the mask being scanned across the image, in this case the opposite is occurring—the template is hard-wired into the FPGA while the image pixels are clocked past it.

Another important opportunity for increased efficiency lies in the potential to combine multiple templates on a single FPGA. The simplest way to do this is to spatially partition the FPGA into several smaller blocks, each of which handles the logic for a single template. Alternatively, one can seek to identify templates having some topological commonality, and which can therefore share parts of adder trees. This is illustrated in Fig. 11, which shows two templates that share several pixels in common, and which can be mapped using a set of adder trees that leverage this overlap. The advantage of using FPGAs is that FPGAs can be dynamically optimized at the gate level to exploit template characteristics. A general-purpose correlator would have to provide large general-purpose adder trees to handle the summing of all possible template bits. The FPGA, however, exploits the sparse nature of the templates, and only constructs the small adder trees required. FPGAs can exploit other factors such as collapsing adder trees with common ele-
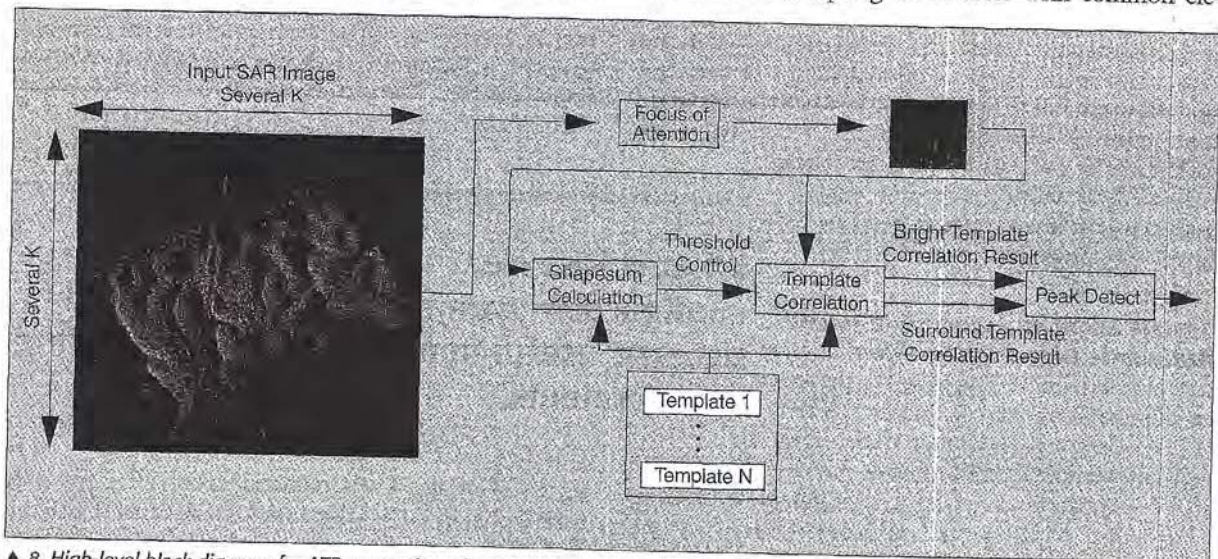

▲ 8. High-level block diagram for ATR processing. The focus of attention algorithm identifies regions of interest in SAR images. Pixels in regions of interest are correlated against a series of binary target template pairs, with each pair containing a bright template (identifying pixels of strong expected radar return) and a surround template (strong radar absorption). Templates with the highest correlation are selected in the peak detection step.
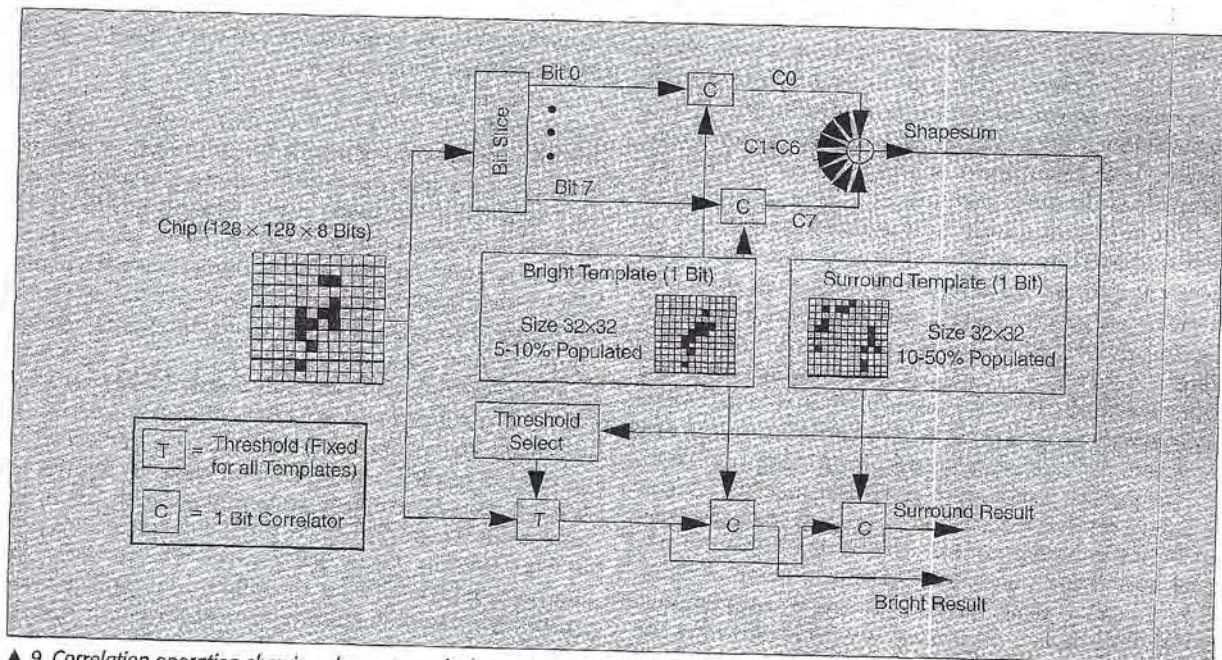
▲ 9. Correlation operation showing shapesum calculation (top) and thresholding/correlation (bottom).
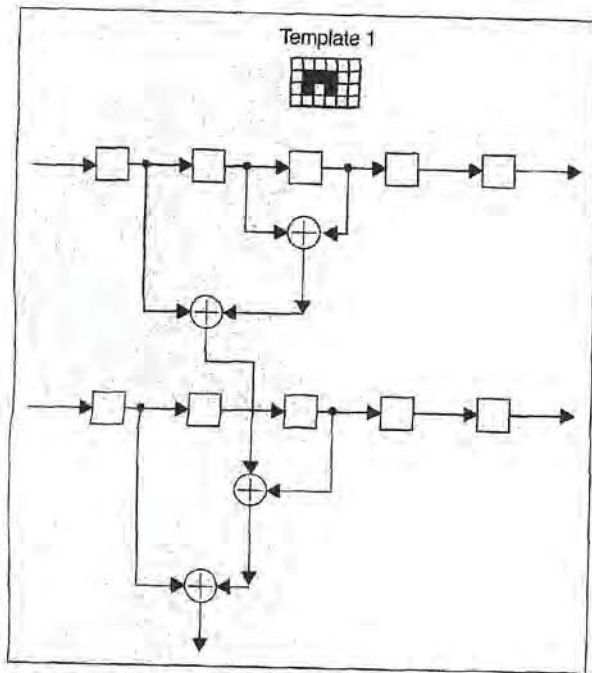
▲ 10. Example binary template with five "on" pixels (top) and corresponding adder tree (bottom).
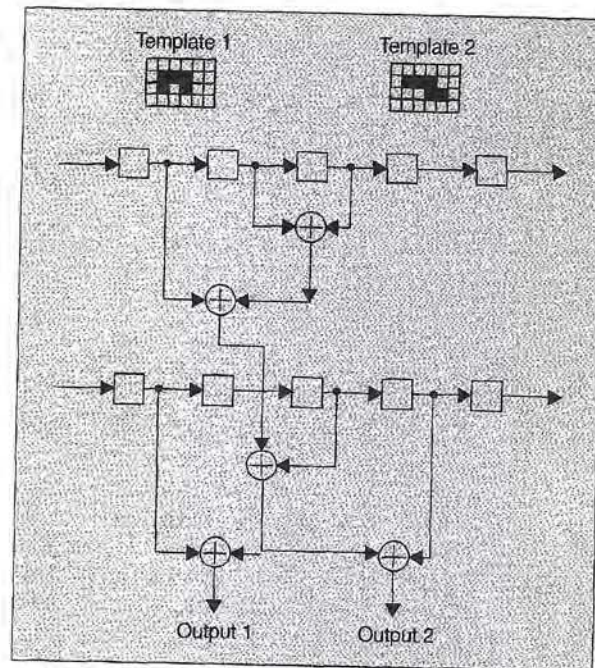
▲ 11. Template commonalities are exploited to reduce hardware requirements for computing multiple correlations.

ments, and storing pixels that are not needed by the adder trees using RAM-based shift registers.

Table 2 illustrates the FPGA resource trade-offs involved in template mapping. The table gives the FPGA utilization for the Xilinx 4062 when four through seven template pairs are simultaneously mapped into the FPGA using the approach described above. Each template pair consists of two 32×32 binary images and is represented in the hardware using two template-specific adder trees. The number of templates per second that can be evaluated using this approach is a function of many factors including the clock rate, the FPGA configuration time, the number of templates per configuration, the candidate image and target sizes, the number of clock cycles needed to evaluate the templates at each relative image/template offset, and on I/O considerations. The performance can be upper bounded by assuming that the I/O is fully efficient; i.e., that the FPGA is always either computing correlations or being reconfigured. Assuming efficient I/O is fairly reasonable in the prototype systems we have constructed, we have been able to avoid letting the FPGA be idle by using scaled down versions of the templates. When all of these factors are considered together, we find that configuration can consume more time than computation; i.e. there is a significant performance penalty due to reconfiguration. This overhead will diminish to 10% or less when partially reconfigurable FPGAs become more widely available. However, for parts that are not partially reconfigurable, the benefits of increased computation power offered by larger FPGAs are to some extent mitigated by the larger configuration bitstreams and longer reconfiguration times that these parts require.

Figure 12 shows a configurable computing board that was constructed at UCLA as a prototype for the template-matching problem. The board contains a "dynamic" FPGA that is used for template correlations and is run-time reconfigured, a "static" FPGA for control, SRAM for storage of pixels and results, EPROM for configuration bitstream storage, and an interface to an i960 embedded processor for more advanced configuration control.

Ongoing Research

Configurable computing has grown from a field with a handful of researchers in 1989 [45] to one that now receives the attention of hundreds of researchers and engineers in academia, industry, defense, and a rapidly increasing number of start-up companies. In this section we identify some of the open issues in this field and describe selected recent and ongoing projects that aim to address them.

One of the most interesting questions in configurable computing concerns the extent to which current FPGA device and machine architectures should be altered to better support computing as opposed to the prototyping that drove much of the early evolution of FPGAs. Academic researchers pursuing this question face the obvious challenge of being unable to fully exploit the existing infrastructure of commercial FPGAs and design tools, and typically design custom FPGAs to validate their architecture proposals. Various projects are underway, each attacking one or more of the well-known weaknesses of commercial FPGAs. For example, some researchers are

investigating architectures that are based on relatively wide 16-bit datapaths (as opposed to the 1-bit datapaths found in today's FPGAs). While less flexible than FPGAs, these computing devices are much more efficient in silicon terms and achieve higher arithmetic performance on 16-bit integer data. Other researchers are investigating novel configuration approaches that either reduce configuration time through context-switching or that distribute configuration data with data to be processed. Still other researchers are merging general-purpose processors and FPGA resources on the same die in an attempt to combine the best features of both technologies.

Peter Athanas' group at Virginia Tech is exploring 16-bit computing devices based on the "wormhole" technique: a computing approach that distributes configuration data with the data to be processed [33]. Consisting of a single multiplier and a 4 × 4 array of 16-bit arithmetic logic units (ALUs) interconnected by a crossbar, their COLT device combines configuration data and data into a single packet. Resembling dataflow computing in many aspects, configuration data in the packet are used to route data through the array and to configure ALUs for subsequent processing. COLT has been fabricated and is currently being tested.

Carl Ebeling's group at the University of Washington is working on RAPID, another device based on 16-bit datapaths [14]. A RAPID array consists of a mostly linear array of RAPID cells, each cell consisting of an integer multiplier, three integer ALUs, six registers, and three small memories. RAPID is primarily statically configured but uses limited dynamic control to provide run-time flexibility.

Matrix, developed by Andre DeHon and others at MIT, is based on a cell that can serve as an instruction store, a memory element, or a computational element. All datapaths are 8-bit and these cells are interconnected with multilevel interconnect that can be used both for data and instruction distribution. Matrix is currently undergoing commercial development by a new startup company, Silicon Spice. DeHon has also conducted an in-depth study that sets FPGA-based computing in a general-purpose computing context and has suggested several machine architectures [12].

FPGA vendors are also pursuing their own research and development projects as well as more aggressive fabrication processes. For example, over the next two years, devices using supply voltages of 2.5 Volts and below will become common. In addition, the technology lag of FPGAs with respect to ASICS in terms of feature size and number of metal layers is rapidly shrinking, with 3-5-layer FPGAs fabricated using sub .3 micron technology expected to become common. The vendors are also likely to both introduce and adopt architectural innovations that have shown promise in academic research. Since future FPGAs will track ASIC technology more closely and will benefit from a richer set of architectural features, they are likely to compare more favorably with ASICs for many applications than that of today.

The BRASS project at U.C. Berkeley under John Wawrzynek [22] is developing a single chip (Garp) that incorporates a MIPS-II processor and an FPGA core whose elements roughly correspond to those found in the Xilinx 4000 series. The BRASS researchers have modified the MIPS-II processor, replacing the floating-point unit with an FPGA core of their own design, and have augmented the instruction set to include operations that manage the FPGA resources. Their goal is to execute data-intensive operations on the FPGA core and leave general-purpose operations on the processor. A related effort at National Semiconductor Corporation is building an FPGA that will combine a programmable processor and a fine-grained FPGA on the same chip [18].

Other researchers are investigating solutions to mixing configurable computing elements with more traditional processors. For example, Jan Rabaey of U.C. Berkeley has examined the allocation of tasks in typical digital signal processing and has proposed a multigranularity architecture that allows computations to be directed to the hardware that best supports them [34]. Rabaey is also investigating strategies for low-power FPGAs. Though some power reduction will occur automatically due to technology changes, there is substantial opportunity to

redesign the logical units in FPGAs with power as a principal constraint.

Several groups are looking at FPGAs that have multiple configurations, or contexts, stored on-chip simultaneously. At any given time one context is active and the others are stored in lower planes. Contexts can be swapped extremely quickly—requiring from one to several hundred clock cycles to complete—potentially eliminating much of the overhead involved in loading configuration bitstreams from off-chip. Of course, context switching involves other overheads such as the resources needed to hold multiple contexts on-chip, and the hardware and tools to manage context-switching. The earliest work on context-switched FPGAs was done at Xilinx beginning in 1991, though it remained proprietary until very recently [42]. In the academic community context switching was studied by Tom Knight, DeHon, and their colleagues at MIT [11, 41].

Work to develop new configurable computing devices also benefits from an understanding of how algorithms map into the range of architectures represented by today's FPGAs and FPGA systems. Some of the most extensive algorithm mapping work has been performed by the BYU group led by Brad Hutchings, which has experimented with most commercially available (and noncommercially available) FPGAs as well as prototype systems such as the HP Teramac [2] and Splash-2 [4]. BYU has demonstrated applications in the following areas: neural networks [15], morphology [48], ATR [35] and genetic algorithms [19]. BYU has also developed a variety of design and implementation strategies [25] and provides tutorials for many different FPGA platforms via their web site: http://splish.ee.byu.edu. A large bibliography of related papers is also available at this site.

