an alternative proposal for supporting function activations [57], and this has been followed by the *super actor machine* (SAM) [73] described in another chapter of this book. In the SAM the set of initial operands required by a group of actors is retrieved in advance into a *register cache* so that once execution of the *super actor* is begun it may continue to completion without delays due to latency of main memory references. In addition, sharing of the register cache by several super actor executions may achieve a further reduction in main memory accesses.

Based on his experience with the MIT tagged-token architecture [12], Iannucci combined dataflow ideas with sequential thread execution to define a hybrid computation model described in his Ph.D thesis [75]. The ideas later evolved into a multithreaded architecture project at IBM Yorktown as described elsewhere in this volume. The architecture includes features such as a cache memory with synchronization controls, and a hardware mechanism of processor ready queues with process and packet priorities to reduce software overhead. The architecture also provides features for efficient process migration to facilitate load balancing.

Several projects are developing multithreaded architectures using dynamic dataflow principles as embodied in the Monsoon multiprocessor. The most prominent of these are at the Electrotechnical Laboratory (ETL) in Japan, and the Sandia National Laboratories in the United States [60, 61]. Both have now adopted the "explicit token store" [102] idea, and use frames of storage for the local variables of dataflow graph instances.

The Electrotechnical Laboratory began investigation of dataflow architecture with development of the Sigma 1 multiprocessor [130, 70] with symbolic computations in mind. The Sigma 1 is an implementation of the tagged-token dataflow architecture and uses a fast hardware hash addressing scheme to perform token matching. It is a pure dataflow machine in that each instruction is independently scheduled for execution, and may be regarded as a separate thread of a multithreaded program.

In the EM-4 project, a second phase of dataflow prototype development at ETL, the essential elements of a dynamic dataflow architecture using frame storage for local variables are incorporated into a single chip processor [84, 110]. In this design a "strongly connected" subgraph of a function body is implemented as a sequence of instructions (a thread) that uses registers for intermediate results. The further evolution of this design is reported in [83] and elsewhere in this volume.

Other directions of exploration derived from the Monsoon project include the study of new program representations using sequential threads and the fork and join operations (the P-RISC formulation [97]), and application of fine-grain scheduling of threads in von Neumann processors to support lenient execution of functional language programs [31].

6. Form the result token (if required).

The pipeline operates synchronously with each stage acting on data from the preceding stage on each clock cycle. Since no cache memory is used, the pipeline pulse is determined by the access time of the three memories used by pipeline stages: Instruction Memory, Presence-Bit Memory, and Frame Memory. To achieve a fast machine cycle, static memory is used. Note that the first execution of any instruction that uses the (two-way) join option will drop out of the pipeline at the Token Match stage, and the remaining stages will suffer idle cycles [103].

Normally, several threads are executing concurrently and the next instruction from any thread is available to enter the pipeline. There is no advance issue of instructions—within any thread, the next instruction is not available to enter the pipeline until the previous instruction of the thread has been processed by the last stage of the pipeline.

Preliminary results of performance studies of running programs in the Id functional programming language on a Monsoon multiprocessor machine have been reported in [67].

To summarize, let us compare the Monsoon architecture and the static dataflow architecture of the previous section. If base addressing is used in the static architecture, the two designs support similar models of computation: nested activations of functions. A major difference is that Monsoon supports a single global address space for heap objects implemented by the I-structure units. The local main memory of the augmented static architecture is used for heap objects as well as activation frames and program segments, leading to less latency for accessing locally stored heap objects. The static architecture uses a buffered pipeline designed to tolerate unpredictable latency of local memory accesses. Cache memory is used to reduce the main memory bandwidth requirement. In Monsoon, synchronization is done by a dedicated pipeline stage; in the static dataflow processor synchronization is done by the independent Thread Scheduler module.

The activity specifier for a Monsoon thread consists of the base address of the frame, the location of the first instruction of the thread, and a result value for use by the thread. In the static architecture (with base addressing) the activity specifier consists of the data segment base address, and the offset of the instruction from the base address of the program segment.

7 Other Multithreaded Architecture Projects

The three illustrative architectures described in the previous sections represent only a fraction of the worldwide collection of research projects exploring various approaches to realizing the benefits of multithreaded computer architecture. Here we briefly examine some of these other projects. We first review projects based on dataflow program execution models, and then consider designs that have evolved from conventional multiprocessor architectures.

```
    Fetch [V], label.L;
    Stop;

    IAdd j, Y_BASE;
    Fetch [V], label.R;
    Stop;

label [temp]:
    FAdd [Vl], [Vr];
```

The constants `X_BASE` and `Y_BASE` are the base locations of the two arrays in the I-structure memory.

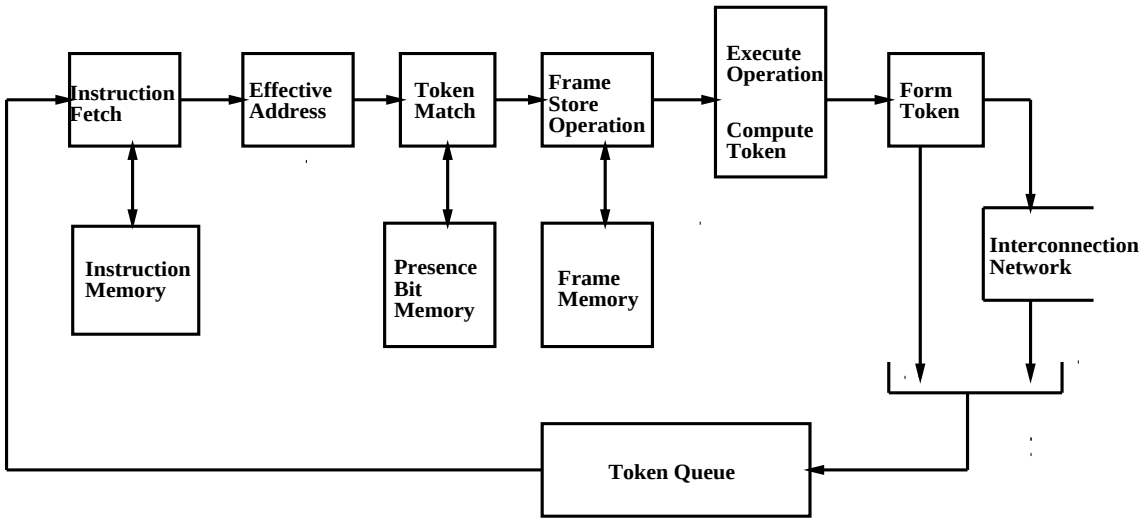

Figure 12: The Monsoon execution pipeline.

The pipelined instruction execution unit of the Monsoon processor is shown in Fig. 12. The stages of the pipeline perform six functions:

1. Fetch an instruction from the Instruction Memory.
2. Perform address calculation
3. Perform a join operation (if specified) using the Presence Bit Memory.
4. Perform a Frame Memory access
5. Perform an ALU operation (three stages).

Dataflow synchronization is supported by a *join option* that is useful with two-operand instructions. The join option implements execution of a (two-operand) instruction just when two threads have computed operand values for the instruction. The Monsoon code for doing this is:

```
thread1:
    [ Compute left operand in [V]]
    Jump thread3.L

thread2:
    [ Compute right operand in [V]]
    Jump thread3.R

thread3 [temp]:
    FAdd Vl, Vr;
```

The jump instructions cause thread execution to continue at the instruction labeled `thread3`. The label in each jump instruction is tagged `L` or `R` to indicate whether the value in the `V` register is to be the left or right operand of the join instruction at `thread3`. The label of the join instruction designates an associated frame location `temp` that serves two functions. One of these is to hold the first operand value to arrive. The second function of the `temp` location is thread synchronization, and is done using two *presence bits* associated with each frame memory location. When execution of a thread arrives at the join, the left or right presence bit of location `temp` is set according to the tag on the label of the jump instruction to indicate arrival of the left or right operand. In the case that control arrives at the join instruction by normal program sequencing (not by a jump) the left presence bit is set. If both presence bits are set, the join instruction is executed using operands from frame location `temp` and the `[V]` register, the presence bits are cleared, and the thread continues. Otherwise, the join instruction is not executed and execution of the thread terminates.

Access to data in global memory is by *split-phase* fetch and store instructions that cause packets to be sent to I-structure units through the interconnection network. To illustrate, evaluation of the expression

```
x[i] + y[j]
```

may be implemented by three threads, the first two of which obtain the operands `X[i]` and `y[j]` and send these to a join instruction that performs the addition:

```
    IAdd i, X_BASE;
```

data structures held in a heap have a long latency and are handled by scheduling execution of a new thread upon their completion, as we shall see. As explained below, the Monsoon hardware is able to interleave execution of up to eight threads in its execution pipeline.

An innovation in Monsoon is its support of *I-structures*, arrays of data in which reads of an element are made to wait until the element is defined by a write operation. We will not dwell here on the details of I-structures. Suffice it to say that I-structure memory also supports conventional reads and writes of global memory locations using the split-phase protocol explained below. For more on I-structures, see [14, 13].

As shown in Figure 11, a Monsoon computer consists of a mixture of Processor and I-Structure Units, interconnected by a packet-switched network. The processor units include memory for instructions and activation frames; the I-structure units contain the memory for large arrays and other data structures.

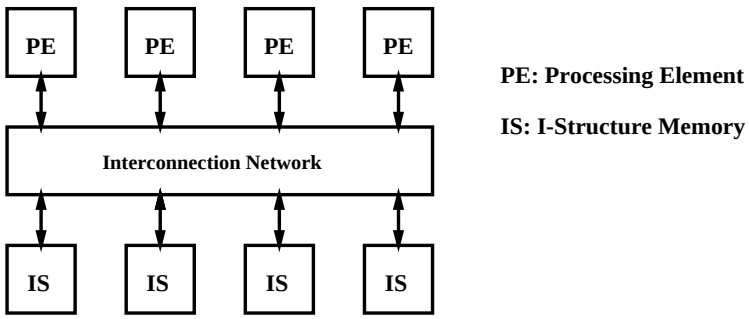

Figure 11: Organization of a Monsoon system.

The following is an example of a Monsoon instruction (the prefixes `I` and `F` distinguish opcodes for integer and floating point arithmetic):

```
IAdd [R1], #(2);
```

The two operand specifiers may refer to registers `[R1]`, `[R2]`, `[R3]`, to immediate constants, or to frame locations specified by symbolic names. The result of an ALU operation is always placed in a special *value register* `[V]`, which may be referenced explicitly by operand specifiers in subsequent instructions of the thread. A move instruction

```
Move [R1], x;
```

moves its first operand into the specified register or frame location. There may be only one Frame Memory reference per instruction.

and entering an activity specifier in the Activity Queue. This can be done in a few machine cycles by appropriate hardware mechanisms [44]. No interruption of active threads is needed. Many communication transactions of this sort may be in progress concurrently so that a high bandwidth network can be fully utilized even though messages are short.

6 Monsoon

The Monsoon multiprocessor [102, 103] is the product of a joint effort of the MIT Computation Structures Group led by Prof. Arvind, and the Motorola company. Its design evolved from the "tagged-token" dynamic dataflow architecture described in [11, 12] and implemented, first at Manchester University [127] and later in the Sigma 1 multiprocessor of the Japanese Electrotechnical Laboratory [130]. The program execution philosophy adopted in these machines derives from the original *unraveling interpreter* [9] which models programs as collections of functions having bodies represented by dataflow graphs. The design and implementation of the first prototype Monsoon processor is described in the doctoral thesis of Greg Papadopoulos [101]. Preliminary results of performance studies of running programming language Id on a Monsoon multiprocessor machine have been reported in [67].

Tagged-token dataflow architectures use an associative *matching store* to determine when instructions are ready for execution. For a two-operand instruction to become enabled, tokens carrying the two operand values must be received. The first token to arrive is held in the matching store until its mate has arrived. Two tokens provide a pair of operands to the instruction if they have the same tag, where the tag consists of a unique identifier of the function activation and the index of the instruction within the code of the function.[2] Because the number of distinct tags is made large to accommodate a large number of potential simultaneously active functions, a sizable associative memory is needed to implement the matching store. For reasonable sizes of matching stores, the possibility of it overflowing must be provided for, or a bound must be imposed somehow on the number of active tokens.

In Monsoon the associative search for matching tokens is replaced by establishing a memory location where each synchronization takes place. The place is a location within an *activation frame* associated with each function activation. The benefit of eliminating the associative search is obtained for the cost of (in general) dynamically allocating storage for an activation frame at each function initiation.[3]

Instead of requiring dataflow synchronization at each instruction of a program, Monsoon provides efficient support for execution of short sequences of instructions (threads). Each thread is coded as a sequence of instructions that access the local variables of its function instantiation from its activation frame, and pass intermediate results using a small register set. References to

[2] In published descriptions of the tagged-token architecture [102, 103] the tag has an additional field intended to index the cycles of an iteration performed by a function.

[3] Preallocation is possible when a static analysis of the program shows that this is feasible.

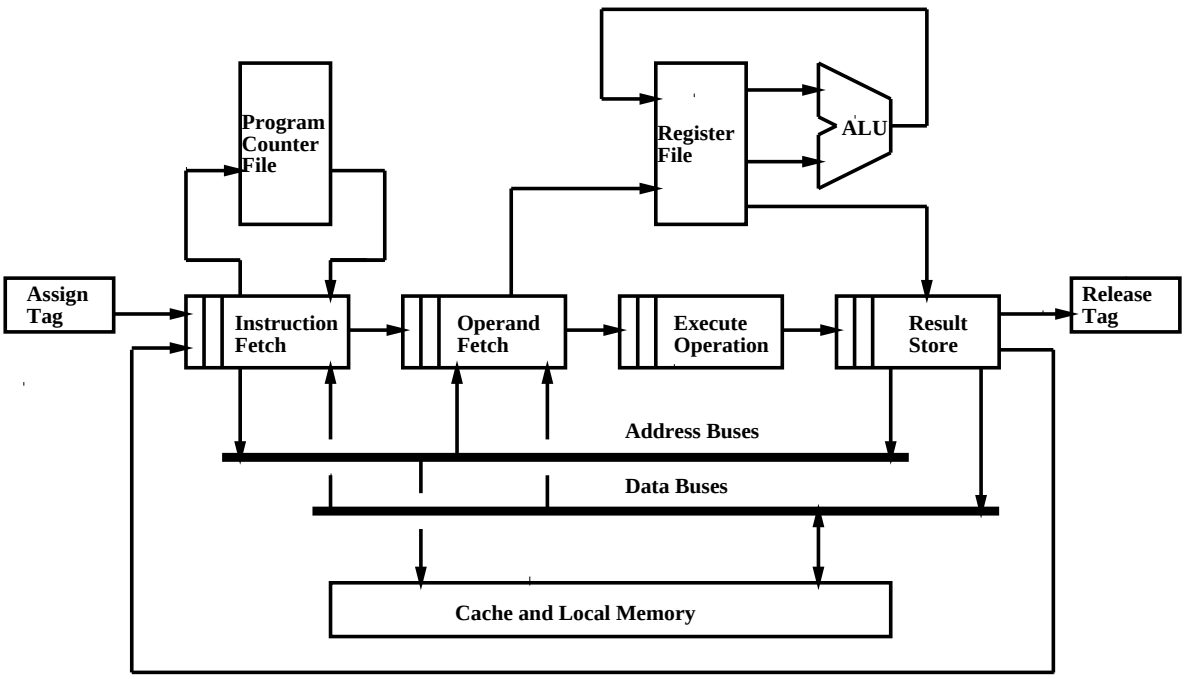

Figure 10: The instruction execution pipeline.

offsets from the data segment base address, and the branch addresses of local signal and execute actions are offsets from the program base address. The two principal base addresses must be passed to the execution pipeline where they are made available to pipeline stages as necessary. Many concurrent activities may participate in carrying out one function instance. Each has an activity specifier consisting of the data segment base address and the relative address of the first instruction of a thread ready for execution. The base address of the program segment may be conveyed as part of the activity specifier, or it may be placed at a standard location in the data segment. Further details are given in [44].

Interprocessor communication has several purposes in the multiprocessor just described. One purpose is to permit activities in one node of the machine to have access to data values held or generated at another node. A second purpose arises if it is desired to spread the body graph of a function over several nodes so that the resources of several nodes may be applied to its execution. In both cases the desired communication may be achieved by sending short messages over the interprocessor network. To fetch a word from another node a message containing the address (base and offset) may be sent to a *daemon* activity in the remote node that accesses the value locally and sends it back in a message to the originating node. If a signal graph describing a (software) pipelined computation [46] is split among several nodes, each arc that is cut by the partition of the graph may be implemented by the transmission of messages between the parts of the function body concerned. In both types of communication, the receipt of a message may be implemented by buffering the message content in the data segment (activation frame)

```
begin:      IMove #(1) -> i;
            Execute loop;
            Quit;

loop:       IMove i -> [X];
            FMult a, x[X] -> [A];
            FAdd [A], y[X] -> s[X];
            IAdd [X], #(1) -> [X];
            IMove [X] -> i;
            ICmp [X], #(n), le:  Execute loop;
            Signal exit;
            Quit;
```

Figure 9: The DAXPY routine programmed using two threads.

may call for an Execute action which causes a thread address to be directly entered into the Activity Queue, or a Signal action that causes the Thread Scheduler to decrement the count for a specified thread. These actions may be done conditionally, as shown by the comparison instruction `ICmp` that causes the action Execute loop if the index in `[X]` is less than or equal to the immediate constant `n`.

Figure 10 shows the organization of the Execution System of the dataflow processor. It is a pipeline organized to achieve a high transaction rate with the local processing element memory system. Up to sixteen threads may have active instructions in the pipeline simultaneously. Separate program counters and register sets are provided for each thread; these are shown as the Program Counter File and the Register File in the figure. A four-bit tag accompanies each instruction through the pipeline and serves to select the appropriate program counter and register set. The Assign Tag block assigns a free tag to a thread at the beginning of its execution; the Release Tag block makes a tag available to other threads once the last instruction of a thread has completed. The figure shows four stages in the pipeline, each with an input buffer for work units requiring services of the stage. The buffers are provided because the work of a stage may take a variable amount of time depending on whether memory references make hits in the cache memory, or depending on the complexity of an ALU operation.