BYU's early research agenda was twofold: one, determine what characteristics make an application a good candidate for implementation on an FPGA-based computing platform, and two, research and understand the strengths and weaknesses of current devices, system organizations, and tools. Following up on this basic research, BYU is now in the process of developing new system organizations and application-development strategies that are based upon high-performance circuit libraries, domain-specific compilation, and RTR. BYU also continues to experiment with applications in an effort to find additional applications that can exploit this technology.

John Villasenor and his colleagues at UCLA have demonstrated a video communications system in which a single 5000-gate FPGA was reconfigured four times per image frame to allow compression and transmission of an image [26]. The Mojave project at UCLA, led by John Villasenor and Bill Mangione-Smith, has resulted in several generations of boards and domain-specific design libraries for the ATR application described previously. These boards included an interface to an embedded processor that performed on-the-fly analysis of results and modified the FPGA configuration sequence accordingly [43, 44].

Researchers including Mohammad Shajaan and John Sorensen of the Technical University of Denmark [38] have examined architectures for performing digital filtering using FPGAs. Because today's FPGAs perform multiplications poorly, much of the attention in filtering using FPGAs has focused on multiply-free implementations. In the future, it is also likely that adaptive filtering algorithms will find application in FPGAs that are partially reconfigured as the filter coefficients evolve.

Another area of research focus is in compilers and tools for configurable computing platforms. Ian Page of Oxford University has developed Handel, a programming language that allows programmers to simultaneously develop the FPGA circuit descriptions and processor software with a single description language based on OCCAM [31]. Reiner Hartenstein of the University of
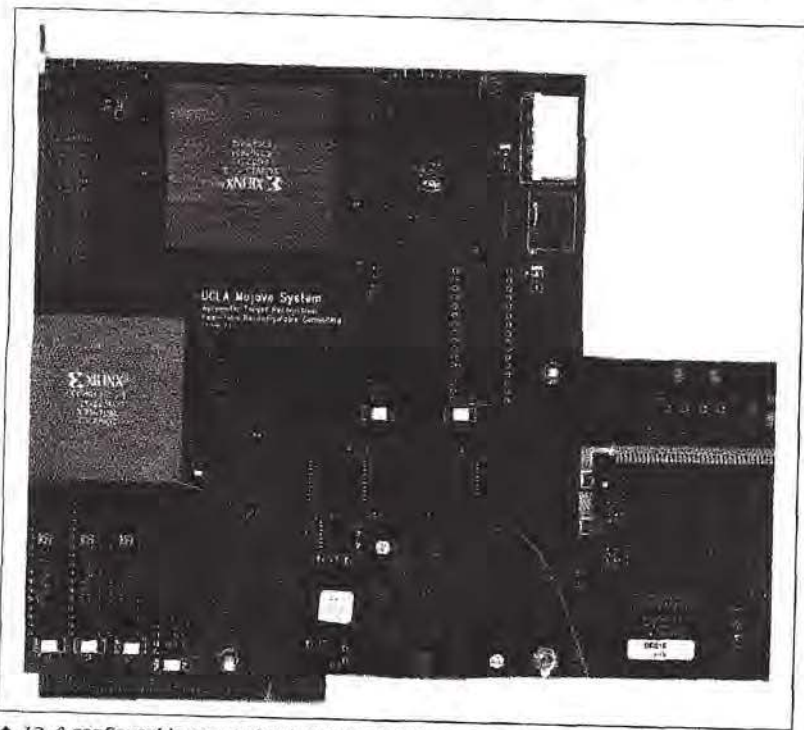
▲ 12. A configurable computing board for ATR built at UCLA. The board includes a "dynamic" FPGA that implements template correlations and is rapidly reconfigured during execution, a "static" FPGA for control, SRAM for image data storage, and an EPROM for configuration bitstream storage. The board resides in a host PC and receives images across a PCI bus.

Kaiserslautern has developed a machine-level abstraction called the Xputer [21] that also derives the target machine description and its program from the same description. Wayne Luk at Imperial College is investigating formal approaches to FPGA design based on the language RUBY [29]. Transmogrifier-C [17], developed by researchers at the University of Toronto, is a programming approach targeted at Toronto's TM-2 custom platform, which is currently under development [27]. Anant Agarwal and his colleagues at MIT are working on automated programming approaches for very large configurable-computing platforms [6]. In addition, HP developed a very easy-to-use compiler for their Teramac system that automatically partitioned, placed, and routed a netlist of 1-million gates into the nearly 1000 custom FPGAs that formed Teramac [2].

Configurable computing is represented by a growing presence in the commercial world. In addition to FPGA vendors including Xilinx and Altera there is a rapidly growing list of start-up companies with products that are based on configurable computing. These including Annapolis Microsystems of Annapolis Maryland, which commercialized the SPLASH-II architecture; Virtual Computing Corporation of Reseda, California; Morphologic of Nashua, New Hampshire; and Giga Operations of Berkeley.

Conclusions and Future Directions

It is now clear that for applications requiring deeply pipelined, highly parallel, bit-level operations including cryptography, target recognition, and some types of image processing, configurable computing machines offer compelling speed and cost advantages over alternative implementations. For these types of applications, configurable computing machines are likely to become solutions of choice. What is less clear is the extent to which configurable computing techniques will become useful in more general computing environments, in particular for applications that involve high arithmetic complexity. Given the dominance and ever-increasing capabilities of microprocessors for general-purpose computing, it seems highly unlikely that any other computing model, including that offered by configurable computing, will make significant inroads against microprocessors in the foreseeable future. Widespread adoption of configurable computing is also hampered by the lack of exactly what microprocessors possess in abundance: a set of relatively easy to use, widely known software programming languages and associated compilers or interpreters that allow a user with little or no knowledge of the underlying hardware to instruct a computing platform to perform a desired task.

The lack of a sufficiently general high-level software programming model is of course a well-known problem among researchers performing work in configurable computing, and there are many ongoing efforts in which creation of a design tool infrastructure is a goal. Even if such languages can be developed, tested, and adopted, there remains the problem of the "compiler," which in the domain of FPGAs means the tool chain that translates a functional or structural description of the task into a con-

---

Configurable computing is likely to benefit from architectural innovations both in FPGAs and in the hardware to interface to them.

---

figuration bitstream that fully describes the circuit in the FPGA. FPGA place and route tools have always benefited from place and route techniques used in ASIC design, which involves many of the same challenges and tradeoffs in terms of clock speed, design complexity, etc. However, the several hours needed by current-generation commercial FPGA tools to synthesize, place, and route a design on an FPGA, while fast when viewed in the context of ASIC design, are unacceptably slow when compared to software compilers. To make configurable computing practical will require that FPGA place and route tools be made faster by several orders of magnitude, most likely at the cost of highly suboptimal mappings of tasks into hardware. One exciting, but as yet unproven, approach that has been advocated by William Mangione-Smith of UCLA is dynamic compilation, in which small units of precompiled FPGA configuration bitstreams can be combined extremely quickly at run time to constitute a full FPGA configuration bitstream. There are many challenges in dynamic compilation, not the least of which is the proprietary nature of configuration bitstreams for most commercial FPGAs.

As configurable computing advances it is also important to distinguish techniques that are truly new, such as large-scale run-time hardware reconfiguration, from techniques that have existed in computing for many years. Many of the "new" approaches in configurable computing are in fact existing computing concepts that are being implemented in a new domain. For example, the ATR algorithm described previously gains its efficiency from RTR, which can legitimately be claimed as an innovation due to configurable computing, and from mapping target templates into template-specific adder trees, which is an example of the years-old technique of partial evaluation.

In addition to the obvious trend toward larger devices, configurable computing is likely to benefit from architectural innovations both in FPGAs and in the hardware to interface to them. Configurable computing is a young field with enormous potential to grow as FPGAs, their derivatives, and the tools to use them advance. The FPGAs that will be emerging in the next few years will be in excess of half a million equivalent gates, which is large enough to support a very diverse range of applications. In addition, the state of the art in architectures for configurable computing devices will be significantly enriched by the many ongoing research efforts studying architecture issues. Existing and perhaps new vendors of configurable computing devices, who are now well aware of the potential of configurable computing, can be expected produce devices and the associated tools that will make FPGAs of today look primitive.

John Villasenor is a Professor at UCLA's Electrical Engineering Department in Los Angeles, California. *Brad Hutchings* is an Associate Professor at Brigham Young University's Electrical and Computer Engineering Department in Provo, Utah.

References

1. A.L. Abbott, P.M. Athanas, L. Chen, and R.L. Elliott, "Finding lines and building pyramids with Splash 2". In D.A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 155-161, Napa, CA, April 1994.

2. R. Amerson, R. Carter, B. Culbertson, P. Kuekes, and G. Snider, "Teramac - configurable custom computing." In D.A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 32—38, Napa, CA, April 1995.

3. J.M. Arnold, "The Splash 2 software environment." In D.A. Buell and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 88-93, Napa, CA, April 1993.

4. J.M. Arnold, D.A. Buell, and E.G. Davis, "Splash 2." In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 316-324, June 1992.

5. P.M. Athanas and A.L. Abbott, "Real-time image processing on a custom computing platform." *IEEE Computer*, 28(2):16-24, February 1995.

6. J. Babb, M. Frank, E. Waingold, and R. Baruna, "The RAW benchmark suite: Computation structures for general purpose computing." In J.M. Arnold and K.L. Pocek, editors, *Proceedings of IEEE Workshop on {FPGA}s for Custom Computing Machines*, Napa, CA, April 1997, to be published.

7. P. Bertin, D. Roncin, and J. Vuillemin, "Introduction to programmable active memories." In J. McCanny, J. McWhirther, and E. Swartslander Jr., editors, *Systolic Array Processors*, pp. 300-309. Prentice Hall, 1989.

8. G. Brebner, "The swappable logic unit: a paradigm for virtual hardware." In J.M. Arnold and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1997, to be published.

9. J. Burns, A. Donlin, J. Hogg, S. Singh, and M. de Wit, "A dynamic reconfiguration run-time system." In J.M. Arnold and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1997, to be published.

10. D.A. Clark and B.L. Hutchings, "Supporting FPGA microprocessors through retargetable software tools." In J. Arnold and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 195-203, Napa, CA, April 1996.

11. A. DeHon, "DPGA-coupled microprocessors: Commodity ICs for the early 21ˢᵗ century." In D.A. Buell and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 31-39, Napa, CA, April 1994.

12. A. DeHon, Reconfigurable Architectures for General-Purpose Computing. PhD thesis, Massachusetts Institute of Technology, September 1996.

13. T. Drayer, W. King, J. Tront, and R. Conners, "MORRPH: A Modular and reprogramable real-time processing hardware." In D.A. Buell and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 11-19, Napa, CA, April 1995.

14. C. Ebeling, D.C. Cronquist, and P. Franklin, "RaPiD - reconfigurable pipelined datapath." In J.M. Arnold and K.L. Pocek, editors, *International Workshop on Field-Programmable Logic, FPL'96*, pp. 126-135, Darmstadt, Germany, September 1996.

15. J.G. Eldredge and B.L. Hutchings, "Run-time reconfiguration: A method for enhancing the functional density of SRAM-based FPGAs." *Journal of VLSI Signal Processing*, vol. 12, pp. 67-86, 1996.

16. C.W. Fraser and D. Hanson, *A Retargetable C Compiler*. Benjamin/Cummings Company, 1995. ISBN 0-8053-1670-1.

17. D. Galloway, "The transmogrifier C hardware description language and compiler for FPGAs." In D.A. Buell and K.L. Pocek, editors, Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, pp. 136-144, Napa, CA, April 1995.

18. M. Gokhale and E. Gomersall, "High-level compilation for fine grained FPGAs". In J.M. Arnold and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1997, to be published.

19. P. Graham and B. Nelson, "A hardware genetic algorithm for the travelling salesman problem on SPLASH 2." In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, pp. 352-361, Springer, Oxford, England, August 1995.

20. P. Graham and B. Nelson, "Genetic algorithms in software and in hardware - A performance analysis of workstation and custom computing machine implementations." In J. Arnold and K. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 216-225, Napa, CA, April 1996.

21. R.W. Hartenstein, A.G. Hirschbiel, M. Riedmuller, K. Schmidt, and M. Weber, "A novel ASIC design approach based on a new machine paradigm." *IEEE Journal of Solid-State Circuits*, vol. 26, no. 7, pp. 975-989, July 1991.

22. J.R. Hauser and J. Wawrzynek, "Garp: A processor with a reconfigurable coprocessor." In J.M. Arnold and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1997, to be published.

23. Brian Von Herzen, "Signal processing at 250 MHz using high-performance FPGAs." In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 62-68, Monterey, CA, February 1997.

24. D.T. Hoang, "Searching genetic databases on Splash 2." In D. A. Buell and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 185-191, Napa, CA, April 1993.

25. B.L. Hutchings and M.J. Wirthlin, "Implementation approaches for reconfigurable logic applications." In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, pp. 419-428, Springer, Oxford, England, August 1995.

26. B. Schoner J. Villasenor and C. Jones, "Video communications using rapidly reconfigurable hardware." *IEEE Trans. on Circuits and Systems for Video Technology*, pp. 565-567, December 1997.

27. D.M. Lewis, D.R. Galloway, M. van Ierssel, J. Rose, and P. Chow, "The transmogrifier-2: A 1 million gate rapid prototyping system." In

ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 53-61, Monterey, CA, February 1997.

28. D.P. Lopresti, "Rapid implementation of a genetic sequence comparator using field-programmable gate arrays." In C. Sequin, editor, *Advanced Research in VLSI: Proceedings of the 1991 University of California/Santa Cruz Conference*, pp. 138-152, Santa Cruz, CA, March 1991.

29. W. Luk, "A declarative approach to incremental custom computing." In D.A. Buell and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 164-172, Napa, CA, April 1995.

30. P. Lysaght and J. Stockwood, "A simulation tool for dynamically reconfigurable field programmable gate arrays." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 3, pp. 381-390, September 1996.

31. I. Page and W. Luk, "Compiling occam into FPGAs." In *FPGAs, International Workshop on Field Programmable Logic and Applications*, pp. 271-283, Oxford, UK, September 1991.

32. G.M. Quenot, I. Kraljic, J. Serot, and B. Zavidovique, "A reconfigurable compute engine for real-time vision automata prototyping." In D.A. Buell and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 91-100, Napa, CA, April 1994.

33. R. Bittner Jr. and P.M. Athanas, "Computing kernels implemented with a wormhole RTRCCM." In J.M. Arnold and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1997, to be published.

34. J.M. Rabaey, "Reconfigurable processing: The solution to low-power programmable DSP." In *Proceedings of ICASSP'97*, Munich, Germany, April 1997, to be published.

35. M. Rencher and B.L. Hutchings, "Automated target recognition on SPLASH-2." In J.M. Arnold and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1997, to be published.

36. D. Ross, O. Vellacott, and M. Turner, "An FPGA-based hardware accelerator for image processing." In W. Moore and W. Luk, editors, *More FPGAs: Proceedings of the 1993 International workshop on field-programmable logic and applications*, pp. 299-306, Oxford, England, September 1993.

37. H. Schmitt, "Incremental reconfiguration for pipelined applications." In J.M. Arnold and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1997, to appear.

38. M. Shajaan, K. Nielsen, and J.A Sorensen, "Time-area efficient multiplier-free filter architectures for FPGA implementation." In *Proceedings of*

39. M. Shand, "Flexible image acquisition using reconfigurable hardware." In P.M. Athanas and K.L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 125-134, Napa, CA, April 1995.

40. N. Shirazi, W. Luk, and P. Cheung, "Compilation tools for run-time reconfigurable designs." In J.M. Arnold and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1997, to be published.

41. E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon, "A first generation DPGA implementation." In *FPD'94 - Third Canadian Workshop of Field-Programmable Devices*, pp. 138-143, May 1995.

42. S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed FPGA." In J.M. Arnold and K.L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1997.

43. J. Villasenor and W. Mangione-Smith, Configurable computing. *Scientific American* pp. 66-71, June 1997.

44. J. Villasenor, B. Schoner, K.N. Chia, C. Zapata, H.J. Kim, C. Jones, S. Lansing, and B. Mangione-Smith, "Configurable computing solutions for automatic target recognition." In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 70-79, Napa, CA, April 1996.