By adding base addressing to the architecture just described, the ability to implement multiple and recursive instances of functions may be provided. This is accomplished by viewing the main memory as holding program and data *segments* in which instructions and data values are addressed by relative addressing from the base location of the appropriate segment. An active instance of function execution then involves a program segment containing instructions of the threads that form the body of the function, and a data segment (known also as a *stack frame*) that holds local variables of the function. Data references in instructions are coded as

be accessed and the DAXPY computation begun. The action `Create (1,n) -> z` allocates memory for the array `z`. (The allocation might be done before program execution, in which case array `z` would be represented by a constant value equal to the address of the base location of the array.)

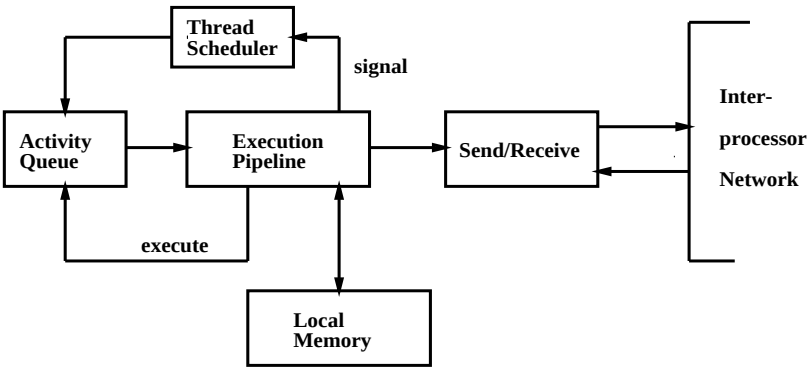

Figure 8: A multithreaded dataflow processing element.

A processing element architecture suited to perform computations expressed as signal graphs is shown in Figure 8. A multiprocessor machine using this processing element would consist of many processing elements interconnected by a packet-switched, multi-stage network. In contrast to the shared-memory arrangement of the HEP, each processing element has its own private memory for data and instructions, and the network is used for uninterpreted messages transmitted between processing elements. The local memory system of each processing element includes cache memories for instructions and data.

Instructions correspond roughly to the nodes of a signal graph and are grouped into threads. The instructions of a thread are executed in sequence without synchronization with actions of other threads. During the execution of a thread, a set of registers may hold data values from one instruction to the next in the thread. Thus the body of the DAXPY loop might be coded using two threads as shown in Figure 9. In this, `[X]` and `[A]` are registers holding integer and floating point values, respectively; lower case identifiers denote memory locations for scalar values, the base addresses of arrays, or the first instruction of a thread. The form `x[X]` means the address obtained by adding the contents of register `[X]` to the contents of location `x`; `#(n)` denotes a literal constant.

For each thread, the Thread Scheduler maintains a count of how many further signals must be received before the thread may be activated. When the count reaches zero, the address of the first instruction of the thread (its activity specifier) is placed in the Activity Queue, and the count is reset. Entries from the Activity Queue are passed to the Execution Pipeline.

When chosen for execution, the constituent instructions of a thread are fetched and delivered to the execution pipeline. As illustrated by the example, instructions in a thread

```
function DAXPY
    (x,y: array[real]; n: integer returns array[real])

    forall i in 1,n
    return array of a * x[i] + y[i]

end function
```

A signal graph representation of this routine is given in Figure 7. In this graph, each node is a box inscribed with a computational step (action). A conventional three-address format is used in which operand and result fields contain the names of program variables (memory locations). For example,

```
t  +  v  ->  w
```

denotes the action of adding the values at locations t and v and storing the result at location w.

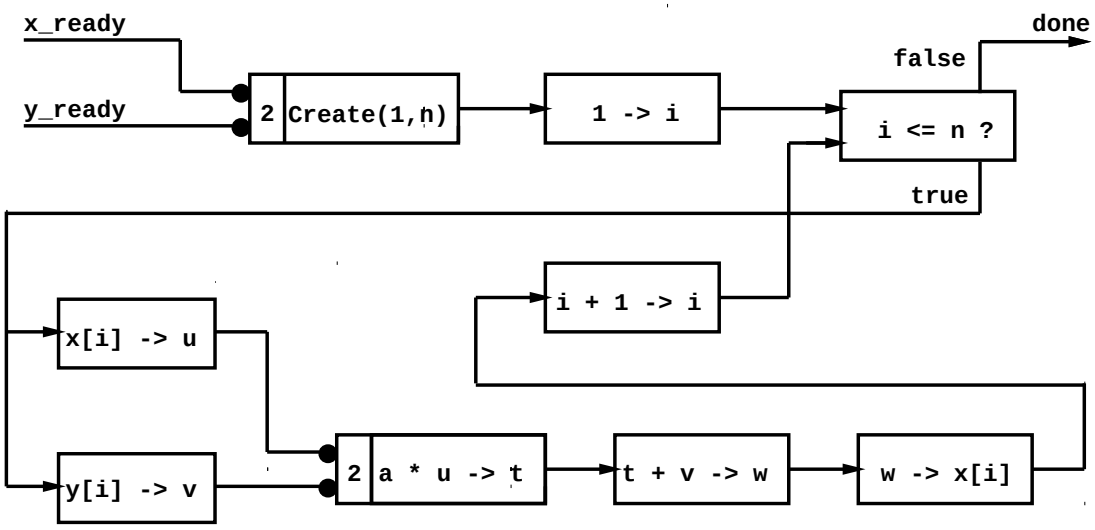

Figure 7: A signal graph for the DAXPY routine.

The arcs of a signal graph control the execution order of the nodes. Each arc conveys a *signal* to its destination node upon completion of action by its source node. Some nodes in Figure 7 have count values shown at their left edges. These indicate the number of signals that must be received before the node may be executed (and its action performed). Nodes that do not show an explicit count field, have a default count value of one; they are performed upon arrival of one signal. The signals `x_ready` and `y_ready` affirm that the two arrays `x` and `y` may

pipeline gaps caused by memory access latency due to cache misses, or by data dependencies between operations within a single thread. The benefit of program locality is realized through use of a large register set and support of local access to program memory and a portion of data memory.

5 A Dataflow Architecture

Work on dataflow computer architecture has evolved from the use of dataflow program graphs to represent and expose the parallelism in programs. In a dataflow graph [39, 41], operations are specified by actors that are enabled just when all predecessor actors have completed their execution. The dependence relationships between pairs of actors are denoted by the arcs of a graph which may be thought of as conveying result values of an actor to successor actors where the values are used as operands. A dataflow graph corresponds to a mathematical function in that it describes a functional dependence of output values on input values. Decision and control actors are included in dataflow graphs to represent conditional expressions and iterations. Arrays and other data structures may be constructed, analyzed, and modified (if desired) by appropriate dataflow actors.

Dataflow computer architecture developed from the concept of using dataflow program graphs as a machine-level program representation. Two forms of dataflow architecture have become known: In a *static architecture* the *tokens* passed from one instruction to another contain only the result value of the source instruction. In this scheme there can be only one instance of a dataflow graph in execution at any time. In a *dynamic dataflow architecture*, tags are associated with values so that values associated with different activations of a function may be distinguished.

The dataflow architecture discussed here has evolved from static dataflow principles, and is proposed as the node architecture for a scalable massively parallel supercomputer for scientific computation. The architecture is presented in greater detail in [44], and is a refinement of the ideas presented in [45]. A second dataflow example, the Monsoon multiprocessor, evolved from dynamic dataflow principles and the requirements of a general purpose functional programming language, and is described in Section 6.

Early proposals for dataflow computers suggested that machine instructions would be in (nearly) one-to-one correspondence with the actors of dataflow program graphs [48, 30, 42, 47]. The design discussed here departs from this view. Instead of designing hardware to mimic data flow graphs, the objective is to define hardware structures that can efficiently implement the semantic content of dataflow program graphs [41]. The architecture departs from a direct implementation of dataflow graphs by having instructions fetch data from memory instead of having instructions deposit operands in "operand receivers" of successor instructions. The design follows a modified model of dataflow computation called *signal graphs*. To illustrate, the familiar DAXPY routine may be written in the Sisal programming language as

instructions have an execution time that is both variable and longer than that of normal instructions, a special mechanism is used to support their out-of-order completion, a mechanism that also applies to the arithmetic division operation. A second set of task queues and "snapshot register" is used to schedule the completion processing of asynchronous instructions.

The Thread Control Functional Unit can modify the program counter of a thread as required by branch instructions, create a new thread by selecting and filling a free PSW entry in the Process Queue and entering its process tag in the appropriate task queue, or terminate a thread by freeing its entry in the PSW file. This unit supports HEP instructions that create a new thread (Create) or terminate a thread (Quit).

In the HEP several threads may share access to register memory of the PEM, leading to the need to provide means for threads to coordinate their access to registers. For this reason, registers have *state descriptors* that include full/empty flags. These may be tested and set in the Execution Pipeline for instructions that access them. To execute an operation, the source operand registers must be in the "full" state and the result register must be in the "empty" (or a "reserved") state. These conditions are checked at the beginning of instruction execution and execution of the instruction is deferred if they are not met. When an instruction is successfully completed, its result register will be in the "full" state.

The full/empty tags on data memory words may be used to implement producer/consumer relationships among threads. However, there is no architectural provision for waking up a thread that has suspended itself waiting for a memory tag to change state. Thus a consumer waiting for "full" or a producer waiting for "empty" must retry a memory operation periodically.

The HEP program execution model is supported through extensions to Fortran for process creation and termination and for synchronizing activities of several threads. HEP Fortran includes a Create statement that causes a subprogram to be executed by a new thread. The Return statement of a subprogram called in this way causes the thread to terminate. HEP Fortran also supports "asynchronous variables" that allow the Fortran programmer to write code that utilizes the full/empty tags of data memory to implement producer/consumer relationships. When an asynchronous variable appears on the left-hand side of an HEP Fortran assignment, the assignment operation is delayed until the memory location becomes empty, then the assignment is done and the location set to "full". A right-hand-side reference to an asynchronous variable waits until the location is "full", then reads the location and sets it to "empty". With this minimum modification to standard Fortran, HEP Fortran has been used to write substantial parallel programs for practical problems. Besides HEP Fortran, an implementation of the functional language Sisal has also been done [5].

The peak performance of a PEM is determined by the maximum rate of instruction issue, one every 100 ns. This rate is achievable if there is sufficient parallelism in user jobs to keep at least eight threads continuously active in the Execution Pipeline even though several threads may be delayed by data memory transactions. Busy-waiting on asynchronous variables will degrade performance through unnecessary use of pipeline slots. The machine is not subject to

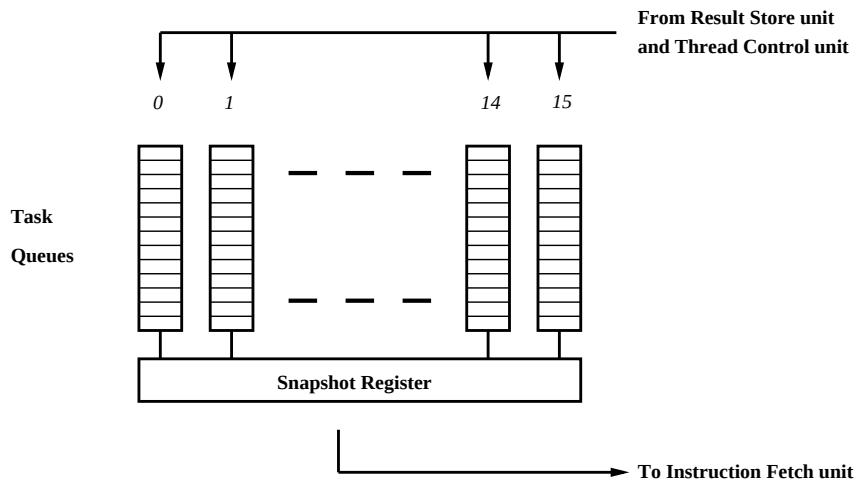

Figure 6: The HEP Instruction Scheduler.

An issued instruction is entered into the execution pipeline. The pipeline has eight stages, so a normal "synchronous instruction" (any register operation except divide, and data memory accesses using the local path) takes 800 ns to flow through the pipeline and reenter its process tag in a task queue. For ease of explanation, we have simplified the pipeline to five stages: Instruction Fetch, Effective Address, Operand Fetch, Execute Operation or Data Memory Access, and Result Store. (In the actual hardware some of these functions are split between several pipeline stages).

The functions performed by the HEP pipeline are similar to those of the generic RISC architecture described in Section 2. The Instruction Fetch stage uses the instruction address from the PSW to access the Program Memory. The Effective Address stage checks addresses in the instruction against limits contained in the TSW and performs any address arithmetic needed to obtain effective addresses for register or constant memory. The Operand Fetch stage obtains data from Register and Constant Memory for use by scalar operations, for a data memory store operation, or as index values for indexed addressing. The Result Store stage writes the results of a scalar operation, or a word read from data memory into Register Memory. It also adds the process tag to the tail of its task queue. In the Instruction Fetch and Effective Address stages the process tag is used to access the PSW, and, by using its task field, to access the TSW. One additional function performed by the pipeline is to increment the instruction address in the PSW. Several "trap" conditions prevent incrementing the instruction address and cause activation of a thread in the corresponding supervisor task.

The Data Memory Controller dispatches messages through the interconnection network for data memory read or write operations that are not local or that test the full/empty tag of the memory word. The handling of these requests to data memory modules is pipelined so that many memory transactions may be pending simultaneously. Because these "asynchronous"

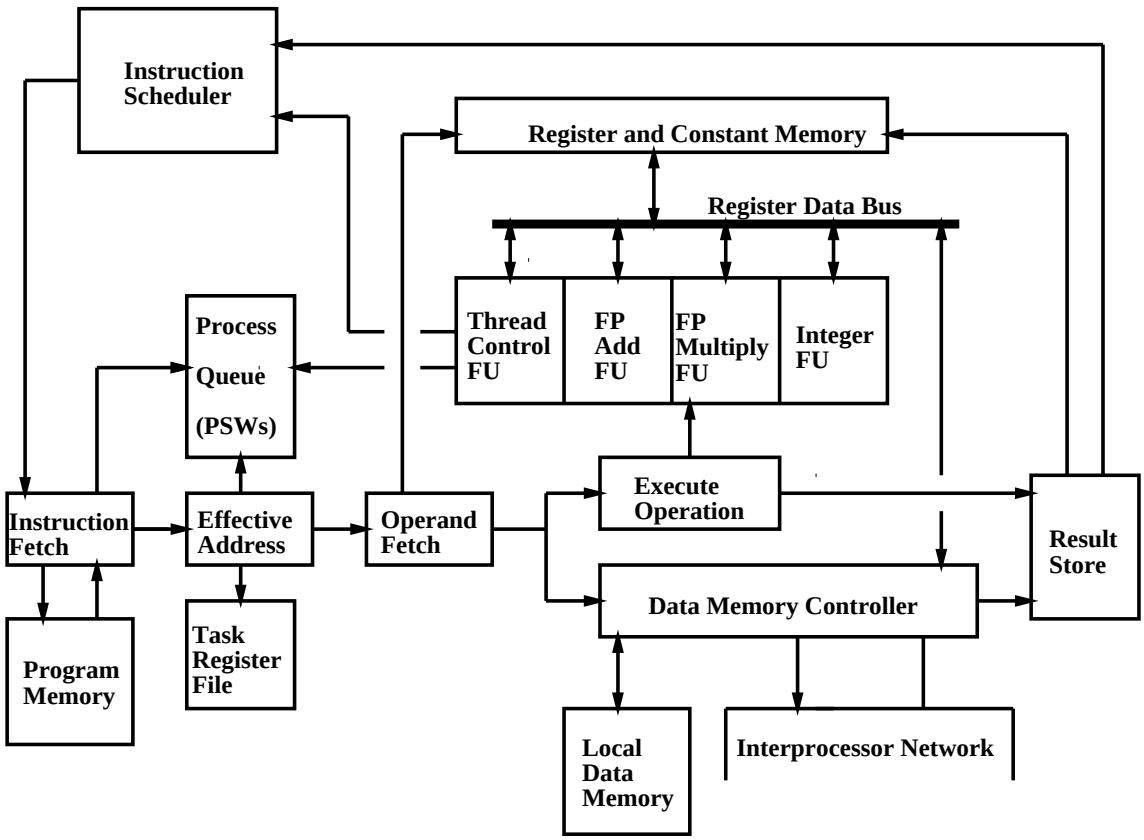

Figure 5: The HEP Processing Element Module.

for its threads. Each computational task always has an associated supervisory task. Therefore one PEM can support at most eight user tasks. A PSW contains a 4-bit task identifier, an instruction address (program counter), and addressing offsets for the register memory and constant memory to facilitate implementation of reentrant programs.

In the HEP machine many instructions may be in various stages of execution simultaneously, including the pipelined execution of scalar operations and pipelined data memory transactions. The seven-bit process tag follows the path of instruction execution through the machine and serves to connect thread execution with needed resources and for entry into a task queue upon completion of instruction execution.

As shown in Figure 6, the Instruction Scheduler contains a group of 16 *task queues*, each of which can hold up to 64 process tags in a FIFO discipline. A "snapshot register" holds the head item from each of the sixteen task queues. Successive process tags for instructions to be issued are taken from the snapshot register in round-robin fashion; when exhausted, the snapshot register is refilled from the heads of the task queues. This mechanism is intended to provide a reasonably fair distribution of processor capacity over the set of active jobs (users).