45. J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, "Programmable active memories: Reconfigurable systems come of age." *IEEE Transactions on VLSI Systems*, vol. 4 no. 1, pp. 56-69, 1996.

47. M.J. Wirthlin and B.L. Hutchings, "DISC: The dynamic instruction set computer." In J. Schewel, editor, *Proceedings of the International Society of Optical Engineering (SPIE). Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, vol. 2607, pp. 92-103, Philadelphia, PA, October 1995.

47. M.J. Wirthlin and B.L. Hutchings, A dynamic instruction set computer." In P. Athanas and K.L. Pocek, editors, *Proceedings of IEEE Workshop on (FPGA)s for Custom Computing Machines*, pp. 99-107, Napa, CA, April 1995.

48. M.J. Wirthlin and B.L. Hutchings, "Sequencing run-time reconfigured hardware with software." In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 122-128, Monterey, CA, February 1996.

49. M.J. Wirthlin and B.L. Hutchings, "Improving functional density through run-time constant propagation." In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 86-92, Monterey, CA, February 1997.
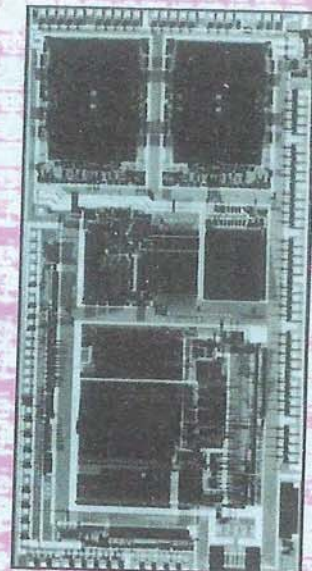
Attachment 3A

SPECIAL ISSUE ON

Hardware and software for DSP

SPECIAL ISSUE ON

HARDWARE AND SOFTWARE FOR DIGITAL SIGNAL PROCESSING

Edited by Sanjit K. Mitra and Kalyan Mondal

SEPTEMBER

COVER   A family of successive digital signal processors (overlaying a programmable bit-level pipelined MAC chip) representing an important development within the topic area of this special issue.

CONTRIBUTED PAPERS

The PROCEEDINGS OF THE IEEE welcomes for consideration technical papers on topics of broad significance and long-range interest in all areas of electrical, electronics, and computer engineering. In-depth reviews and tutorials are particularly appropriate, although papers describing individual research will be considered if they meet the general criteria above. Papers that do not meet these criteria should be submitted to the appropriate IEEE Transactions and Journals.

It is suggested that a prospective author, before preparing a full-length manuscript, submit a proposal containing a description of the topic and how it meets the above criteria, an outline of the proposed paper, and a brief biography showing the author's qualifications to write the paper (including reference to previously published material as well as the author's relation to the topic). If the proposal receives a favorable review, the author will be encouraged to prepare the paper, which after submittal will go through the normal review process.

Please send papers and proposals to the Technical Editor, PROCEEDINGS OF THE IEEE, 345 East 47th Street, New York, NY 10017-2394, USA (Telephone: 212-705-7906).

Data-Driven Multicomputers in Digital Signal Processing

JEAN-LUC GAUDIOT, MEMBER, IEEE

New technologies of integration allow the design of powerful systems which may include several thousands of elementary processors. These multiprocessors may be used for a range of applications in signal and data processing. However, assuring the proper interaction of a large number of processors and the ultimate safe execution of the user programs presents a crucial scheduling problem. The scheduling of operations upon the availability of their operands has been termed the data-driven mode of execution and offers an elegant solution to the issue. This approach is described in this paper and several architectures which have been proposed or implemented (systolic arrays, data-flow machines, etc.) are examined in detail. The problems associated with data-driven execution are also studied. A multi-level approach to high-speed digital signal processing is then evaluated.

I. INTRODUCTION

If we are to approach the computational throughputs equivalent to billions of instructions per second which will be required from the processing systems of the future, improvements on all levels of computer design must be made. Faster technology and better packaging methods can be applied to raise clock rates. However, a one billion instructions per second machine would require a clock period as low as a nanosecond. This approach is inevitably bounded by physical limits such as the speed of light. Therefore, instead of considering the *technological* approach to performance improvement, we emphasize here the *architectural method*. Indeed, instead of merely increasing the clock frequency for a corresponding increase in overall throughput, performance can also be improved by allowing multiple processing elements to collaborate on the same program. This inevitably introduces synchronization problems, and issues of resource allocation and sharing must be solved. Programmability is indeed the central problem. In one solution, a conventional language such as Fortran is used to program the application. A sophisticated compiler is relied upon to partition a sequential program for execution on a multiprocessor. This approach has the

advantage of imposing no "software retooling." However, complex numerical applications will not be easily partitioned and much potential parallelism may remain undetected by the compiler.

Ada, CSP [26], extended Fortran (e.g., HEP, Sequent), on the other hand, allow the programmer to deal with parallel processes by the use of primitives for parallel task spawning, synchronization, and message passing. However, while the programmer can express some of the parallelism characteristic of the application, much potential concurrency may never be uncovered because of the inherent sequential concepts of the language which must be countered through the use of special "parallelism spawning" instructions. Also, development time becomes important since the programmer must "juggle" with many parallel tasks to synchronize. In addition, debugging becomes correspondingly more difficult due to the sometimes undeterministic appearance of errors.

For these reasons, an *implicit* approach must be devised. In the above two methods, instruction scheduling is based upon a central program counter. We propose to demonstrate here the data-driven approach to programming multiprocessors; instructions can be scheduled by the availability of their operands. This model of execution is a subset of the *functional* model of execution [9]. It provides a significant improvement to the programmability of multiprocessors by excluding the notion of global state and introducing the notion of values *applied* to functions instead of instructions *fetching* the contents of memory cells as they are in the conventional "control-flow" model.

The overall objective of this paper is to demonstrate the applicability of data-driven principles of execution to the design of high-performance signal and data processing architectures. Several approaches will be demonstrated and their particular domain of application will be contrasted. The description of low-level processing systems is beyond the scope of this paper and the interested reader is referred to an excellent survey by Allen [3]. Instead, we will concentrate here on the issues related to building high-performance multiprocessors for signal processing applications. In Section II, we show the type of problems considered in signal processing. The data-flow principles of execution as they relate to digital signal processing problems are described in detail in Section III while several exist-

ing data-driven architectures are described in Section IV. In Section V, we analyze a multi-level data-driven architecture and examine its programming environment. Conclusions are drawn in Section VI.

II. THE REQUIREMENTS OF SIGNAL PROCESSING

Digital signal processing techniques are applied to many different technical problems. These include radar and sonar systems, image processing, speech recognition, etc. The elementary building blocks of these were originally concentrated on such tasks as convolution, correlation, and Fourier transform. More complex algorithms (matrix operations, linear systems solvers, etc.) are now considered. Higher order operations include not only simple problems such as elementary filtering (IIR, FIR, etc.), but also more complex functions such as adaptive and Kalman filtering [45]. Also, such complex problems as Computer-Aided Tomography or Synthetic Aperture Radar can be considered [39], [16]. Signal processing algorithms are very appropriate for description by functional languages. Indeed, a signal processing algorithm is often represented in a graph form [36] and can be decomposed in two levels:

• a regular level which can be implemented by a *vector operation* (i.e., a loop in which all iterations present no dependencies among themselves);
• a level which contains conditional operations and heuristic decision making.

This description shows that the lower operational levels can easily deliver parallelism (by compiler analysis or programmer inspection). This layer usually consists of simple constructs (arithmetic instructions, FFT butterfly networks, simple filters, etc.). However, the higher levels will require more complex problem insight and even runtime dependency detection in order to allow maximum parallelism. We will now describe principles of execution which will allow us to deliver this concurrency.

III. DATA-FLOW PRINCIPLES

The data-flow solution to the programmability problems of large-scale multiprocessors [5] has been pioneered by Adams [2], Chamberlin [11], and Rodriguez [43]. It is now described in detail in this section.

A. Basic Principles of Execution

In the conventional von Neumann model of execution, an instruction is declared executable when a Program Counter of the machine points to it. This event is usually under direct programmer control. While a control-flow program is a sequential listing of instructions, a data-flow program can be represented as a graph where the nodes are the instructions (*actors*) which communicate with other nodes over *arcs* (Fig. 1). An instruction is declared executable when it has all its operands. In the graph representation chosen above, this means that all the input arcs to an actor must carry data values (referred to as *tokens*) before this actor can be executed. Execution proceeds by first absorbing the input tokens, processing the input values according to the op. code of the actor, and accordingly producing result tokens on the output arcs. In summary, it can

Fig. 1. A simple data-flow graph.

be said that the data-flow model of execution obeys two fundamental principles:

• *Asynchrony of operations:* The executability of an instruction is decided by a *local criterion* only. The presence of the operands can be sensed "locally" by each instruction. This is an attractive property for an implementation in a distributed environment where no central controller should be used for global scheduling.
• *Functionality of the operations:* The effect of each operation is limited to the production of results to be consumed by a specific number of other actors. This precludes the existence of "side-effects." These side-effects may be long-ranging in that the execution of an instruction may effect the state of a cell of memory which will be used only much later by another unrelated operation.

B. Data-Flow Interpreters

When iterations are executed, the underlying principle of data-flow (*single assignment of variables*) must invariably be violated. Indeed, for an actor to be repeatedly evaluated as in an iteration, its input arcs must carry several tokens (from different iterations). Several solutions have been proposed to allow the *controlled violation* of these rules without compromising the safe execution of the program. Among these, the Acknowledgment scheme and the U-interpreter have been given the most consideration.

1) Acknowledgment Scheme [14]: Proper matching of the tokens can be observed by ordering the token production. This would be done by a careful design of the program graph so as to insure that tokens of two different iterations can never overtake each other. In addition, it must be guaranteed that *no* token pileup is encountered on *any* one arc. This condition can be verified by allowing the firing of an actor when tokens are on all input arcs *and* there are no tokens on any output arcs. In order to enforce this last condition, an *acknowledgment* must be sent by the successor(s) to the predecessor when the token has been consumed (Fig. 2). Note that an actor is executable when it has received its input arguments as well as all acknowledgments. The parallelism which can be exploited from this scheme is mostly pipelining between the actors of different iterations. Thus when the number of instructions in the body of an iteration is the same as the number of available processors, the speedup observed by this mechanism of execution is maximal. However, for small iterations (compared to the size of the machine), the exploited parallelism

Fig. 2. The acknowledgment scheme.

Fig. 3. A typical iterative construct in the U-interpreter.

falls below its potential. Thus it may be required that the compiler effect a *code expansion* for vector operations.

An important characteristic of this *static* model of execution is the fact that it allows only one instance of an instruction to exist at one given time. In other words, it is primarily relied upon *pipelining* for the exploitation of parallelism in iterations. However, the basic acknowledgment scheme does not allow the implementation of multiple simultaneous calls to the same function. Several machines which obey these principles of execution have been designed: the MIT cell block architecture [13], the Hughes Data-Flow Machine [21], the DSFP [24], the USC TX16 [19], etc.

2) The Unraveling Interpreter (U-Interpreter): The U-interpreter [4] provides the most asynchronous possible operation. In order to allow safe execution of actors in an iterative construct, tokens are tagged with information pertaining to their context of creation. An actor is only allowed to execute when an input token pair with matching tags can be found. This tag includes the iteration number. Indeed, the U-interpreter closely follows these principles: to each data token is attached a tag of the form $u.P.s.i$, where P identifies the procedure name of the destination actor, while s is the address of this actor within procedure P. The i field corresponds to the iteration number in which the token was created, while the u field is the context of creation. Note that while the former is used to distinguish between tokens destined to different iterations of the same actor, the latter is used in situations involving multiple function calls, operations with recursive function calls, or nested iterations.

Special actors are used which deal with the context and iteration fields of the token tags. A typical iteration construct in the U-interpreter is shown in Fig. 3. The D actor is used to recirculate the data from one iteration to the next. Its input is tagged with $u.P.t.i$ while its output value is identical but has become tagged with $u.P.t'.i + 1$. Nested iterations are handled by *isolating* the inner from the outer iteration by the introduction of the L actor at the top of the graph. The function of this actor is to *create a new context* for the execution of the iteration: the input tokens are tagged by $u.P.s.i$ while the output tokens are identical but are tagged with $u'.P.t'.1$ where u' is itself $u.i$. Note that this mechanism is sufficient to create an entirely different set of tokens for two nested iterations. Indeed, assume that two tokens both belong to the same inner iteration i, but belong to the j_1 and j_2 outer iterations respectively. The first token

would be tagged $u1.P.s.i$ ($u1 = u.j_1$) while the second is tagged with $u2.P.s.i$ ($u2 = u.j_2$). This shows that an appropriate differentiation has been made between the two instances. The original context u is retrieved by the L^{-1} actor before exiting.

Contrarily to the acknowledgment scheme, this *dynamic* data-flow scheme allows full asynchronous execution of the program graph. Indeed, due to the scheme of tags, several *instances of the same instruction may exist simultaneously*. Vector operations may be executed in parallel without compiler-induced replication of the graph. Likewise, multiple function calls and more particularly recursions are allowed since each new actor instantiation receives a different tag. This means that the U-interpreter would be preferred to the static model when the ability for fast recursive calls is required. However, this flexibility comes at the expense of added hardware complexity. Indeed, it will be shown in Section IV-D that implementation of the U-interpreter requires an associative memory for fast tag matching. Several machines based on these principles have been studied: the MIT tagged token data-flow machine [6], the ESL DDSP [28], the University of Manchester machine [23], the ETL Sigma-1 [25], etc.

C. Structure Handling

This is a crucial issue in signal processing for this kind of application requires that many data elements which belong to the same structure be processed in a parallel or pipelined fashion. One of the basic premises of data-flow principles states that an output is a function of its inputs only, regardless of the state of the machine at the time of execution. When a structure of elementary elements must be processed, the absence of side-effects means that it may not be updated for this would imply its transition through several states. Instead, if any updates are needed, a new array which contains the new elements must be created. Copying of all elements must be undertaken for the modification of a single one. This solution imposes an inordinate overhead. This is why the implementation schemes we will now describe can shortcut this complete copying while

preserving the meaning of the program during array update operations.

1) Heaps: Dennis [14] has proposed to represent arrays by directed acyclic graphs (also called *heaps*). Each value is represented as the leaf of a graph tree. The modification of a single element in a heap is represented in Fig. 4. Note

Fig. 4. A heap update.

that the complexity of the modification of a single element of the array is $O(n)$ for a copy operation, while it is $O(\log n)$ for the heap. Several instructions are exclusively devoted to the access of heaps [15]: SELECT receives a pointer to the root node, an index value, and returns a pointer to the substructure (which may be a leaf node) associated with the index; APPEND also needs the same two operands in addition to the value of the element to append to the structure.

2) I-Structures: A heap must be entirely ready before it can be consumed because no consumption (SELECT actors) can take place until the pointer token appears (i.e., the creation of the array is completed). In the I-structure scheme [7] constraints on the creation of arrays allow the selection of individual elements (or substructures) from the array before its complete production. One possible implementation of I-structures makes use of a "presence" bit which indicates when an element of an array has been calculated and is ready for consumption. An attempt to read an empty cell would cause the read to be *deferred* until such time that the cell presence bit is marked. Conversely, a write into a cell, the presence bit of which indicates valid stored data, could be cause for the generation of an error signal. The advantages of this scheme are:

* better performance because pipelining is allowed between I-structure consumers and producers;
* less "serialization" of operations such as APPENDs, because they are allowed to occur independently on the same structure.

3) HDFM Arrays: A special scheme for handling arrays in a VAL high-level environment has been designed for the Hughes Data-Flow Machine (HDFM) [21]. It uses the fact that data-flow arrays as described above are overly "asynchronous," i.e., they do not take advantage of the data dependency information carried by the program graph. Safety of accesses is respected by not allowing the updating of an array before all the reads from the *current version* of the array have been performed. Only then can the array be directly modified. Safety and correct execution of WRITE operations are a compile-time task. This has the advantage of reducing the number of memory accesses (no complex graph of pointers must be travered as in heaps) as well as of offering a better possibility of distribution of an array (no root node). However, spurious data dependencies may be introduced because the compiler is not necessarily aware

of the possibility of parallelism that can be detected only at runtime. For instance, dependencies on A and B related by $A(F(i)) = B(i)$ may be artificially imposed. However, the applications targetted by the HDFM include some amount of regularity which can be easily detected by the compiler and implemented as conventional arrays.

4) Token Relabeling [18]: In the U-Interpreter, the notion of array can be entirely ignored at the lowest level of execution. Instead, the *tag* associated with each token under the rules of the U-interpretation is used as identification of the index of the array element of the high-level language. In other words, it can be simply said that, when an array A is created, its $A(i)$ element will be tagged with i (hereafter denoted $A(i)_{(i)}$), if the elements are produced in the logical order. In the "production" of a unidimensional array, the iteration number can usually be directly interpreted as the *index* of the array element just produced by the iteration.

Special token relabeling program graphs can be created to handle *scatter* and *gather* program constructs [27] (Fig. 5(a)). This figure shows that an inversion function F^{-1}

(a)

(b)

Fig. 5. (a) A gather operation. (b) Token Relabeling gather.

This demonstrates that, without recourse to the calculation of F^{-1}, the proper relabeling of the A elements has been effectively produced.

This algorithm requires no intermediary storage, does not need array operations, and imposes smaller hardware and execution overhead. This relabeling approach eliminates a large portion of the overhead associated with the production and consumption of array A. Pipelining between the source and the sink of a data structure is the goal of this unknown at compile time would be needed to perform the relabeling of data-flow tokens. Such a calculation is not truly necessary. Instead, we introduce (Fig. 5(b)) a sequence generator which produces the $F(j)$'s, tagged by j. An *exchanger* actor (called χ) swaps the tag and the data value and produces $j_{(F(j))}$. Both streams (the A's and $j_{(F(j))}$) are input to a special relabeling actor δ which only modifies the iteration portion of the tag. By the principles of the U-interpreter, only tokens which bear the same tag will be matched as proper inputs to the δ actor. In other words, the mate of

a relabeling actor which takes the A input and relabels it with the data carried by the token on the other input arc. In other words, it outputs $A(F(i))_{in}$. Since i is a dummy variable, and since F is bijective, it can be said that, on a global point of view:

$$A(F(i))_{in} \Leftrightarrow A(k)_{[F^{-1}(k)]}.$$

scheme, just as it was the idea behind the design of the I-structures. However, the token relabeling approach brings a better runtime memory management since tokens corresponding to the various elements of the array still exist and must still be temporarily stored, they need not go through an additional storage as a data structure. Also, there is no need for "requests" for data as would be the case in an I-structure environment: When an I-structure is created, actors which need data from it must demand the element in question until it has arrived. This may introduce heavy overheads as unsatisfied requests must be queued in the structure itself. Garbage collection is automatically handled since when the "array token" is matched, it is automatically removed from the arc. In other words, when it has been used, it is swallowed by applying data-flow principles.

D. High-Level Data-Flow Languages

In addition to the low-level mechanisms of execution which were described earlier, special high-level data-flow languages have been designed for easier translation into data-flow graphs. To be sure, these high-level languages are not a necessity: the Texas Instruments data-flow project [31] relied upon Fortran programming through the use of a modified version of a vectorizing compiler originally destined to the TI ASC. However, many high-level languages have been designed for data-flow prototypes. Most notable are VAL (Value Algorithmic Languages) for the MIT static data-flow project [37], [1], Id (Irvine Dataflow) for the MIT tagged token data-flow architecture [4], LUCID [8], [30], etc. SISAL (Streams and Iterations in Single Assignment Language) has been designed by McGraw and Skedzielewski [38] and is intended as the definition of a "universal" language for the programming of future multiprocessors.

Data-flow languages have also been defined for the specific purpose of programming signal processing applications. These include the SIGNAL language designed by Le Guernic et al. [36]. The intent of the language is to provide a formal specification of signal processing problems and to ease the design of signal processing multiprocessors, be they special- or general-purpose. One of the main characteristics of the language is that it incorporates the notion of *time* to describe the interaction of the various processing tasks. This makes it a *synchronous* language as opposed to asynchronous languages such as CSP and Occam [10]. SDF (Synchronous Data Flow) is another formal description of signal processing algorithms based on data-driven principles of execution proposed by Lee and Messerschmitt [35].

IV. Data-Driven Architectures

We now describe in detail several systems which operate at runtime, compile-time, or design-time under data-driven execution. Although it is generally considered that data-flow principles of execution are in effect at runtime, we extend their domain of application to design or compile time and refer to them as *data-driven systems*. We thus initially examine multiprocessor systems where data dependencies have been frozen at design time (systolic arrays). We then consider programmable systolic arrays (the Wavefront Array Processor) and multiprocessors scheduled at compile time by the use of data-flow program graphs (the ESL polycyclic processor). Finally, we study systems where the data dependencies provide scheduling information at runtime (the Hughes Data-Flow Machine) and examine the influence of the level of resolution upon the performance (the USC TX16).

A. Systolic Arrays [32]

The primary goal of a systolic array is to make use of the large amount of processing power available in VLSI technology by using repetitive circuitry to perform signal processing problems, matrix operations, image processing, etc. In summary, a systolic array is simply a collection of interconnected Processing Elements (PEs). In order to incorporate as many processors as possible, the structure of the PEs themselves is kept to a maximum simplicity and usually includes only a few operation units. For design simplification, there are few types of PEs in the same system. By the same token, interconnections are kept to a nearest neighbor topology in order to minimize communication delays as well as power distribution issues. Note that topologies include two neighbors (linear arrays), four neighbors (square arrays), or six neighbors (hexagonal arrays) as required by the problem to solve. This is notably due to the fact that scheduling mechanisms must be based upon *local* criteria such as data availability. However, it should be noted that there is a global clock in all the computation cells. Linear systolic arrays can tolerate clock skews at both ends, but multidimensional designs require slower clocks in order to compensate. In order to simplify runtime mechanisms, the design of a systolic array emphasizes an efficient mapping of the problem onto the architecture. An example of a band matrix–vector multiplication is shown in Fig. 6. It displays

Fig. 6. A linear systolic array.

how the synchronization of the processors and of the input data rate has been mapped to meet the requirements of the problem. Note that each processor is designed to operate upon *the arrival of the arguments*. In summary, it should be

1224

noted that systolic arrays are very efficient at computationally intensive problems which involve many repetitive low-level calculations. Also, the very nature of their design renders their function fixed at design time.

B. The Wavefront Array Processor (WAP) [33], [34]

Execution on the WAP is similar at runtime to the execution of a program on a systolic array. Indeed, both approaches rely upon the scheduling of operations based on the availability of their operands. However, the analysis of the data dependencies is effected during the design of a systolic array while the WAP is scheduled by compiler detection of parallelism: the WAP is a "programmable systolic array." It has been shown that most signal and data processing algorithms possess a certain amount of *locality* and *recursivity*. They will thus exhibit the phenomenon of *computational wavefront*. This has an important implication in that an entire *front of processors* can be programmed for the same operation. In addition, it can be shown that two successive wavefronts cannot intersect. This enables the proper implementation of data-driven principles of execution. For instance, a matrix multiplication can be executed as a computational wavefront (Fig. 7). A special

$$C_{ij}^k = C_{ij}^{k-1} + A_{ik} \times B_{kj}$$

Fig. 7. Matrix multiply in the WAP.

language called the Matrix Data-Flow Language (MDFL) has been designed to express such algorithms on the WAP.

C. ESL Polycyclic Architecture [41]

The ESL polycyclic architecture is a horizontally microprogrammed multifunctional vector processor. It comprises several functional units (adders, multipliers, storage units) connected by a cross-bar interconnection network. Entire vector loops (no data dependencies across the iterations) can be scheduled by using the model presented in [40]. The essential idea is to discourage "greedy" scheduling by insertion of "non-compute" delays in the train of calculations. The effect of these delays is to enable an optimal schedule. A pipeline is viewed as a certain number of resources (the various segments of the pipe) which can be reserved by tasks. The problem is reduced to the produc-

tion of a reservation table with no collisions (i.e., no two tasks can reserve the same segment of the pipeline at the same time). For that purpose, a *usage interval* is defined as the time interval between two reservations of a segment by a single task. Two tasks will collide when they have the same initiation time as one of the usage intervals. For a homogeneous multiprocessor (identical PEs), the method is done in two steps:

1) Determine the Minimum Initiation Interval MII as $MII = \lceil N/P \rceil$. N is the number of instructions in the body of the loop, and P is the number of processors available for execution. The initiation interval is the length of time between the initiation of two consecutive iterations. Successive iterations will be scheduled at MII units interval. All the iterations will be identically scheduled.

2) Schedule the operations in accordance with the data dependences. However, no more than P operations may be scheduled for the same time modulo MII. Note that this last constraint also implies that delays must be inserted in the schedule.

The following example shows the scheduling of a simple vector operation (Fig. 8(a)) on a polycyclic processor with two adders and one multiplier (note that for simplification, communication costs have been assumed to be null). There are two multiply operations for a single multiplier while there are three additions on two adders. The MII would therefore be 2. This means that one iteration of the loop can be performed at a rate of one for every two cycles. By using the data dependency graph of the example (Fig. 8(b)), the optimal schedule can be used by applying the MII of 2 (Fig. 8(c)). Proper "dovetailing" of successive iterations is assured

Fig. 8. (a) A simple vector operation. (b) Corresponding data dependency graph. (c) Scheduling a single iteration. (d) Dovetailing iterations.

by this scheduling algorithm when the iterations are processed at the rate of one for every two cycles (Fig. 8(d)). This architecture applies particularly well to signal processing applications where the same computation must be repetitively applied to a different element in a steady data stream.

For example, a butterfly block in an n-point FFT operation would be executed $n \times \log n$ times. However, in addition to the purely computational actors shown above, "store" and "retrieve" operations should also be considered. This is demonstrated in the data-flow graph of Fig. 9(a) which

(a)

Time	Multiplier	Adder#1	Adder#2	Mem1	Mem2	Mem3	Mem4
0	-	-	-	R1	R2	-	-
1	-	-	-	R3	-	-	-
2	M1	-	-	-	-	-	-
3	-	A1	A2	-	-	-	-
4	-	-	-	-	-	W1	W2

(b)

Fig. 9. (a) FFT butterfly block. (b) Scheduling of an FFT butterfly.

corresponds to a single iteration (butterfly block) of a real FFT. Note that the indexes (for reads and writes) have been assumed to be generated elsewhere (e.g., table look-up) and are ignored in this discussion for simplification. Assuming that two adders and one multiplier can be used, and that we have four memory modules at our disposal, the MII can be determined as the *maximum* of the N/P ratio for each kind of resource. This yields an MII of 2 (Fig. 9(b)). Note that further work in the scheduling of iterations has been carried out in [44]. This research is also applied to SSIMD architectures and allows the existence of dependencies between iterations.

D. The MIT Tagged Token Data-Flow Machine [6]

This machine implements a version of the U-Interpreter. In this distributed architecture model, each PE is independent from its neighbor and there is no global controller. A hypercube communication network allows the transmission of data-flow tokens between PEs. Store-and-forward capabilities are provided so that a pair of PEs which is not directly linked may still communicate.

The structure of each PE is shown in Fig. 10. A switch receives tokens from the network and determines whether the incoming packet is a data token or a structure request to be processed by the I-Structure Memory. In the Matching Store Unit, the tag of the incoming token is associatively checked against that of previously arrived tokens to determine whether it is the first or the second token to arrive at a given instruction. The first token should be stored in the

Fig. 10. A PE in the tagged token data-flow machine.

associative memory of the Matching Store Unit and held until its mate arrives. For the second token, the corresponding instance of the instruction can be activated by sending an *argument packet* to the next unit. The Instruction Fetch Unit receives this packet and fetches the parameters of the instruction. Note that the template contains not only the op.code but also pointer(s) to the destination actor(s) to which the result of the operation should be sent. A complete *instruction-ready packet* can be formed and sent for execution to the ALU. The ALU blindly executes the operation indicated by the incoming template and produces result tokens which are received by the Token Formatting Unit. Finally, the Token Formatting Unit receives tokens which have been produced by the ALU. These tokens comprise several fields: the tag associated with the operation (after modification if the operation was a tag-modifying operation), the data themselves, as well as an *allocation function* field. This field is used by the Token Formatting Unit to determine the destination PE of the token. Indeed, this determination cannot often be made solely on the basis of destination actor for this would mean allocating to the same PE all the iterations of an actor in loop. This is clearly unacceptable if parallelism is to be extracted *across* the iterations of the loop. An often used heuristic allocation function is based upon calculation of the iteration number modulo the total number of PEs. This function has the advantage of allowing proper distribution of a loop across the machine. Depending upon existing conditions, different allocation functions may be used within the same graph. However, it must be noted that the function must be the same for the two tokens destined to the same actor. Failing the verification of this condition, the two tokens would never be matched for they would be sent to different PEs. This demonstrates the need to implement this allocation operation at *compile time*.

E. The Hughes Data-Flow Machine (HDFM) [21]

The goal of this project is to provide a high-performance parallel architecture which is highly programmable and at

1226

the same time offers advantages of modularity and simplicity of implementation for signal and data processing applications. All communications are based on the *message passing* model. A maximum of 512 PEs can be organized in a cube network. Each PE is attached to three busses (row, column, and plane). Design of the PEs has been made for easy implementation in VLSI. PEs can easily be added because of the modular nature of the communication network. Traffic on each bus is based upon contention before data can be transmitted. Any two PEs can communicate by a maximum of three "hops" (Fig. 11). The execution model

Fig. 11. Structure of the HDFM communication network.

is based upon the acknowledgment scheme. Instead of using "hardwired" acknowledgment arcs between two communicating actors, this machine is based upon the principle of "software" acknowledgments. The compiler partitions the data-flow graph into blocks. Special acknowledgment arcs are introduced between the blocks. Note that this method allows pipelining between iterations at the block level.

One of the primary requirements of the machine was to incorporate as few component parts as possible. This

implies a high level of integration for the individual PEs. Indeed, each PE consists of only two custom-designed chips in addition to several commercially available memory circuits. The overall architecture of a PE is shown in Fig. 12.

Fig. 12. A PE in the HDFM.

The COM chip handles all the communication functions and interfaces the actual PE with the three-bus communication network. It implements the necessary "store-and-forward" and performs in addition a buffering function in order to even packets rates. Note that the chip pin-outs requirements limit the number of outside busses to 3. The PROC chip is the actual PE which contains three pipelined stages: 1) instruction/operand fetch and data-flow firing rule check, 2) instruction execution, and 3) result token formatting. It can be easily represented schematically (Fig. 13). Tokens arriving from the COM chip are first checked to determine whether they complete an instruction packet or not. Ready instructions are then dispatched to the exe-

MICROMACHINES
COM: COMMUNICATIONS/FAULT TOLERANCE
TMC: TEMPLATE MEMORY CONTROLLED
ALU: ALU/MICROPROCESSOR AND MICROMEMORY
DT: DESTINATION TAGGER
DMC: DESTINATION MEMORY CONTROLLER

MEMORIES
DM: DESTINATION MEMORY
TM: TEMPLATE MEMORY
FDM: FIRE DETECT MEMORY

QUEUES
FCQ: FROM COMMUNICATION QUEUE
FBQ: FEEDBACK QUEUE
FQ: FIRING QUEUE
RQ: RESULT QUEUE
DQ: DESTINATION QUEUE
AIQ: ASSOCIATED INFORMATION QUEUE
TCQ: TO COMMUNICATION QUEUE

Fig. 13. The PROC chip in the HDFM.