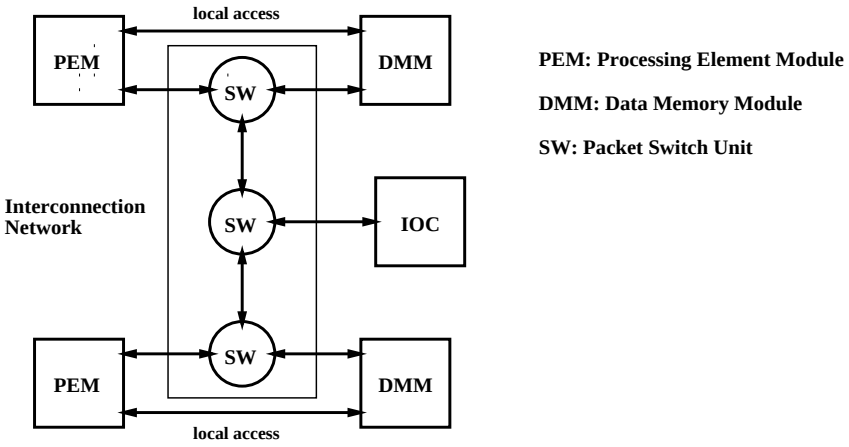

Figure 4: Structure of a HEP multiprocessor system.

ns, which establishes the rate at which messages flow through the network and also the rate of operation of the Execution Pipeline of each PEM.

As a shared memory machine, the physical data address space, which may be as large as 2^{32} 64-bit words, is divided equally among the several data memory modules, and every location is accessible from every processor through the interconnection network. There is also a direct path from each processor to one local DMM which allows fast synchronous access to a portion of the data memory. Each PEM has a separate program memory (having up to 2^{20} words) for holding instructions, which are treated as read-only during user program execution.

The programming model supported by the HEP hardware consists of a collection of tasks, each having several processes that execute independent sequential program threads. The threads may coordinate their concurrent activities by means of full/empty tags provided for every word of data memory and for each of the 2048 fast data registers in each PEM. The creation and termination of threads is supported by specific machine level instructions.

The processing element (Figure 5) is organized to support interleaved execution of instructions from threads which may belong to the same task or to different tasks. Each thread has access to *program memory* to fetch instructions, access to *data memory* through the interconnection network, and has direct access to the local portion of data memory. In addition, each thread has access to portions of a 2048-word *register memory* and a 4096-word *constant memory*.

Tasks and threads are organized in each PEM in the following structure: The PEM has a *task status word* (TSW) register for each of sixteen tasks, and a Process Queue that holds 128 *process status words* (PSWs), identified by 7-bit *process tags*. The TSW contains base and limit addresses for each of the four types of memory: data memory, program memory, register memory, and constant memory; thus the TSW of a task defines a protection domain

other threads) have occurred. Multithreaded processing elements differ in whether activity specifiers are recognized and supported in the hardware, and in the means provided to perform synchronization, for example, spin locks on memory tags (HEP [115, 114, 88]), two-way joins (Monsoon [102]), I-structures (Monsoon [102]), and futures (MASA [64]).

A thread of control is similar to the notion of a *process* widely used in multiprogramming computer systems. The main difference is that a thread may be directly supported (i.e., visible) at the architecture level, whereas a process in a multiprogramming operating system is not, and is implemented by a software layer of the operating system. The start, suspension/continuation, and termination of threads may be directly implemented by machine-level instructions.

3.3 A Brief Historical Note

The idea of applying a single processor to more than one instruction sequence is an old idea. It was used in the Honeywell 800 machine to support running as many as eight input/output limited jobs simultaneously. The purpose was to improve processor utilization for computations limited by input/output (tape) transactions. Another early example is the peripheral processors of the CDC 6600 [122]. There were twelve such peripheral processors implemented by time-slice sharing of common fast CPU and memory logic. Here the motivation was to benefit from the fast technology developed for the 6600 main processor. In neither case was memory latency a significant issue, nor did these early machines support cooperative multiprocessing within a job.

4 HEP: The Heterogeneous Element Processor System

The Heterogeneous Element Processor (HEP) System is a MIMD shared-memory multiprocessor system developed by Denelcor Inc. between 1978 and 1985 [40, 115, 114, 88]. It was the first commercial computer system employing multithreaded architecture, and was designed for high-performance parallel execution of large scientific applications.

A HEP system (Figure 4) consists of up to 16 Processing Element Modules (PEMs) and a number of Data Memory Modules (DMMs). These units are interconnected by a synchronous, pipelined, multi-stage interconnection network that supports transmission of requests from processing elements to memory units and input/output controllers, and delivery of responses back to the processors. The network is built of three-port, bi-directional switch units and uses a novel protocol: At each switch, a message arriving at any of the three ports is sent on to another switch one-half a clock tick later. If there is conflict between two messages, one of the messages may be sent to a switch node not on the shortest path to its destination. If such a message reaches an incorrect destination, the receiving node reenters the errant message back into the network instead of sending a message of its own. The basic system clock period is 100

to execute the next instruction the *register context*. The register context plus the activity specifier make up what is usually called the *context* of a thread. The *contour model* of parallel computation studied by John Johnston [77] and Daniel Berry [21] contained the notion that an activity was specified by an instruction pointer and an environment pointer. Thus, the processor state of a STM can also be characterized as the combination of an activity specifier and an associated register context.

3.2 Multithreaded Architecture Models

A *multithreaded architecture model* differs from the single-threaded model in that there may be several "loci of control", and as a result there may be many enabled instructions which all are candidates for execution. Computations are moved forward by the execution of enabled instructions. Therefore, a characteristic feature of an abstract multithreaded architecture model is that it contains a *set* of activity specifiers, one for each thread of execution.

A realization of the model (a multithreaded computer) generally employs registers to gain efficiency. A multithreaded processing element typically has many hardware register sets, each of which can hold the register context and activity specifier of a thread, and a mechanism for fast switching among them (at the cost of very few, if any, main memory access cycles).

Thus, the state of a multithreaded machine has the following components:

- *Memory State*: The main memory of the machine. This is a mapping of the address space into the set of values that may be held by a memory location.
- *Processor State*: This usually includes two principal components:
  - A collection of activity specifiers, each consisting of a program counter and a stack pointer.
  - A collection of register contexts.

Multithreaded architectures are distinguished by their hardware provision for holding and utilizing multiple activity specifiers and register contexts. The details of these architectural features can have serious impact on the efficiency achievable in implementing different program execution models; hence a strong correspondence between architecture and program model is essential to a successful architecture. Further discussion of program execution models is be given in Section 8.

Another aspect that distinguishes among architectures for multithreaded computers is the kind of hardware support provided for thread synchronization, that is, the mechanisms whereby an activity by one thread may be delayed just until two or more events (actions by

3.1 The Single-Threaded Architecture Model

Today most processor architectures are *single-threaded architectures*. In the abstract, a single-threaded machine (STM) moves a computation forward, one step at a time, through a sequence of states, each step corresponding to the execution of one instruction. A unique feature of sequential program execution is that there is a single "locus of control". At each state there is only one candidate instruction for execution (we call it an *enabled* instruction), and its execution (which we call *firing*) will cause a transition to the next state.

The "flow of control" of a typical sequential machine program, when it is derived from a modern high-level language program, is governed by the nested invocation of a collection of procedures, as well as conditional tests for branching between "basic blocks" within instances of procedure execution. The nested invocation of procedures is supported by a sequential stack model.

The state of a STM consists of the *memory state* and the *processor state*. The memory state is defined by the contents of the main memory of the machine, and often comprises sections for program memory, data memory, and the *stack* that holds activation frames for each active instance of a procedure.

A processor state usually includes several principal components:

- an *instruction pointer*, the address of the next instruction to be interpreted.
- a *stack pointer*, the address of the current activation stack frame.
- a set of *register contents* that retain intermediate values between instructions.

The instruction pointer uniquely identifies the next enabled instruction—hence the control for the next state transition. Processor state may also contain a program status word that includes a condition code for testing by a future branch instruction, and bits providing information relating to process management. Note that the machine may have cache memory for instructions and/or data, but that cache memory is designed to be "transparent", leaving the abstract model invariant.

One should distinguish between features essential to support a program execution model and features included for reasons of efficiency. For example, we may distinguish between that portion of the context (processor state) that is the minimum information needed to define the next state transition of the model, and that portion corresponding to features built into the machine to achieve high performance. The minimum information to specify the state transition of such a single threaded execution model is characterized by what may be called an *activity specifier*. In contemporary microprocessors, the activity specifier usually consists of the program counter and the stack pointer. Let us call the remaining processor resident information needed

- *cost of synchronization:* "Waits for synchronization events often require task switching, which is expensive on von Neumann machines. Therefore, only certain types of parallelism can be exploited efficiently."

In contemporary RISC machines and superscalar microprocessors, memory latency is avoided for most accesses by use of cache memories. However, the first access of a loop or first use of a routine will likely result in a cache miss that causes a pipeline gap. Such gaps will also occur if a line in a cache is displaced by other data before access to it is repeated. Accesses to information held in the local memories of remote processor nodes generally leads to greater latency due to network delays. Furthermore, such remote memory latency is often unpredictable, rendering static scheduling techniques for filling pipeline delay slots ineffective.

In massively parallel computation where the performance of floating point operations is critical, good utilization of the execution pipeline is difficult to maintain using current RISC architectures. This is because insufficient parallelism can be identified in a single instruction stream to avoid pipeline gaps.