Destination Tagger and sent to the COM chip or back to the input Token Unit as the case may be. The Template Memory Controller (TMC) enforces the data-flow rules of execution and checks the completion of an input set before it sends a complete instruction packet to the ALU for execution. Results are sent by the ALU to the Destination Tagger (DT). In collaboration with the Destination Memory Controller (DMC), this unit associates the data values produced by the ALU with their proper destination address (which can be found in the Destination Memory DM). Note that in the Cell Block architecture, the templates are wholly stored in one location. In this architecture, the templates are, instead, split in two portions: the op. code and input operand portion stored in the Template Memory TM, while the corresponding result pointers are stored in the Destination Memory. The rationale for this design decision can be found in two points: first, this allows a better space management since the destination list of any template may be of undeterminate length. Second, if the whole template were to be stored in the Template Memory, the Destination pointer information would have to be propagated through the ALU before it could be used only in the last stage of the processor. This would prove particularly inefficient since the whole processor must be integrated on a single chip, thereby multiplying the area necessary for busses. Simulation of radar processing algorithms has demonstrated that each PE capable of a throughput of 2–4 MIPS while a 64 PE could produce throughputs of 64 MIPS.

F. The NEC μPD7281 [12]

The NEC μPD7281 is a single-chip digital signal processor. Its most important application is image processing. Some immediate applications include image restoration, enhancement, compression, and pattern recognition. It is based on a data-flow model of computation and implements such complex operations as multiplication in the basic instruction set. More specifically, its primitives are designed for an efficient execution of image processing algorithms. The use of data-flow principles of execution increases the programmability of the machine and renders the multiprocessor architecture invisible to the programmer. Another characteristic of the μPD7281 is that it can be cascaded with several other identical chips in a *ring* architecture. Indeed, the architecture of the μPD7281 enables the design of multiprocessor systems for improved performance. By cascading several such PEs, a high degree of pipelining can be observed. In addition to the high-level organization of the chips, each chip is itself organized in a ring architecture (Fig. 14) which operates in a pipelined fashion.

Constants may be stored in the Data Memory for storage during execution. The program is represented in both the Function Table and the Link Table. In the Function Table, the actors themselves are stored. Similarly, the Link Table contains a representation of the arcs between the actors. After the initial loading of the PEs, when a token enters a μPD7281, it is first checked to determine whether this PE should process it. If not, it is directly transferred from the Input Controller (IC) to the Output Controller (OC) where it is forwarded to the next processor along the chain. Otherwise, it can be sent to the Link Table for processing. In the Link Table and the Data Memory, it is matched with other tokens before it can be sent for actual processing. The Link Table always contains the first of the two operands that arrive. The Address Generator and Flow Controller are used to generate addresses of constants. Note that after actual processing of a data-flow actor in the Processing Unit, the resulting token is again processed by the Link Table of the same Processing Element. When the ultimate consumer of the token is allocated to another PE a special output instruction is executed in the queue so that the token can be switched to the Output Controller. Overall, this circular pipeline contains seven segments and can deliver a maximum throughput of one instruction per cycle. The primitives of the μPD7281 are oriented towards image processing applications:

- CONVO (Convolve) which can be used to perform cumulative operations such as

$$\sum_{i=1}^{n} A_i B_i$$

Fig. 14.   The NEC μPD7281.

IN1   IN2
B₁   B₁

QUEUE
MUL

C_1

CONVO

ID   ID+2
                →SUM $= \sum_{i=1}^{n} A\,B$
    ID+1

QUEUE
ADD

Fig. 15.   A convolver actor.

(Fig. 15). Note that this operation is not strictly speaking a data-flow actor in that the summation implies a "state" of the actor. Formally, it would correspond to a "macro-actor" [20] which includes a graph of several elementary data-flow actors.

- ACC (Accumulative Addition Instruction).
- Bit manipulation, data conversion instructions, etc.

The multiplication of a 3 × 3 matrix by a 3-element vector is illustrated in Fig. 16. It is assumed that the A matrix has

B_1 B_2 B_3

COPY OK   COPY OK   COPY OK

READ $A_1(i)$ READ $A_2(i)$ READ $A_3(i)$
     *   *   *

QUEUE
+

QUEUE
+

DIST
OUT

C_1 C_3
       C_2

$$\begin{bmatrix} A_1(1)\ A_2(1)\ A_3(1) \\ A_1(2)\ A_2(2)\ A_3(2) \\ A_1(3)\ A_2(3)\ A_3(3) \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix}$$

Fig. 16.   Matrix vector multiplication.

already been allocated to the data memory. Each element of the vector B is received and is replicated three times by the COPYBK actors. Note that one of the internal parameters of the actor is 3, the number of required replications. The *multiply* actor is a coupled with a *readA$_i$(j)* actor so as to perform the multiplication of B_i with $A_i(j)$. The results from the three multiplications are matched and accumu-

lated to produce the result C_i. Note that the queue actors enable a better pipelining of the successive computation waves. They allow an execution similar to the model presented by the Acknowledgment scheme (Section III-B1). However, while in the Acknowledgment scheme only one token is allowed at any time on any single arc, this model of execution allows as many tokens per arc as the size of the queues. Benchmark evaluations have shown a near linear speedup with increasing numbers of chips from 1 to 3:

Algorithm	1 PE	3 PE
512 × 512 binary image rotation	1.5 ms	0.6 ms
512 × 512 binary image $\frac{1}{2}$ shrinking	80 ms	30 ms
512 × 512 binary image smoothing	1.1 s	0.4 s
512 × 512 binary image 3 × 3 conv.	3.0 s	1.1 s
64 stage FIR filter (17 bits)	50 μs	18 μs
cos(x) (33 bits)	40 μs	15 μs

G. The USC TX16 [19]

The TX16 is based upon the Inmos Transputer. The Inmos Transputer has been heralded as the first of a new generation of microprocessors. Indeed, while conventional microprocessors are interfaced with the external world through a single memory bus (address, data, and control), the Transputer possesses in addition four serial communication links. Each of these communication links allows point-to-point transmissions between two Transputers. This architecture is reflected at the language level: the arrival of data on a link will trigger a process inside the receiving Transputer.

The programming language Occam allows the presence of several different processes while only one is active at a time. Message transmission with other processes is based upon the synchronous principles of CSP [26], [10]. This means that when the active process must communicate an intermediary result with another process (possibly located in another processor), the active process is held until the other process has been found to be ready for the transmission. While the process is held, it is stacked into the inactive process queue. Another ready process is then activated until it either terminates or is itself hung because of a required transmission. This low-level context change mechanism compares favorably to the busy–wait model found in conventional multiprocessor systems. Instead of idling a processor while waiting for an intermediary operand to arrive, the system allows context switching to another ready process.

The system consists of 16 interconnected Transputers interconnected in an ILLIAC-IV topology. The four links of each Transputer are used for scalar data communications and for interprocess synchronization messages. Each PE owns a single bank of the memory system (Fig. 17). A processor can directly access its own memory bank through the external memory bus of the Transputer (*local access*). A *remote access* can be made into the bank owned by another processor. In this case, the Bus Controller formats the request from the PE into a packet and takes control of the bus. The request is then forwarded to the destination PE. When the request is a read request, a response will be sent in the same fashion back to the originator.

The data-flow language SISAL (Section III-D) was chosen as the high-level interface for the TX16 because the data-flow principles of execution can be directly mapped into Occam.

Fig. 17. A Processing Element in the TX16.

The converse is not true, however, since it is possible to design unsafe Occam programs which would have no corresponding part in the data-flow world. This mapping is made possible by the fact that both programming approaches rely upon the principles of *scheduling upon data availability*. Several numerical algorithms have been tested on a simulated machine and have demonstrated a near linear speedup for the size of the machine considered. It should be noted that this was obtained without the intervention of a sophisticated high-level language compiler. Instead, a data-flow language was used to provide the *programmability* needed. Indeed, for the same programming effort, a higher speedup would be obtained by the data-flow approach.

H. Comparison of Data-Driven Architectures
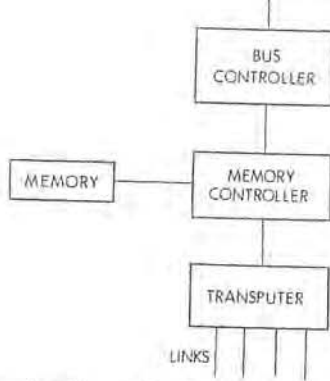
The different architectures presented in the above sections all represent different approaches to the problem of specifying scheduling in multiprocessor systems for digital signal processing applications. They each fit a different niche in the realm of problems encountered in this domain:

* The *systolic method* efficiently and cheaply implements parallel algorithms on potentially large numbers of simple processing elements. However, the design of the algorithm on the array of processors remains fixed and constrains the system to consistently solve the same problem.
* Architectures such as the *WAP* and the *ESL polycylic processor* possess a greater degree of programmability. The WAP notably has no global synchronization mechanism since it relies upon the notion of a computational wavefront.
* The *data-flow multiprocessors* which we have described (the HDFM, the MIT tagged token data-flow machine, and the USC TX16) offer much more flexibility in that their scheduling is in a larger part decided at compile time. They possess no notion of central control and can deliver maximum parallelism in very complex algorithms without any intervention from the designer, programmer, or compiler. Data-flow machines find their applications in two cases: 1) in problems which involve large amounts of heuristics and decision making, or 2) in applications which require frequent reprogramming, thereby requiring the *high programmability* characteristic of data-flow systems. The data-flow interpretation model also presents the crucial advantage of *scalability* in that the same programming

method of a given algorithm can be used, regardless of the size and topology of the target machine. Finally, the *programmability* afforded by this approach translates into a higher performance for a given amount of programming effort.

V. A DATA-FLOW ARCHITECTURE WITH MULTIPLE LEVELS OF RESOLUTION

The data-driven model of execution has thus been demonstrated to provide a very efficient programming environment for the parallel execution of programs. We now show how the concentration of the model on small atomic operations can lead to many runtime inefficiencies. We examine the performance of a multi-level architecture.

A. The Multi-Level Approach

It has been observed [17] that the data-flow model of execution was often applied at too low a level and imposed much overhead at runtime. For instance, as was demonstrated in Section III, the description of a simple loop under the principles of the U-interpreter can impose a large number of overhead actors such as D, L, etc. For each loop, a minimum of five actors must be included. In addition, since we are in a data-driven environment, each datapath in the same loop (for instance, the index and the iterated variable) must "own" their own set of iteration actors, thereby multiplying the overhead. Let us consider a simple vector operation (Fig. 18(a)) which would be translated into the graph of Fig. 18(b). This is obviously a large overhead. Indeed, the data-flow interpretation mode should be used to uncover at runtime parallelism which would be difficult or impossible for a compiler to detect. Here, a relatively easy compiler intervention would quickly detect and exploit the parallelism available in the vector operation while the data-flow constructs would impose a large overhead.

This shows that the data-flow principles of execution can be advantageously applied with a higher granularity as it has been demonstrated [20], [42]. It is indeed intended to retain much of the distributed concepts introduced in the tagged token data-flow machine [22]. The architecture we consider comprises a large number of independent PEs which can communicate over a packet-switched interconnection network. The size and structure of the individual PEs, however, should match the higher granularity envisioned in this project and would implement powerful primitives such as complex vector operations.

The architecture of the machine is to be organized in a hierarchical fashion. It respects at the higher level the data-flow principles of execution but comprises powerful PEs at the lowest level. The PEs are to be vector processors. The advantages brought by this approach are several-fold:

* The principles of data-flow are maintained at all levels of execution which implies the same programming model. (Indeed, a vector operation can easily be detected in a higher level data-flow language.)
* A continuous succession of more powerful but conversely more tightly coupled levels is implemented.
* The increase in performance brought by higher granularity can be directly implemented on this hierarchy of levels with increasing communication costs.

```
FORALL i in [1,N]
  c = a(i)+b(i)
  returns array of c
ENDFOR
```

(a)

(b)

(c)

Fig. 18. (a) A vector addition in SISAL. (b) U-interpreter graph of a vector addition. (c) IF1 representation of a vector addition.

B. The Elementary Processing Element

In order to apply the data-flow principles of execution at a higher level of granularity, it appears that the individual PEs should be dealing with complex constructs which are more representative of the application under study. In the case of large signal and data processing problems, the input data which are processed are usually received at a high rate. Each datum undergoes the same operation and interacts only slightly with other data elements. This is the case for image processing applications in which *local* transforms are usually undertaken. This demonstrates that signal processing problems usually entail a low level which corresponds to *vector operations*. However, the highest level of operations includes such constructs as conditional, decision making, etc., for which the data dependencies cannot be identified clearly at compile time as they can be in a vector operation. Instead, *runtime dependencies* must be detected in order to provide safety of execution and adequate scheduling of our multiprocessor organization.

The individual PE is organized around a vector processor but also includes the capability to perform scalar operations as needed. The vector architecture can remain unspecified for the purpose of this section but could be an SIMD processor, a pipelined vector architecture, etc. For proper I/O function (i.e., communication with other PEs), the Processing Element is separated in the actual Processing Unit and the Communication Unit. The Processing Unit implements the actual vector functions while the Communication Unit is responsible for transferring data packets to/from the communication network and for the forwarding data packets to other processors.

C. The Software Environment

The elementary principles of execution are based upon an application of multi-level data flow. It was earlier demonstrated that the high granularity would considerably and positively affect the performance of a data-flow system. However, it is also known that the high-level, statically scheduled data-flow programming methodology could be used to design extremely powerful vector processors. The architecture we have described comprises therefore two levels: sophisticated processors are connected into a second hierarchy. This hierarchy also exists on a software point of view: program constructs must be partitioned together in order to best utilize the characteristics of the architecture.

We have chosen for our high-level programming interface the paradigm provided by SISAL (Streams and Iterations in a Single Assignment Language) as introduced by McGraw and Skedzielewski [38]. As shown in Section III, SISAL is a high-level language, the syntax of which resembles Pascal. It is different from conventional languages, in that it contains none of the side-effects associated with the usual programming approaches. An example of the specification in SISAL of the addition of two arrays A and B was given in Fig. 18(a). The existing compiler provides an Intermediate Form output (IF1). Not only does this output include the data-flow graph necessary for the runtime detection of data dependencies (called Data-Flow Graph DFG), but it also includes program structure information (Program Structure Graph PSG). As an example, the IF1 representation of the SISAL program in Fig. 18(a) is shown in Fig. 18(c). It shows the Program Structure Graph (PSG) in solid lines, while the actual Data-Flow Graph (DFG) is represented by dashed lines. The *forall* pseudo-node belongs to the PSG and is the head of a three-pronged tree: the left-most node contains the RangeGenerator actor which produces the index I from 1 to N (see SISAL program). The middle pointer is the actual

from the body of the loop. The right-most node *gathers* the results is received by both array selectors (*AElement*) which receive A (respectively, B), and I and produce A(I) (respectively, B(I)). The *Plus* operator adds the two. Partitioning can easily be done along the edges of the PSG, provided a *cost matrix* is kept in order to easily assess the communication costs among the modules so isolated. An immediate heuristic comes to mind concerning the system under study: since the atomic vector processing unit is so well tuned to the scheduling of Generalized Vector Computations (GVCs), the partitioning process should examine the PSG from its leaves until it encounters a FORALL pseudo-node. A partition which would comprise the whole subgraph can thus be created. In addition, it should be noted that beyond this relatively simple partitioning approach, several optimization methods have been implemented. For example, nested loops can be exchanged or combined, code could be *hoisted* when data dependencies allow. These and other strategies have also been described in [29].

D. Applications

Kalman filtering [45] can be chosen as a representative example of some signal processing algorithms. It maps remarkably well on our architecture because of the multiple levels of hierarchy which are embedded in the algorithm itself. Indeed, the entire system could be described at the low level used by "conventional" data-flow architectures (Section IV). However, it should be immediately noted that most of these low-level operations can be grouped into higher order tasks. For example, the computational block which corresponds to covariance matrix estimation implements a complex matrix inversion. This algorithm itself entails repetitive applications of transpose operations.