Current microprocessors have little special support for the synchronization of concurrent activities. Software overhead for process switching, message send/receive, and other primitive operations for coordinating asynchronous actions is so large that fine-grain coordination is too inefficient to be used. The consequence is grave difficulty in automatically constructing efficient parallel programs from application code expressed in a high level language.

Mike Johnson [76] suggests that superscalar architecture is not the best approach to match machine parallelism to instruction parallelism for "scientific" (as contrasted with "general purpose") applications. Perhaps this is because superscalar designs, while able to increase the absolute performance of a microprocessor, are not the most cost-effective approach to implementing parallelism. Multiprocessors are better, however they are ill-suited (at present) to general purpose computing. Superscalar architecture, by itself, offers no latency tolerance mechanism to support global memory access. Therefore, the two fundamental issues set forth above not only remain, but may be even more difficult to handle. This is because multiple-issue architectures are more complex, leading to a much higher cost if context switching is required to handle process synchronization and to avoid inactive intervals during remote memory access operations.

3 Multithreaded Program Execution Models

In this section, we introduce some concepts that will help comprehend the designs of multithreaded computers and the program execution models they implement. First we review features of a single-threaded processor architecture model for comparison with a multithreaded architecture model discussed subsequently.

pendences and anti-dependences must be recognized and considered in the instruction issuing logic. The *scoreboard* used in 1970 in the CDC 6600 machine [122] was an early mechanism for controlling out-of-order instruction issue. Some superscalar microprocessors, such as the Motorola 88100, use a hardware structure similar to scoreboarding, yet issue instructions in program order.

Another architecture feature designed to support out-of-order instruction issue is *register renaming* [80]. In this scheme, additional registers that are not programmer visible are dynamically allocated for each new value generated. This enforces a *single-assignment* rule and removes anti- and output-dependences. The IBM RS 6000 superscalar processor implements a form of register renaming in its floating point unit. Six rename registers are used in addition to the 32 general purpose floating point registers.

The out-of-order processing of instructions in superscalar processors leads to difficult problems of saving a consistent state of an active process when a trap or interrupt occurs. Intricate hardware mechanisms have been devised to solve these problems, including the *history buffer* and the *reorder buffer* [105, 76].

2.4 Other Multiple-Issue Machines

In a *superpipelined architecture*, a deeper instruction pipeline is employed where the basic pipeline cycle time is a fraction of the base machine cycle time. During each machine cycle, several instructions can be issued into the pipeline. In a superpipelined machine of degree m, the instruction parallelism capable of being exploited is m. Superpipelined and superscalar machines are considered the dual of each other, although they may have different design tradeoffs [78].

Very long instruction word (VLIW) architectures exploit parallelism by packing several basic machine operations in a single instruction word [54]. These basic operations consist of simple load/store memory operations, and register-to-register ALU operations. Each operation is simple in that it can be issued in a single clock cycle, but may take several cycles to complete. VLIW architectures make extensive use of compiler techniques to detect parallelism and package them into long instruction words. Examples of VLIW architectures include Trace family machines [53, 52], the Cydra-5 [108], and iWARP [7].

2.5 Issues and Challenges

In 1983, Arvind and Iannucci set forth "two fundamental issues in multiprocessing." They are [10]:

- *memory latency:* "Most von Neumann processors are likely to "idle" during long memory references, and such references are unavoidable in parallel machines."

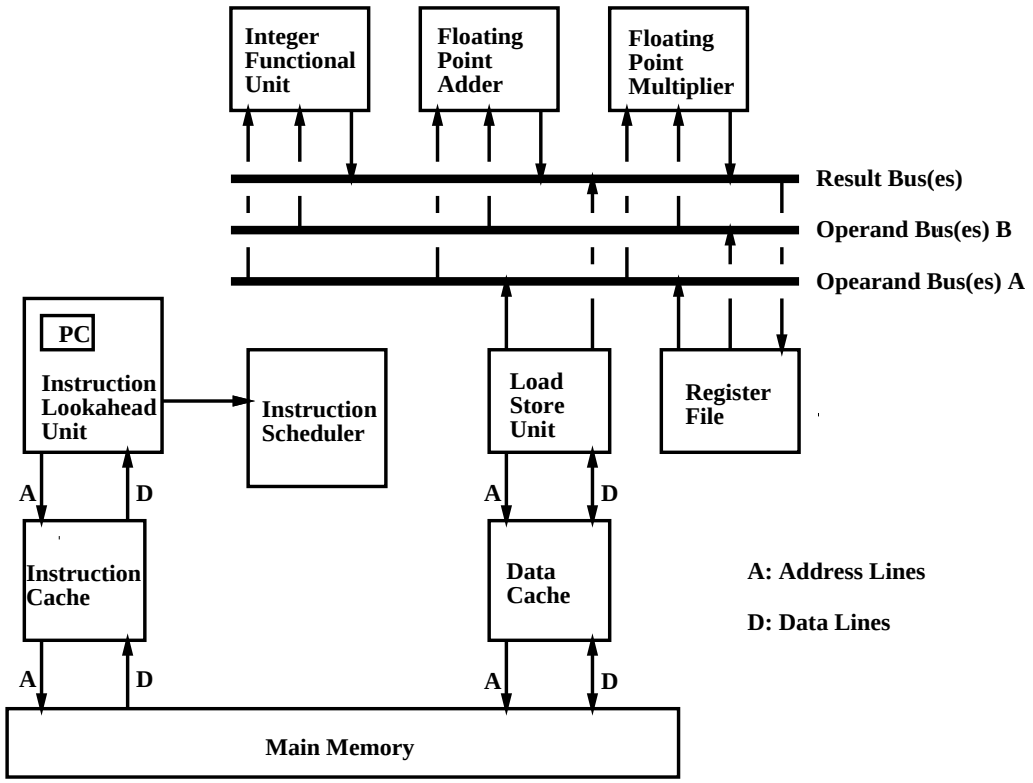

Figure 3: Organization of a superscalar processor.

In a superscalar architecture, enough hardware resources are provided to support the issuing of multiple instructions at a time. However, these parallel resources can be fully utilized only if sufficient parallelism is available in the instruction stream and the instruction execution mechanism is flexible enough to exploit it. Again, the control mechanism must recognize opportunities for multiple instruction execution while maintaining correct program behavior.

Using our generic RISC processor architecture as a reference, we use the term *instruction issue* to refer to the moving of an instruction from the instruction fetch stage to the first execution stage. The execution control mechanism of a superscalar processor examines instructions in their program sequence and issues an instruction for execution only if the resources required by the instruction (functional unit of the right type, operand registers filled, and result register free) are available. A processor employing *out-of-order issue* may begin execution of an instruction even though not all earlier instructions in the program sequence have begun execution. A significant parameter of a superscalar architecture is how far ahead in the program sequence the processor may reach to find issueable instructions. The greater this *look-ahead* capability [80], the greater the ability of the processor to exploit instruction level parallelism.

Superscalar processors employing out-of-order instruction issue require sophisticated mechanisms to deal with the reuse of registers that occurs in machine programs. Both output de-

see further reduction of clock cycle time. The question is: As more chip area becomes available, what architecture will utilize it most effectively? It seems that three avenues of development are possible:

- Increase the performance of a conventional (RISC) processor by increasing its parallelism and complexity. Several architectures—superscalar, superpipeline, and VLIW—follow this choice. Each of these approaches employs multiple-instruction-issue while retaining the single thread execution model. Machines in this class are called *multiple-issue machines*.
- Make a fundamental break with the single-threaded execution model so the processor architecture can usefully exploit the additional chip area. Multithreaded architectures follow this choice.
- Place several processors on one chip. Putting many processors on one chip is an approach that can be applied for any processor architecture, including RISC, superscalar, and multithreaded.

Superscalar, superpipelined and VLIW architectures are all motivated by the observation that lowering the CPI of the CPU has been the major factor driving the evolution of RISC architecture during the 1980s. The next appealing objective is to break the one cycle per instruction mark set by conventional RISC architecture by going for multiple instruction issue per cycle. Here we focus on superscalar architecture as representative of multiple-issue machines, and finish with some remarks about the other forms of multiple-issue architecture.

2.3 Superscalar Architectures

A computer based on superscalar architecture is capable of issuing multiple instructions from a single instruction sequence. These instructions may be processed by different functional units of the machine simultaneously. Examples of superscalar machines include the IBM RISC-6000 architecture [15], the Intel i860 and i960 microprocessors [87], the Motorola 88000 microprocessor system [93], and its successor, the Motorola 88110 [49].

Figure 3 illustrates the structure of a superscalar processor which may be viewed as an evolution from the RISC architecture of Figure 1. The principal differences are use of multiple functional units including separate functional units for floating point addition and floating point multiplication, and more and larger data buses to handle the higher data rates. In addition, a Load/Store Unit controls the execution of multiple, concurrent memory operations. The Instruction Scheduler has the job of assigning instructions to the various execution units so as to maximize performance without violating program dependences. This structure characterizes the Motorola 88110 microprocessor; some other superscalar processors such as the Intel i860 use a separate register set and buses for floating point computation.

In the case of a memory read that incurs a cache miss, or a floating point operation that takes more than one machine cycle, the time saved by operand forwarding is not enough to avoid pipeline gaps if successive instructions have pairwise data dependences. For example, suppose instruction $i2$ in Figure 2 is a load operation. The result from cache memory is not available until the end of the corresponding MA cycle (cycle #4). Therefore, instruction $i4$, if it uses this result via forwarding, could not have been issued earlier. Instruction $i3$ occupies a *delay slot* and will be a "no operation" instruction unless the slot can be filled with some operation that may be performed concurrently with instructions $i2$ and $i4$. A similar situation also occurs if an ALU instruction has a latency greater than one.

Therefore, achieving the best possible performance of a RISC processor depends on the ability of a compiler to perform *instruction scheduling*, the reordering of instructions so that delay slots are filled with useful instructions [20, 19, 112]. Statistics have shown that RISC compilers have been quite successful in finding instructions to fill in delay slots of one cycle. When delay slots longer than one cycle need to be filled, the problem of instruction scheduling becomes considerably harder.[1] In practice, measurements have suggested that the second delay slot can be filled less than 30 percent of the time, while the third delay slot can be filled in fewer than 10 percent of the cases [66].

Conditional instructions may also interfere with the smooth flow of instructions through the execution pipeline because instructions immediately after a branch in the program sequence may enter the pipeline before the test outcome of the branch is known. Methods of improving performance in the presence of branch instructions include a policy to always execute the instruction immediately after a branch unconditionally. A variety of more elaborate techniques have been proposed [66].

Keeping a high hit ratio in the data cache is not trivial, especially for computations involving large data structures such as arrays. A cache miss will cause the instruction pipeline to stall for several cycles, and the penalty will be severe if the miss ratio is high. Because the cache access time is often the dominating factor in determining processor cycle time, much effort has been invested in the design of cache memories to achieve high performance through compromise in choices of architecture parameters including total cache size, associativity, and block size [113, 69, 106, 66]. The challenge facing architects is how to make the cache both faster and larger, goals generally in conflict. Although the current trend is to devote a substantial proportion of the chip area of a RISC microprocessor to cache memory, other architectural choices may prove more cost-effective.

2.2 Supporting more Instruction-level Parallelism

The technology advance we have seen in the 1990's is likely to continue through the remainder of this decade. Thus can expect to see further increases in the level of device integration and to

[1] The problem is conjectured to be NP hard in general [20, 19].

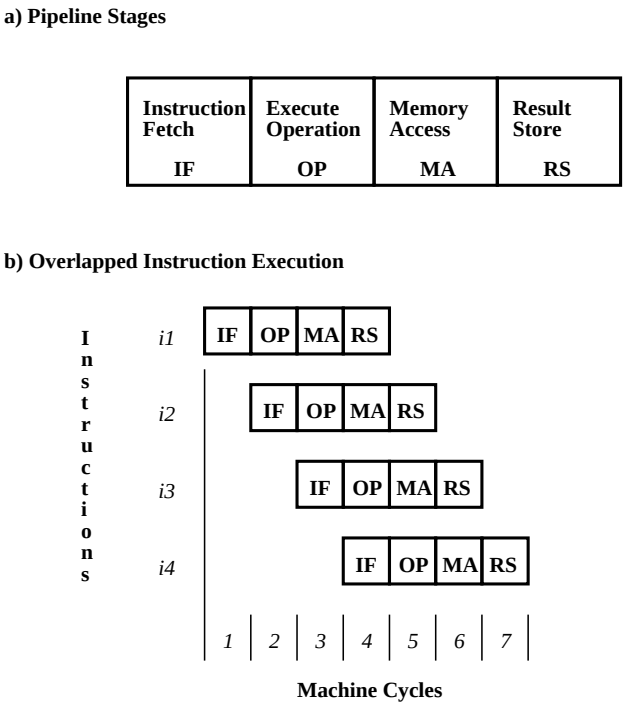

Figure 2: Pipelined instruction execution in a RISC processor.

if the instructions were executed without overlap. The goal is to achieve as much overlap as possible without violating the constraints imposed by dependences in the instruction sequence. A *data dependence* is present when one instruction reads a value put in a register by another instruction. In the absence of any control of instruction execution based on data dependences, overlapped execution may lead to reversal of access order and incorrect program execution. Early high performance computers such as Stretch and the CDC 6600 used hardware mechanisms to enforce correct sequencing of concurrent instruction execution. In many RISC architectures no hardware interlocks are provided and the compiler is expected to statically schedule instruction execution so that an access order is ensured that realizes correct program execution.

To illustrate, suppose two instructions $i1$ and $i2$ in Figure 2 have a data dependence because $i2$ uses a register operand value which is the result produced by $i1$. In most RISC architectures it is safe to allow instruction $i2$ to be issued in the cycle immediately after i1 is issued, assuming that $i1$ calls for an ALU operation with an execution latency of one cycle. This may seem odd because, although the result of $i1$ is available at the output of the ALU at the end of cycle #2, it cannot be obtained from a register before completion of cycle #4. The hardware technique known as *bypass* or *forwarding* recognizes this kind of situation and can route the result of the scalar operation performed by $i1$ directly to the appropriate operand input of the functional unit for $i2$. In this way, the delay of a register write and read is avoided.

feature is the two cache memories located in the retrieval paths for instructions and data. Cache memories are provided on the processor chip to support the increased instruction rates of modern RISC designs. Each cache memory holds lines of information, where a line often contains two or more words. Each cache line is associated with a set of main memory locations, and, at any time, contains the group of words at one such location. The cache hardware operates so that each line holds the most recently accessed location of its set. The Instruction Cache may be made read-only, which reduces its complexity because accesses never require updating main memory locations. If most memory accesses are *hits*, and find their targets in cache memory, then the bandwidth that must be handled by the off-chip data paths to main memory is greatly reduced. A second effect of cache memory is to reduce the normal access time for instructions or data to the access time of the cache memory, and this often determines the clock period of the processor.

The other major feature of modern RISC processors is their use of pipelined instruction execution to achieve an average CPI close to one. Figure 1 shows a data path structure that might be used for a RISC processor with a four-stage pipeline. The Instruction Fetch unit forms the first stage. In the absence of Instruction Cache misses and branch instructions, it obtains an instruction from the cache on each machine cycle. Operation of the remaining stages is handled by the Pipeline Controller. For each stage we indicate what happens (1) for register-to-register scalar operations, and (2) for memory read and write instructions (loads and stores):

- **Instruction Fetch:** Use the instruction address from the program counter to access the Instruction Cache.
- **Execute Operation:** 1. Fetch operands from registers for register-to-register scalar operations. 2. Fetch indexing or offset values for computing the effective address of a load or store instruction. Get data for a store instruction from its source register.
- **Memory Access:** 1. Complete register-to-register scalar operations if more than one machine cycle is required. 2. Access the Data Cache for load and store instructions.
- **Result Store:** Put the result of a scalar operation or the word fetched by a load instruction into the destination register.

The data path shown in Figure 1 is organized so the work of each stage can be done in one machine cycle, even as other stages are operating for other instructions. For example, the Register File has two read ports and one write port so both operands can be read for a scalar operation while the result of an earlier instruction is written to its destination register. Figure 2 shows how successive instructions flow through the pipeline stages.

The pipeline of a RISC processor must be managed to implement correct execution of the sequence of instructions presented to it. This usually means that the pipelined (overlapped) execution of the sequence must have the same net effect on memory and register contents as

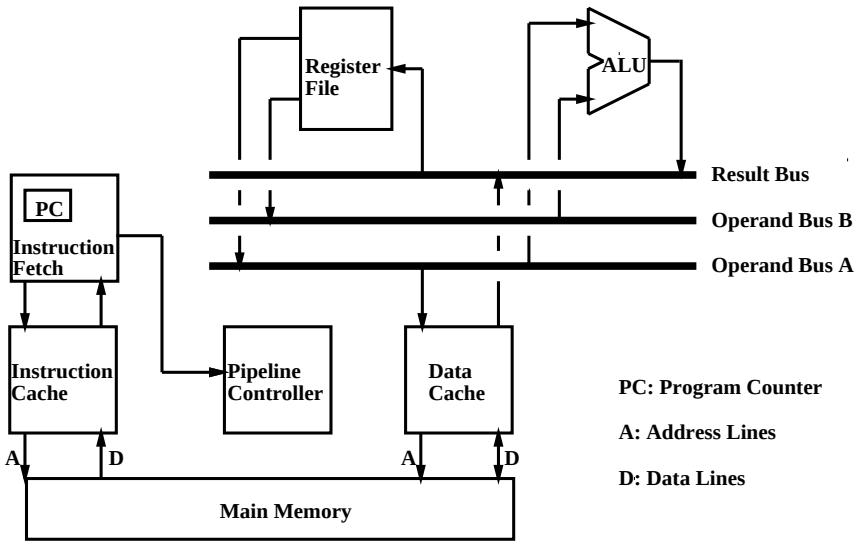

Figure 1: Data path organization of a simple RISC processor.