As an illustrative example of the matrix operations which can be directly mapped onto our architecture, we have chosen the multiplication of two matrices. The SISAL code which corresponds to matrix multiplication is shown in Fig. 19(a). The corresponding IF1 output is shown in Fig. 19(b). In the graph, actors "RangeGenerator1," and "RangeGenerator3" broadcast index values i and k to the actors "AElement1" (Array Element select) and "AElement2," respectively. Once the actors "AElement1" and "AElement2" have received the index values, they forward the pointers A[i, *] and B[k, *] to the actors "AElement3" and "AElement4." "AElement3" and "AElement4" are also waiting for the index values k and j which are sent from the actors "RangeGenerator3" and "RangeGenerator2," in order to generate the elements A[i, k] and B[k, j], respectively. The "Times" actor receives the two elements A[i, k] and B[i, j] and sends the product to the "Reduce" actor which accumulates the received data and forwards the result to actors "AGather1" as well as "AGather2" to form a two-dimensional array. The allocator analyzes the PSG and determines that the lowest level consists of a vector operation which can be easily assigned to a single processor for execution. However, it should be noted that for performance improvement reasons, the allocator *optimizes* the allocation of actors. Simple vectorizing compiler techniques can be applied to optimize the mapping of this application on our hybrid multiprocessor system.

```
type OneDim = array[ integer ];
type TwoDim = array[ OneDim ];

function MatMult( A, B: TwoDim ; M,N,L: integer
  returns          TwoDim)

for i in 1,M Cross j in 1,L
  S :=
    for K in 1,N
      R := A[ I,K ] * B[ K, J ]
    returns value of sum R
    end for
  returns array of S
end for
end function % MatMult
```

(a)

(b)

Fig. 19. (a) Matrix multiplication in SISAL. (b) Matrix multiplication in IF1.

VI. Conclusions

In this paper, we have demonstrated that data-driven principles are particularly well suited to the determination of the schedulability of operations in signal and data processing problems on multiprocessor architectures. The degree at which these principles are applied determines, for a large part, the domain of application of the system. When the various processors in the architecture are organized in a systolic array, the executability of instructions has been determined directly by the designer and the design remains frozen. This means that the application is fixed. However, comparatively high computational throughputs can be obtained from such organizations. When operation scheduling is decided by compiler intervention, systems such as the ESL polycyclic multiprocessor or the Wavefront Array Processor can be designed. These offer more programmability than systolic arrays. They also offer the potential for scaling up without a complex overall redesign. However, complex data-dependent operations cannot be easily implemented on these architectures.

In order to deliver high computational throughputs through parallelism detection, the data-flow approach has been demonstrated. While conventional explicit parallelism approaches can sometimes show high performance, they require a focused effort on the part of the programmer in order to understand and describe the parallelism of the programmed problem. On the other hand, the functional programming approach allows the implicit detection of parallelism at runtime. At the same time, only a fraction of the programming effort needs to be expended. This shows that one of the main advantages brought by a data-flow architecture is its *programmability* which, in turn, translates into *higher performance for a given amount of programming effort*. In addition, the approach is completely scalable and the configuration of the multiprocessor systems can be adapted to the size of the application. On the other hand, the runtime scheduling of instructions imposes overhead on regular operations and lowers the expectable performance. This expected loss of performance has been traced to the high level of resolution (small granularity) which has been adopted by many data-flow projects. In a signal processing application, the regularity of the low level of processing makes it more appropriate to design a system with multiple levels of resolution. Indeed, we have demonstrated here an architecture with two hierarchy constructs. The lowest consists in a layer of vector processors while the highest provides a true data-flow approach. Future research will study how multiple layers could even include systolic arrays as leaf processors for dedicated applications. These would then be included into multiple hierarchy systems.

In summary, it can be said that the data-driven principles of execution are a necessity in the design of multiprocessor systems, be they incorporated at design, compile, or runtime. The granularity of the scheduling model often presents a tradeoff between delivering maximum amounts of parallelism and reducing communication costs.

ACKNOWLEDGMENT

REFERENCES

[1] W. B. Ackerman, "Data-flow languages," in *Proc. 1979 Nat. Computer Conf.* (New York, NY, June 4–7), vol. 48. Arlington, VA: AFIPS Press, 1979, pp. 1087–1095.

[2] D. A. Adams, "A computation model with data flow sequencing," Tech. Rep. CS117, Comput. Sci. Dep., Stanford Univ., Stanford, CA, Dec. 1968.

[3] J. Allen, "Computer architecture for digital signal processing," *Proc. IEEE*, vol. 73, no. 5, pp. 852–873, May 1985.

[4] Arvind and K. P. Gostelow, "The U-interpreter," *IEEE Computer*, vol. 15, no. 2, pp. 42–48, Feb. 1982.

[5] Arvind and R. A. Iannucci, "Two fundamental issues in multiprocessing: The dataflow solutions," MIT Laboratory for Computer Science, Tech. Rep. MIT/LCS/TM-241, Sept. 1983.

[6] Arvind, V. Kathil, and K. Pingali, "A processing element for a large multiprocessor dataflow machine," in *Proc. Int. Conf. on Circuits and Computers* (New York, NY, Oct. 1980). New York, NY: IEEE, 1980.

[7] Arvind and R. E. Thomas, "I-Structures: An efficient data type for functional languages," Rep. LCS/TM-178, Lab. for Computer Science, MIT, June 1980.

[8] E. A. Ashcroft and W. W. Wadge, "LUCID, A nonprocedural language with iteration," *Commun. ACM*, vol. 20, no. 7, pp. 519–526, July 1977.

[9] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978.

[10] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe, "A theory of communicating sequential processes," *J. Assoc. Comput. Mach.*, vol. 31, no. 3, pp. 560–599, 1984.

[11] D. D. Chamberlin, "Parallel implementation of a single-assignment language," Ph.D. dissertation, Stanford Univ., Computer Science Dept., 1971.

[12] M. Chase, "A pipelined data flow architecture for digital signal processing: The NEC μPD7281," in *IEEE Workshop on Signal Processing*, Nov. 1984.

[13] J. B. Dennis, "Data-flow supercomputers," *Computer*, pp. 48–56, Nov. 1980.

[14] ——, "First version of a data flow procedure language," in *Programming Symp.: Proc. Colloque sur la Programmation* (Paris, France, Apr. 1974), B. Robinet, Ed., *Lecture Notes in Computer Science*, vol. 19. New York, NY: Springer-Verlag, 1974, pp. 362–376.

[15] J. B. Dennis and K.-S. Weng, "An abstract implementation for concurrent computation with streams," in *Proc. 1979 Int. Conf. on Parallel Processing*, pp. 35–45, Aug. 1979.

[16] A. Di Cenzo, "Synthetic aperture radar and digital processing: An introduction," *JPL Publication 80-90*, Feb. 15, 1981.

[17] D. D. Gajski, D. A. Padua, D. J. Kuck, and R. H. Kuhn, "A second opinion on data-flow machines and languages," *IEEE Computer*, vol. 15, pp. 58–69, Feb. 1982.

[18] J. L. Gaudiot, "Structure handling in data-flow systems," *IEEE Trans. Comput.*, vol. C-35, pp. 489–502, June 1986.

[19] J. L. Gaudiot, M. Dubois, L. T. Lee, and N. Tohme, "The TX16: A highly programmable multi-microprocessor architecture," *IEEE Micro*, vol. 6, pp. 18–31, Oct. 1986.

[20] J. L. Gaudiot and M. D Ercegovac, "Performance evaluation of a simulated data-flow multicomputer with low resolution actors," *J. Parallel and Distributed Comput.* New York, NY: Academic Press, Dec. 1985.

[21] J. L. Gaudiot, R. W. Vedder, G. K. Tucker, D. Finn, and M. L. Campbell, "A distributed VLSI architecture for efficient signal and data processing," *IEEE Trans. Comput.*, vol. C-34, pp. 1072–1087, Dec. 1985.

[22] K. P. Gostelow and R. E. Thomas, "A view of dataflow," in *Proc. Nat. Computer Conf.* (New York, NY, June 4–7), vol. 48. Arlington, VA: AFIPS Press, 1979, pp. 629–636.

[23] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester data-flow computer," *Commun. ACM*, vol. 28, no. 1, pp. 34–52, Jan. 1985.

[24] I. Hartimo, K. Kronlof, O. Simula, and J. Skytta, "DFSP: A data flow signal processor," *IEEE Trans. Comput.*, vol. C-35, no. 1, pp. 23–33, Jan. 1986.

[25] K. Hiraki, T. Shimada, and K. Nishida, "A hardware design of the SIGMA-1—A data flow computer for scientific computations," in *Proc 1984 Int. Conf. on Parallel Processing*, Aug. 1984.

[26] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, Aug. 1978.

[27] R. W. Hockney and C. R. Jesshope, *Parallel Computers*. Bristol, UK: Adam Hilger, Ltd., 1981.

[28] E. B. Hogenauer, R. F. Newbold, and Y. J. Inn, "DDSP—A data flow computer for signal processing," in *Proc. 1982 Int. Conf. on Parallel Processing*, Aug. 1982.

[29] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York, NY: McGraw-Hill, 1984.

[30] L. H. Jamieson and E. A. Ashcroft, "Performance analysis of dataflow signal processing algorithms," in *Proc. 1986 Int. Conf. on Parallel Processing* (St. Charles, IL, Aug. 1986), pp. 608–610.

[31] D. Johnson et al., "Automatic partitioning of programs in multiprocessor systems," in *Proc. IEEE COMPCON 80* (IEEE, New York, Feb. 1980).

[32] H. T. Kung, "Why systolic architectures?" *IEEE Computer*, vol. 15, pp. 37–46, Jan. 1982.

[33] S. Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. Bhaskar Rao, "Wavefront array processor: Language, architecture and applications," *IEEE Trans. Comput.*, vol. C-31, no. 11, pp. 1054–1066, Nov. 1982.

[34] S. Y. Kung, R. E. Owen, and J. G. Nash, eds., *VLSI Signal Processing II.* New York, NY: IEEE PRESS, 1986.

[35] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C-36, no. 1, pp. 24–35, Jan. 1987.

[36] P. Le Guernic, A. Benveniste, P. Bournai and T. Gautier, "SIGNAL—A data flow-oriented language for signal processing," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-34, pp. 362–374, April 1986.

[37] J. R. McGraw, "Data-flow computing: The VAL language," *ACM Trans. Programming Languages and Systems*, vol. 4, pp. 44–82, 1982.

[38] J. R. McGraw and S. K. Skedzielewski, "SISAL: Streams and iteration in a single assignment language, language reference manual, version 1.2," Lawrence Livermore Nat. Lab. Tech. Rep. TR M-146, Mar. 1985.

[39] D. C. Munson, J. D. O'Brien, and W. K. Jenkins, "A tomographic formulation of spotlight-mode synthetic aperture radar," *Proc. IEEE*, vol. 71, no. 8, pp. 917–925, Aug. 1981.

[40] J. H. Patel and E. H. Davidson, "Improving the throughput of a pipeline by insertion of delays," in *Proc. 3rd Ann. Symp. on Computer Architecture*, pp. 159–164, Jan. 1976.

[41] B. R. Rau, C. D. Glaeser, and R. L. Picard, "Efficient code generation for horizontal architectures: compiler techniques and architectural support," in *Proc. 9th Int. Symp. on Computer Architecture*, June 1982.

[42] J. E. Requa and J. R. McGraw, "The piecewise data flow architecture: Architectural concepts," *IEEE Trans. Comput.*, vol. C-32, pp. 425–438, May 1983.

[43] J. E. Rodriguez, "A graph model for parallel computation," TR ESL-R-398, MAC-TR-64, Lab. for Computer Science, MIT, Sept. 1969.

[44] D. A. Schwartz and T. P. Barnwell III, "Cyclo-static solutions: Optimal multiprocessor realizations of recursive algorithms," in *VLSI Signal Processing II.* New York, NY: IEEE PRESS, 1986.

[45] H. W. Sorenson, "Kalman filtering techniques," in *Advances in Control Systems*, C. T. Leondes, Ed., vol. 3, pp. 219–264, 1966.

Jean-Luc Gaudiot (Member, IEEE) was born in Nancy, France, in 1954. He received the Diplôme d'Ingénieur from the Ecole Supérieure d'Ingénieurs en Electrotechnique et Electronique, Paris, France, in 1976, and the M.Sc. and Ph.D. degrees in computer science from the University of California, Los Angeles, in 1977 and 1982, respectively.

His experience includes microprocessor systems design at Teledyne Controls, Santa Monica, CA (1979–1980) and research in innovative architectures for the TRW Technology Research Center, El Segundo, CA (1980–1982). Since graduating in 1982, he has been on the faculty of the Department of Electrical Engineering-Systems, University of Southern California, where he is currently an Assistant Professor. His research interests include data-flow architectures, fault-tolerant multiprocessors, and parallel logic programming systems. In addition to his academic duties, he has consulted for several aerospace companies in the Southern California area.

Dr. Gaudiot is a member of ACM.

Attachment 3B

| IEEE.org | IEEE *Xplore* Digital Library | IEEE-SA | IEEE Spectrum | More Sites | Cart (0) | Create Account | Personal Sign In

Browse   My Settings   Get Help

Back to Results

Data-driven multicomputers in digital signal processing

View Document

7
Paper
Citations

1
Patent
Citation

40
Full
Text Views

Related Articles

Improved force-directed scheduling in high-throughput digital signal processing

VLSI system compiler for digital signal processing: modularization and synchroni...

View All

1
Author(s) J.-L. Gaudiot

View All Authors

Abstract Authors Figures References Citations Keywords Metrics Media

Abstract:

New technologies of integration allow the design of powerful systems which may include several thousands of elementary processors. These multiprocessors may be used for a range of applications in signal and data processing. However, assuring the proper interaction of a large number of processors and the ultimate safe execution of the user programs presents a crucial scheduling problem. The scheduling of operations upon the availability of their operands has been termed the data-driven mode of execution and offers an elegant solution to the issue. This approach is described in this paper and several architectures which have been proposed or implemented (systolic arrays, data-flow machines, etc.) are examined in detail. The problems associated with data-driven execution are also studied. A multi-level approach to high-speed digital signal processing is then evaluated.

Download PDF

Download Citation

View References

Email

Print

Request Permissions

Alerts

Authors

References

Citations

Keywords

Related Articles

Back to Top

IEEE Account

» Change Username/Password

» Update Address

Purchase Details

» Payment Options

» Order History

» View Purchased Documents

Profile Information

» Communications Preferences

» Profession and Education

» Technical Interests

Need Help?

» US & Canada: +1 800 678 4333

» Worldwide: +1 732 981 0060

» Contact & Support

About IEEE Xplore | Contact Us | Help | Accessibility | Terms of Use | Nondiscrimination Policy | Sitemap | Privacy & Opting Out of Cookies

IEEE websites place cookies on your device to give you the best user experience. By using our websites, you agree to the placement of these cookies. To learn more, read our Privacy Policy.

Accept & Close

Data-Driven Multicomputers in Digital Signal Processing

JEAN-LUC GAUDIOT, MEMBER, IEEE

New technologies of integration allow the design of powerful systems which may include several thousands of elementary processors. These multiprocessors may be used for a range of applications in signal and data processing. However, assuring the proper interaction of a large number of processors and the ultimate safe execution of the user programs presents a crucial scheduling problem. The scheduling of operations upon the availability of their operands has been termed the data-driven mode of execution and offers an elegant solution to the issue. This approach is described in this paper and several architectures which have been proposed or implemented (systolic arrays, data-flow machines, etc.) are examined in detail. The problems associated with data-driven execution are also studied. A multi-level approach to high-speed digital signal processing is then evaluated.