The RISC idea came originally from several independent efforts. On one hand Patterson used a simplified instruction set so that a complete CPU design could be implemented on a single VLSI chip [104]. On the other hand, Cocke and associates in the IBM 801 Project explored a simplified instruction set combined with compiler code-generation strategies to obtain improved cost/performance [107]. Thus RISC concepts emerged from the desire to pack the greatest functionality into a chip and the idea of making an astute division of function between hardware and software.

As semiconductor technology advanced, yielding chips with higher component densities, the goal of RISC architecture has been to develop processor designs that can come close to initiating one instruction on each clock cycle of the machine. This measure, *clock cycles per instruction* (CPI), is commonly used to characterize architectures for high performance processors. The goal of building RISC chips with a CPI of one has been met by several products including the MIPS chip, the Sparc chip, and others. Two architectural features have made this achievement possible: *instruction pipelining* and *cache memories*. Pipelined instruction execution [86] allows subsequent instructions to begin execution before previously issued instructions have finished execution. This inherently involves exploiting instruction-level parallelism. The cache memory [113] allows instruction execution to continue, in most cases, without waiting the full access time of the main memory.

A major feature of RISC instruction set architecture is that the basic memory operations, load and store, are provided in instructions separate from those supporting the basic scalar operations, which always operate directly on the contents of fast machine registers.

Figure 1 shows the data path structure of a simple RISC microprocessor. A salient

We begin in Section 2 with a review of the principles of RISC architecture and an assessment of the challenges posed by the evolution of technology. Several proposed directions of architectural development are outlined, using the superscalar approach as an example. In Section 3, we introduce basic concepts and characteristics of multithreaded computer architectures. Sections 4, 5, and 6 present three multiprocessor designs that illustrate the variety of architectures that embody principles of multithreading. The three designs are the Heterogeneous Element Processor (HEP) System, a modern static dataflow architecture project, and the MIT Monsoon multiprocessor. These case studies are followed in Section 7 by a survey of other significant projects in multithreaded computer architecture, including some based on dataflow ideas, and others that have evolved from conventional multiprocessor designs.

Design issues for multithreaded processing elements suitable for use in parallel computers for scientific computing are the subject of Section 8. These issues include the choice of an appropriate program execution model, the organization of the processing element to achieve good utilization of resources, support for fine-grain interprocessor communication and global memory access, compiling machine code for multithreaded processors, and the challenge of implementing virtual memory on large-scale multiprocessor systems. The chapter concludes by inquiring how future architectures of high performance systems will respond to forthcoming possibilities of technology and the increasing demand for attention to issues of programmability.

2 Microprocessor Evolution: Principles and Challenges

The use of microprocessors has led to the phenomenal growth of personal computers and workstations, and the servers that can link them into powerful distributed computing systems for engineering and commerce. There is a continuing effort to exploit the further advance of VLSI technology. Each year designers are able to put more transistors on a chip and to increase the clock speed of on-chip operations. Indeed, the technology has evolved to the point that essentially all principal architectural features of conventional uniprocessor computers have been incorporated into microprocessor designs. A major step in this evolution has been the introduction of *reduced instruction set computer* (RISC) architecture. In this section, we review the principles and problems of RISC architecture, discuss their on-going evolution into multiple-instruction-issue architectures, and give an assessment of the challenges facing the killer micros as processing elements in massively parallel computers.

2.1 Reduced Instruction Set Computer (RISC) Architecture

The evolution of reduced instruction set computer (RISC) concepts and principles has been driven by the workstation market, where the demand is for the highest instruction processing rate within the constraints of single-chip design for workloads of business, engineering, and graphics applications running in a window-interfaced operating environment. Floating point performance is at best a secondary consideration in this domain.

1 Introduction

The contemporary computing scene includes two expanding worlds of computer systems: massively parallel computers in high performance scientific computation, and client/server networks of workstations in engineering and business applications. Just as client/server systems appear destined to take over much of general purpose computing from conventional "mainframe" computers, massively parallel computers are making strong inroads into territory dominated by conventional (vector) supercomputers.

The United States Government has set forth several "Grand Challenge Problems" that require enormous computational capacity to solve, and has made the solution of these problems a national goal [25]. These problems all present opportunities for large-scale parallel computing, and it is now generally appreciated that multiprocessor architectures provide the only promising approach to meeting the computational requirements of the Grand Challenge Problems.

The VLSI microprocessor lies at the heart of these developments. It is the engine of workstations and servers in distributed systems, and it has become the key component of today's massively parallel computers for high performance scientific applications. These are the "killer micros" foreseen by Eugene Brooks [26]. The economics of computer technology ensures that the processor-on-a-chip will continue to be the cost-effective basis for high performance computer design. Soon it will be possible to place several complete processing elements on a single chip having several million transistors.

Today the most common architecture for high performance, single-chip microprocessors is RISC (for Reduced Instruction Set Computer). These devices have been successful in workstations, and in have been chosen as the main processors for massively parallel computers such as the Intel Paragon and the Thinking Machines CM-5. Nevertheless, RISC microprocessors have significant limitations as processing elements of massively parallel computers. The design of current generation RISC microprocessors is aimed primarily at the workstation market, and their support for efficient interprocessor communication and synchronization is weak, in view of the requirements of massively parallel scientific computation for high bandwidth, low latency communication. In our view, this weakness is not likely to be overcome unless some fundamental problems of multiprocessing are addressed.

RISC microprocessors now dominate the CPU market for high-performance workstations. The applications which run on these workstations range from scientific and engineering applications, business transaction processing, symbolic computation, to various tasks of personal computing. However, the requirements for massively parallel computing are very different from those driving the workstation industry. The advent of massively parallel scientific computation has introduced a new domain of computing hardware, program models, and compiling approaches that are significantly divergent from the domain of current workstations. Hennessy and Jouppi [65] and Stone and Cocke [116] have projected that the near term course of evolution of microprocessor designs will be in the direction of superscalar and superpipelined architectures. Here we examine an alternative: multithreaded processing element architecture.

List of Figures

1 Data path organization of a simple RISC processor. 3

2 Pipelined instruction execution in a RISC processor. 5

3 Organization of a superscalar processor. 8

4 Structure of a HEP multiprocessor system. 14

5 The HEP Processing Element Module. 15

6 The HEP Instruction Scheduler. 16

7 A signal graph for the DAXPY routine. 19

8 A multithreaded dataflow processing element. 20

9 The DAXPY routine programmed using two threads. 21

10 The instruction execution pipeline. 22

11 Organization of a Monsoon system. 24

12 The Monsoon execution pipeline. 26

List of Tables

1 Examples of global memory implementations. 41

8 Issues in Multithreaded Architecture **33**

8.1 Program Execution Models 33

8.2 Processor Design to Support Interleaved Threads 36

8.3 Processor Design for Fast Context Switching 37

8.4 Synchronization and Scheduling of Threads 39

8.5 Multiprocessor Issues 40

8.6 Compiling Programs for Multithreaded Architectures 42

8.7 Virtual Memory for Multiprocessors 44

9 Conclusions **46**

10 Acknowledgements **46**

Contents

1 Introduction **1**

2 Microprocessor Evolution: Principles and Challenges **2**

2.1 Reduced Instruction Set Computer (RISC) Architecture 2

2.2 Supporting more Instruction-level Parallelism 6

2.3 Superscalar Architectures 7

2.4 Other Multiple-Issue Machines 9

2.5 Issues and Challenges 9

3 Multithreaded Program Execution Models **10**

3.1 The Single-Threaded Architecture Model 11

3.2 Multithreaded Architecture Models 12

3.3 A Brief Historical Note 13

4 HEP: The Heterogeneous Element Processor System **13**

5 A Dataflow Architecture **18**

6 Monsoon **23**

7 Other Multithreaded Architecture Projects **27**

7.1 Multithreaded Architectures Based on Dataflow Models 28

7.2 Architectures not Derived from Dataflow Models 29

Abstract

The architecture of future high performance computer systems will respond to the possibilities offered by technology and to the increasing demand for attention to issues of programmability. Multithreaded processing element architectures are a promising alternative to RISC architecture and its multiple-instruction-issue extensions such as VLIW, superscalar, and superpipelined architectures.

This paper presents an overview of multithreaded computer architectures and the technical issues affecting their prospective evolution. We introduce the basic concepts of multithreaded computer architecture and describe several architectures representative of the design space for multithreaded, parallel computers. We review design issues for multithreaded processing elements intended for use as the node processor of parallel computers for scientific computing. These include the question of choosing an appropriate program execution model, the organization of the processing element to achieve good utilization of major resources, support for fine-grain interprocessor communication and global memory access, compiling machine code for multithreaded processors, and the challenge of implementing virtual memory in large-scale multiprocessor systems.

McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Multithreaded Architectures: Principles, Projects and Issues

Jack B. Dennis†
Guang R. Gao

ACAPS Technical Memo 29
February 4, 1994

Also appears in Chapter 1 of "Multithreading: A Summary of The State of The Art,"
edited by R. Iannucci, G. R. Gao, R. Halstead and B. Smith,
Kluwer Academic Publishers, 1994

†MIT Laboratory for Computer Science, 545 Technology Sq., Cambridge, Massachusetts, USA