I. INTRODUCTION

If we are to approach the computational throughputs equivalent to billions of instructions per second which will be required from the processing systems of the future, improvements on all levels of computer design must be made. Faster technology and better packaging methods can be applied to raise clock rates. However, a one billion instructions per second machine would require a clock period as low as a nanosecond. This approach is inevitably bounded by physical limits such as the speed of light. Therefore, instead of considering the *technological approach* to performance improvement, we emphasize here the *architectural method*. Indeed, instead of merely increasing the clock frequency for a corresponding increase in overall throughput, performance can also be improved by allowing multiple processing elements to collaborate on the same program. This inevitably introduces synchronization problems, and issues of resource allocation and sharing must be solved. Programmability is indeed the central problem. In one solution, a conventional language such as Fortran is used to program the application. A sophisticated compiler is relied upon to partition a sequential program for execution on a multiprocessor. This approach has the

advantage of imposing no "software retooling." However, complex numerical applications will not be easily partitioned and much potential parallelism may remain undetected by the compiler.

Ada, CSP [26], extended Fortran (e.g., HEP, Sequent), on the other hand, allow the programmer to deal with parallel processes by the use of primitives for parallel task spawning, synchronization, and message passing. However, while the programmer can express some of the parallelism characteristic of the application, much potential concurrency may never be uncovered because of the inherent sequential concepts of the language which must be countered through the use of special "parallelism spawning" instructions. Also, development time becomes important since the programmer must "juggle" with many parallel tasks to synchronize. In addition, debugging becomes correspondingly more difficult due to the sometimes undeterministic appearance of errors.

For these reasons, an *implicit* approach must be devised. In the above two methods, instruction scheduling is based upon a central program counter. We propose to demonstrate here the data-driven approach to programming multiprocessors: instructions can be scheduled by the *availability of their operands*. This model of execution is a subset of the *functional* model of execution [9]. It provides a significant improvement to the programmability of multiprocessors by excluding the notion of global state and introducing the notion of values *applied* to functions instead of instructions *fetching* the contents of memory cells as they are in the conventional "control-flow" model.

The overall objective of this paper is to demonstrate the applicability of data-driven principles of execution to the design of high-performance signal and data processing architectures. Several approaches will be demonstrated and their particular domain of application will be contrasted. The description of low-level processing systems is beyond the scope of this paper and the interested reader is referred to an excellent survey by Allen [3]. Instead, we will concentrate here on the issues related to building high-performance multiprocessors for signal processing applications. In Section II, we show the type of problems considered in signal processing. The data-flow principles of execution as they relate to digital signal processing problems are described in detail in Section III while several exist-

ing data-driven architectures are described in Section IV. In Section V, we analyze a multi-level data-driven architecture and examine its programming environment. Conclusions are drawn in Section VI.

II. THE REQUIREMENTS OF SIGNAL PROCESSING

Digital signal processing techniques are applied to many different technical problems. These include radar and sonar systems, image processing, speech recognition, etc. The elementary building blocks of these were originally concentrated on such tasks as convolution, correlation, and Fourier transform. More complex algorithms (matrix operations, linear systems solvers, etc.) are now considered. Higher order operations include not only simple problems such as elementary filtering (IIR, FIR, etc.), but also more complex functions such as adaptive and Kalman filtering [45]. Also, such complex problems as Computer-Aided Tomography or Synthetic Aperture Radar can be considered [39], [16]. Signal processing algorithms are very appropriate for description by functional languages. Indeed, a signal processing algorithm is often represented in a graph form [36] and can be decomposed in two levels:

* a regular level which can be implemented by a *vector operation* (i.e., a loop in which all iterations present no dependencies among themselves);
* a level which contains conditional operations and heuristic decision making.

This description shows that the lower operational levels can easily deliver parallelism (by compiler analysis or programmer inspection). This layer usually consists of simple constructs (arithmetic instructions, FFT butterfly networks, simple filters, etc.). However, the higher levels will require more complex problem insight and even runtime dependency detection in order to allow maximum parallelism. We will now describe principles of execution which will allow us to deliver this concurrency.

III. DATA-FLOW PRINCIPLES

The data-flow solution to the programmability problems of large-scale multiprocessors [5] has been pioneered by Adams [2], Chamberlin [11], and Rodriguez [43]. It is now described in detail in this section.

A. Basic Principles of Execution

In the conventional von Neumann model of execution, an instruction is declared executable when a Program Counter of the machine points to it. This event is usually under direct programmer control. While a control-flow program is a sequential listing of instructions, a data-flow program can be represented as a graph where the nodes are the instructions (*actors*) which communicate with other nodes over *arcs* (Fig. 1). An instruction is declared executable when it has all its operands. In the graph representation chosen above, this means that all the input arcs to an actor must carry data values (referred to as *tokens*) before this actor can be executed. Execution proceeds by first absorbing the input tokens, processing the input values according to the op. code of the actor, and accordingly producing result tokens on the output arcs. In summary, it can

Fig. 1. A simple data-flow graph.

be said that the data-flow model of execution obeys two fundamental principles:

* *Asynchrony of operations:* The executability of an instruction is decided by a *local criterion* only. The presence of the operands can be sensed "locally" by each instruction. This is an attractive property for an implementation in a distributed environment where no central controller should be used for global scheduling.
* *Functionality of the operations:* The effect of each operation is limited to the production of results to be consumed by a specific number of other actors. This precludes the existence of "side-effects." These side-effects may be long-ranging in that the execution of an instruction may effect the state of a cell of memory which will be used only much later by another unrelated operation.

B. Data-Flow Interpreters

When iterations are executed, the underlying principle of data-flow (*single assignment of variables*) must invariably be violated. Indeed, for an actor to be repeatedly evaluated as in an iteration, its input arcs must carry several tokens (from different iterations). Several solutions have been proposed to allow the *controlled violation* of these rules without compromising the safe execution of the program. Among these, the Acknowledgment scheme and the U-interpreter have been given the most consideration.

1) Acknowledgment Scheme [14]: Proper matching of the tokens can be observed by ordering the token production. This would be done by a careful design of the program graph so as to insure that tokens of two different iterations can never overtake each other. In addition, it must be guaranteed that *no* token pileup is encountered on *any* one arc. This condition can be verified by allowing the firing of an actor when tokens are on all input arcs *and* there are no tokens on any output arcs. In order to enforce this last condition, an *acknowledgment* must be sent by the successor(s) to the predecessor when the token has been consumed (Fig. 2). Note that an actor is executable when it has received its input arguments as well as all acknowledgments. The parallelism which can be exploited from this scheme is mostly pipelining between the actors of different iterations. Thus when the number of instructions in the body of an iteration is the same as the number of available processors, the speedup observed by this mechanism of execution is maximal. However, for small iterations (compared to the size of the machine), the exploited parallelism

Fig. 2. The acknowledgment scheme.

Fig. 3. A typical iterative construct in the U-interpreter.

falls below its potential. Thus it may be required that the compiler effect a *code expansion* for vector operations.

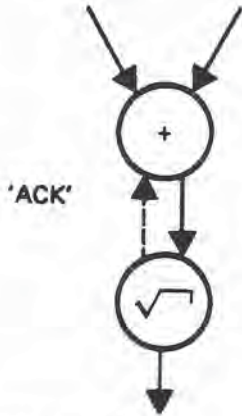An important characteristic of this *static* model of execution is the fact that it allows only one instance of an instruction to exist at one given time. In other words, it is primarily relied upon *pipelining* for the exploitation of parallelism in iterations. However, the basic acknowledgment scheme does not allow the implementation of multiple simultaneous calls to the same function. Several machines which obey these principles of execution have been designed: the MIT cell block architecture [13], the Hughes Data-Flow Machine [21], the DSFP [24], the USC TX16 [19], etc.

2) The Unraveling Interpreter (U-Interpreter): The U-interpreter [4] provides the most asynchronous possible operation. In order to allow safe execution of actors in an iterative construct, tokens are tagged with information pertaining to their context of creation. An actor is only allowed to execute when an input token pair with matching tags can be found. This tag includes the iteration number. Indeed, the U-interpreter closely follows these principles: to each data token is attached a tag of the form $u.P.s.i$, where P identifies the procedure name of the destination actor, while s is the address of this actor within procedure P. The i field corresponds to the iteration number in which the token was created, while the u field is the context of creation. Note that while the former is used to distinguish between tokens destined to different iterations of the same actor, the latter is used in situations involving multiple function calls, operations with recursive function calls, or nested iterations.

Special actors are used which deal with the context and iteration fields of the token tags. A typical iteration construct in the U-interpreter is shown in Fig. 3. The D actor is used to recirculate the data from one iteration to the next. Its input is tagged with $u.P.t.i$ while its output value is identical but has become tagged with $u.P.t'.i + 1$. Nested iterations are handled by *isolating* the inner from the outer iteration by the introduction of the L actor at the top of the graph. The function of this actor is to *create a new context* for the execution of the iteration: the input tokens are tagged by $u.P.s.i$ while the output tokens are identical but are tagged with $u'.P.t'.1$ where u' is itself $u.i$. Note that this mechanism is sufficient to create an entirely different set of tokens for two nested iterations. Indeed, assume that two tokens both belong to the same inner iteration i, but belong to the j_1 and j_2 outer iterations respectively. The first token

would be tagged $u1.P.s.i$ ($u1 = u.j_1$) while the second is tagged with $u2.P.s.i$ ($u2 = u.j_2$). This shows that an appropriate differentiation has been made between the two instances. The original context u is retrieved by the L^{-1} actor before exiting.

Contrarily to the acknowledgment scheme, this *dynamic* data-flow scheme allows full asynchronous execution of the program graph. Indeed, due to the scheme of tags, several *instances of the same instruction may exist simultaneously*. Vector operations may be executed in parallel without compiler-induced replication of the graph. Likewise, multiple function calls and more particularly recursions are allowed since each new actor instantiation receives a different tag. This means that the U-interpreter would be preferred to the static model when the ability for fast recursive calls is required. However, this flexibility comes at the expense of added hardware complexity. Indeed, it will be shown in Section IV-D that implementation of the U-interpreter requires an associative memory for fast tag matching. Several machines based on these principles have been studied: the MIT tagged token data-flow machine [6], the ESL DDSP [28], the University of Manchester machine [23], the ETL Sigma-1 [25], etc.

C. Structure Handling

This is a crucial issue in signal processing for this kind of application requires that many data elements which belong to the same structure be processed in a parallel or pipelined fashion. One of the basic premises of data-flow principles states that an output is a function of its inputs only, regardless of the state of the machine at the time of execution. When a structure of elementary elements must be processed, the absence of side-effects means that it may not be updated for this would imply its transition through several states. Instead, if any updates are needed, a new array which contains the new elements must be created. Copying of all elements must be undertaken for the modification of a single one. This solution imposes an inordinate overhead. This is why the implementation schemes we will now describe can shortcut this complete copying while

preserving the meaning of the program during array update operations.

1) Heaps: Dennis [14] has proposed to represent arrays by directed acyclic graphs (also called *heaps*). Each value is represented as the leaf of a graph tree. The modification of a single element in a heap is represented in Fig. 4. Note

Fig. 4. A heap update.

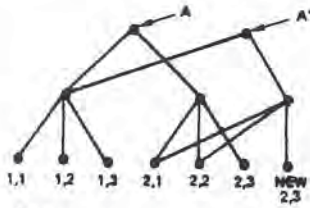that the complexity of the modification of a single element of the array is $O(n)$ for a copy operation, while it is $O(\log n)$ for the heap. Several instructions are exclusively devoted to the access of heaps [15]; SELECT receives a pointer to the root node, an index value, and returns a pointer to the substructure (which may be a leaf node) associated with the index; APPEND also needs the same two operands in addition to the value of the element to append to the structure.

2) I-Structures: A heap must be entirely ready before it can be consumed because no consumption (SELECT actors) can take place until the pointer token appears (i.e., the creation of the array is completed). In the I-structure scheme [7] constraints on the creation of arrays allow the selection of individual elements (or substructures) from the array before its complete production. One possible implementation of I-structures makes use of a "presence" bit which indicates when an element of an array has been calculated and is ready for consumption. An attempt to read an empty cell would cause the read to be *deferred* until such time that the cell presence bit is marked. Conversely, a write into a cell, the presence bit of which indicates valid stored data, could be cause for the generation of an error signal. The advantages of this scheme are:

* better performance because pipelining is allowed between I-structure consumers and producers;
* less "serialization" of operations such as APPENDs, because they are allowed to occur independently on the same structure.

3) HDFM Arrays: A special scheme for handling arrays in a VAL high-level environment has been designed for the Hughes Data-Flow Machine (HDFM) [21]. It uses the fact that data-flow arrays as described above are overly "asynchronous," i.e., they do not take advantage of the data dependency information carried by the program graph. Safety of accesses is respected by not allowing the updating of an array before all the reads from the *current version* of the array have been performed. Only then can the array be directly modified. Safety and correct execution of WRITE operations are a compile-time task. This has the advantage of reducing the number of memory accesses (no complex graph of pointers must be travered as in heaps) as well as of offering a better possibility of distribution of an array (no root node). However, spurious data dependencies may be introduced because the compiler is not necessarily aware

of the possibility of parallelism that can be detected only at runtime. For instance, dependencies on A and B related by $A(F(i)) = B(i)$ may be artificially imposed. However, the applications targetted by the HDFM include some amount of regularity which can be easily detected by the compiler and implemented as conventional arrays.

4) Token Relabeling [18]: In the U-Interpreter, the notion of array can be entirely ignored at the lowest level of execution. Instead, the *tag* associated with each token under the rules of the U-interpretation is used as identification of the index of the array element of the high-level language. In other words, it can be simply said that, when an array A is created, its $A(i)$ element will be tagged with i (hereafter denoted $A(i)_{[i]}$), if the elements are produced in the logical order. In the "production" of a unidimensional array, the iteration number can usually be directly interpreted as the *index* of the array element just produced by the iteration.

Special token relabeling program graphs can be created to handle *scatter* and *gather* program constructs [27] (Fig. 5(a)). This figure shows that an inversion function F^{-1}

Fig. 5. (a) A gather operation. (b) Token Relabeling gather.

This demonstrates that, without recourse to the calculation of F^{-1}, the proper relabeling of the A elements has been effectively produced.

This algorithm requires no intermediary storage, does not need array operations, and imposes smaller hardware and execution overhead. This relabeling approach eliminates a large portion of the overhead associated with the production and consumption of array A. Pipelining between the source and the sink of a data structure is the goal of this unknown at compile time would be needed to perform the relabeling of data-flow tokens. Such a calculation is not truly necessary. Instead, we introduce (Fig. 5(b)) a sequence generator which produces the $F(j)$'s, tagged by j. An *exchanger* actor (called χ) swaps the tag and the data value and produces $j_{[F(j)]}$. Both streams (the A's and $j_{[F(j)]}$) are input to a special relabeling actor δ which only modifies the iteration portion of the tag. By the principles of the U-interpreter, only tokens which bear the same tag will be matched as proper inputs to the δ actor. In other words, the mate of

token $j_{[F(j)]}$ is the element $A(F(j))_{[F(j)]}$. The special actor δ is a relabeling actor which takes the A input and relabels it with the data carried by the token on the other input arc. In other words, it outputs $A(F(i))_{[i]}$. Since i is a dummy variable, and since F is bijective, it can be said that, on a global point of view:

$$A(F(i))_{[i]} \Leftrightarrow A(k)_{[F^{-1}(k)]}.$$

scheme, just as it was the idea behind the design of the I-structures. However, the token relabeling approach brings a better runtime memory management since tokens corresponding to the various elements of the array still exist and must still be temporarily stored, they need not go through an additional storage as a data structure. Also, there is no need for "requests" for data as would be the case in an I-structure environment: When an I-structure is created, actors which need data from it must demand the element in question until it has arrived. This may introduce heavy overheads as unsatisfied requests must be queued in the structure itself. Garbage collection is automatically handled since when the "array token" is matched, it is automatically removed from the arc. In other words, when it has been used, it is swallowed by applying data-flow principles.

D. High-Level Data-Flow Languages

In addition to the low-level mechanisms of execution which were described earlier, special high-level data-flow languages have been designed for easier translation into data-flow graphs. To be sure, these high-level languages are not a necessity: the Texas Instruments data-flow project [31] relied upon Fortran programming through the use of a modified version of a vectorizing compiler originally destined to the TI ASC. However, many high-level languages have been designed for data-flow prototypes. Most notable are VAL (Value Algorithmic Languages) for the MIT static data-flow project [37], [1], Id (Irvine Dataflow) for the MIT tagged token data-flow architecture [4], LUCID [8], [30], etc. SISAL (Streams and Iterations in Single Assignment Language) has been designed by McGraw and Skedzielewski [38] and is intended as the definition of a "universal" language for the programming of future multiprocessors.

Data-flow languages have also been defined for the specific purpose of programming signal processing applications. These include the SIGNAL language designed by Le Guernic et al. [36]. The intent of the language is to provide a formal specification of signal processing problems and to ease the design of signal processing multiprocessors, be they special- or general-purpose. One of the main characteristics of the language is that it incorporates the notion of *time* to describe the interaction of the various processing tasks. This makes it a *synchronous* language as opposed to asynchronous languages such as CSP and Occam [10]. SDF (Synchronous Data Flow) is another formal description of signal processing algorithms based on data-driven principles of execution proposed by Lee and Messerschmitt [35].

IV. Data-Driven Architectures

We now describe in detail several systems which operate at runtime, compile-time, or design-time under data-driven execution. Although it is generally considered that data-flow principles of execution are in effect at runtime, we extend their domain of application to design or compile time and refer to them as *data-driven systems*. We thus initially examine multiprocessor systems where data dependencies have been frozen at design time (systolic arrays). We then consider programmable systolic arrays (the Wavefront Array Processor) and multiprocessors scheduled at compile time by the use of data-flow program graphs (the ESL polycyclic processor). Finally, we study systems where the data dependencies provide scheduling information at runtime (the Hughes Data-Flow Machine) and examine the influence of the level of resolution upon the performance (the USC TX16).

A. Systolic Arrays [32]

The primary goal of a systolic array is to make use of the large amount of processing power available in VLSI technology by using repetitive circuitry to perform signal processing problems, matrix operations, image processing, etc. In summary, a systolic array is simply a collection of interconnected Processing Elements (PEs). In order to incorporate as many processors as possible, the structure of the PEs themselves is kept to a maximum simplicity and usually includes only a few operation units. For design simplification, there are few types of PEs in the same system. By the same token, interconnections are kept to a nearest neighbor topology in order to minimize communication delays as well as power distribution issues. Note that topologies include two neighbors (linear arrays), four neighbors (square arrays), or six neighbors (hexagonal arrays) as required by the problem to solve. This is notably due to the fact that scheduling mechanisms must be based upon *local criteria* such as data availability. However, it should be noted that there is a global clock in all the computation cells. Linear systolic arrays can tolerate clock skews at both ends, but multidimensional designs require slower clocks in order to compensate. In order to simplify runtime mechanisms, the design of a systolic array emphasizes an efficient mapping of the problem onto the architecture. An example of a band matrix–vector multiplication is shown in Fig. 6. It displays
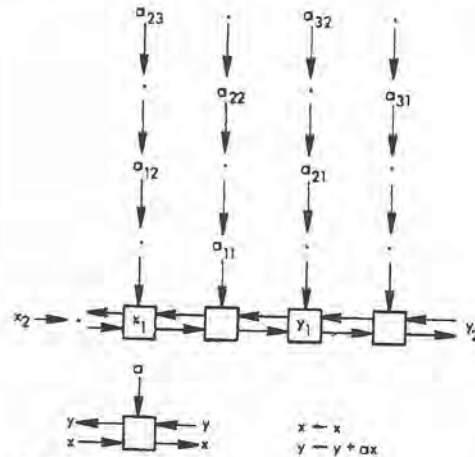

Fig. 6. A linear systolic array.

how the synchronization of the processors and of the input data rate has been mapped to meet the requirements of the problem. Note that each processor is designed to operate upon *the arrival of the arguments*. In summary, it should be

1224

noted that systolic arrays are very efficient at computationally intensive problems which involve many repetitive low-level calculations. Also, the very nature of their design renders their function fixed at design time.

B. The Wavefront Array Processor (WAP) [33], [34]

Execution on the WAP is similar at runtime to the execution of a program on a systolic array. Indeed, both approaches rely upon the scheduling of operations based on the availability of their operands. However, the analysis of the data dependencies is effected during the design of a systolic array while the WAP is scheduled by compiler detection of parallelism: the WAP is a "programmable systolic array." It has been shown that most signal and data processing algorithms possess a certain amount of *locality* and *recursivity*. They will thus exhibit the phenomenon of *computational wavefront*. This has an important implication in that an entire *front of processors* can be programmed for the same operation. In addition, it can be shown that two successive wavefronts cannot intersect. This enables the proper implementation of data-driven principles of execution. For instance, a matrix multiplication can be executed as a computational wavefront (Fig. 7). A special
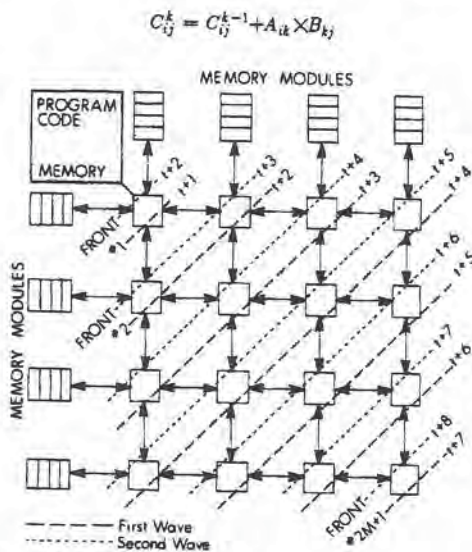
$$C_{ij}^k = C_{ij}^{k-1} + A_{ik} \times B_{kj}$$

Fig. 7. Matrix multiply in the WAP.

language called the Matrix Data-Flow Language (MDFL) has been designed to express such algorithms on the WAP.

C. ESL Polycyclic Architecture [41]

The ESL polycyclic architecture is a horizontally microprogrammed multifunctional vector processor. It comprises several functional units (adders, multipliers, storage units) connected by a cross-bar interconnection network. Entire vector loops (no data dependencies across the iterations) can be scheduled by using the model presented in [40]. The essential idea is to discourage "greedy" scheduling by insertion of "non-compute" delays in the train of calculations. The effect of these delays is to enable an optimal schedule. A pipeline is viewed as a certain number of resources (the various segments of the pipe) which can be reserved by tasks. The problem is reduced to the produc-
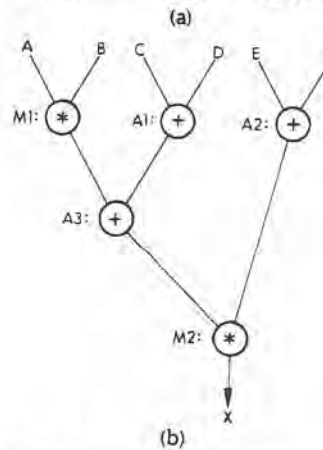
tion of a reservation table with no collisions (i.e., no two tasks can reserve the same segment of the pipeline at the same time). For that purpose, a *usage interval* is defined as the time interval between two reservations of a segment by a single task. Two tasks will collide when they have the same initiation time as one of the usage intervals. For a homogeneous multiprocessor (identical PEs), the method is done in two steps:

1) Determine the Minimum Initiation Interval MII as MII = $[N/P]$. N is the number of instructions in the body of the loop, and P is the number of processors available for execution. The initiation interval is the length of time between the initiation of two consecutive iterations. Successive iterations will be scheduled at MII units interval. All the iterations will be identically scheduled.

2) Schedule the operations in accordance with the data dependences. However, no more than P operations may be scheduled for the same time modulo MII. Note that this last constraint also implies that delays must be inserted in the schedule.

The following example shows the scheduling of a simple vector operation (Fig. 8(a)) on a polycyclic processor with two adders and one multiplier (note that for simplification, communication costs have been assumed to be null). There are two multiply operations for a single multiplier while there are three additions on two adders. The MII would therefore be 2. This means that one iteration of the loop can be performed at a rate of one for every two cycles. By using the data dependency graph of the example (Fig. 8(b)), the optimal schedule can be used by applying the MII of 2 (Fig. 8(c)). Proper "dovetailing" of successive iterations is assured

$$DO\ 1\ I = 1, N$$
$$1 \quad X(I) = \left[\left[A(I)*B(I)\right] + \left[C(I) + D(I)\right]\right] \cdot \left[E(I) + F(I)\right]$$

(a)

(b)

Time	Multiplier	Adder #1	Adder #2
0	M1	A1	A2
1	.	A3	
2	.	.	.
3	M2	.	.

(c)

Time Mod. MII	Multiplier	Adder #1	Adder #2
0	M1(1)	A1(1)	A2(1)
1		A3(1)	
0	M1(2)	A1(2)	A2(2)
1	M2(1)	A3(2)	
0	M1(3)	A1(3)	A2(3)
1	M2(2)	A3(3)	...
0	...	...	...

(d)

Fig. 8. (a) A simple vector operation. (b) Corresponding data dependency graph. (c) Scheduling a single iteration. (d) Dovetailing iterations.

by this scheduling algorithm when the iterations are processed at the rate of one for every two cycles (Fig. 8(d)). This architecture applies particularly well to signal processing applications where the same computation must be repetitively applied to a different element in a steady data stream.

For example, a butterfly block in an n-point FFT operation would be executed $n \times \log n$ times. However, in addition to the purely computational actors shown above, "store" and "retrieve" operations should also be considered. This is demonstrated in the data-flow graph of Fig. 9(a) which
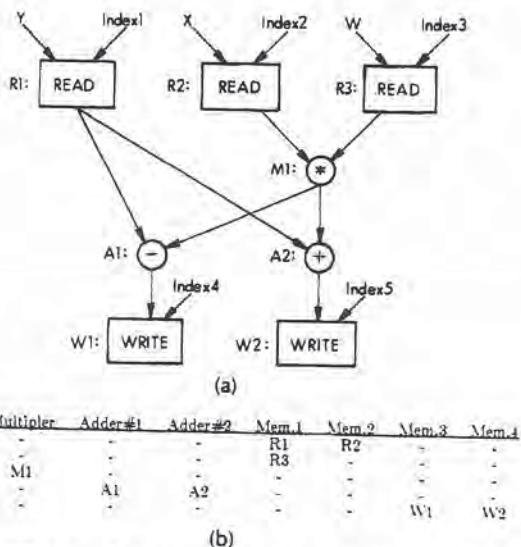

(a)

Time	Multiplier	Adder#1	Adder#2	Mem.1	Mem.2	Mem.3	Mem.4
0	-	-	-	R1	R2	-	-
1	-	-	-	R3	-	-	-
2	M1	-	-	-	-	-	-
3	-	A1	A2	-	-	-	-
4	-	-	-	-	-	W1	W2

(b)

Fig. 9. (a) FFT butterfly block. (b) Scheduling of an FFT butterfly.

corresponds to a single iteration (butterfly block) of a real FFT. Note that the indexes (for reads and writes) have been assumed to be generated elsewhere (e.g., table look-up) and are ignored in this discussion for simplification. Assuming that two adders and one multiplier can be used, and that we have four memory modules at our disposal, the MII can be determined as the *maximum* of the N/P ratio for each kind of resource. This yields an MII of 2 (Fig. 9(b)). Note that further work in the scheduling of iterations has been carried out in [44]. This research is also applied to SSIMD architectures and allows the existence of dependencies between iterations.

D. The MIT Tagged Token Data-Flow Machine [6]

This machine implements a version of the U-Interpreter. In this distributed architecture model, each PE is independent from its neighbor and there is no global controller. A hypercube communication network allows the transmission of data-flow tokens between PEs. Store-and-forward capabilities are provided so that a pair of PEs which is not directly linked may still communicate.

The structure of each PE is shown in Fig. 10. A switch receives tokens from the network and determines whether the incoming packet is a data token or a structure request to be processed by the I-Structure Memory. In the Matching Store Unit, the tag of the incoming token is associatively checked against that of previously arrived tokens to determine whether it is the first or the second token to arrive at a given instruction. The first token should be stored in the
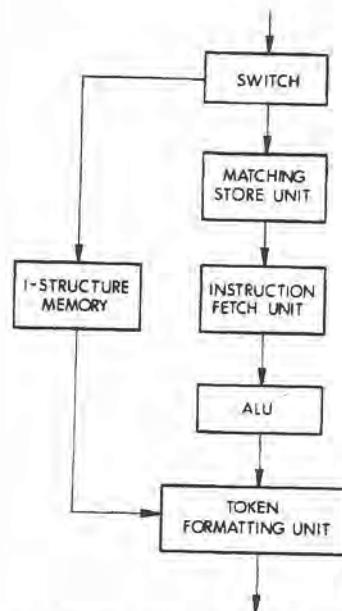

Fig. 10. A PE in the tagged token data-flow machine.

associative memory of the Matching Store Unit and held until its mate arrives. For the second token, the corresponding instance of the instruction can be activated by sending an *argument packet* to the next unit. The Instruction Fetch Unit receives this packet and fetches the parameters of the instruction. Note that the template contains not only the op.code but also pointer(s) to the destination actor(s) to which the result of the operation should be sent. A complete *instruction-ready packet* can be formed and sent for execution to the ALU. The ALU blindly executes the operation indicated by the incoming template and produces result tokens which are received by the Token Formatting Unit. Finally, the Token Formatting Unit receives tokens which have been produced by the ALU. These tokens comprise several fields: the tag associated with the operation (after modification if the operation was a tag-modifying operation), the data themselves, as well as an *allocation function* field. This field is used by the Token Formatting Unit to determine the destination PE of the token. Indeed, this determination cannot often be made solely on the basis of destination actor for this would mean allocating to the same PE all the iterations of an actor in loop. This is clearly unacceptable if parallelism is to be extracted *across* the iterations of the loop. An often used heuristic allocation function is based upon calculation of the iteration number modulo the total number of PEs. This function has the advantage of allowing proper distribution of a loop across the machine. Depending upon existing conditions, different allocation functions may be used within the same graph. However, it must be noted that the function must be the same for the two tokens destined to the same actor. Failing the verification of this condition, the two tokens would never be matched for they would be sent to different PEs. This demonstrates the need to implement this allocation operation at *compile time*.

E. The Hughes Data-Flow Machine (HDFM) [21]

The goal of this project is to provide a high-performance parallel architecture which is highly programmable and at

the same time offers advantages of modularity and simplicity of implementation for signal and data processing applications. All communications are based on the *message passing* model. A maximum of 512 PEs can be organized in a cube network. Each PE is attached to three busses (row, column, and plane). Design of the PEs has been made for easy implementation in VLSI. PEs can easily be added because of the modular nature of the communication network. Traffic on each bus is based upon contention before data can be transmitted. Any two PEs can communicate by a maximum of three "hops" (Fig. 11). The execution model
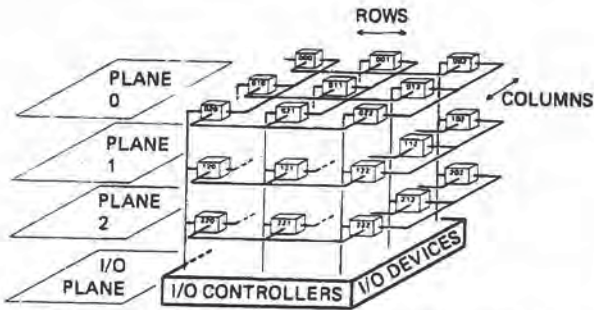

Fig. 11. Structure of the HDFM communication network.

is based upon the acknowledgment scheme. Instead of using "hardwired" acknowledgment arcs between two communicating actors, this machine is based upon the principle of "software" acknowledgments. The compiler partitions the data-flow graph into blocks. Special acknowledgment arcs are introduced between the blocks. Note that this method allows pipelining between iterations at the block level.

One of the primary requirements of the machine was to incorporate as few component parts as possible. This

implies a high level of integration for the individual PEs. Indeed, each PE consists of only two custom-designed chips in addition to several commercially available memory circuits. The overall architecture of a PE is shown in Fig. 12.
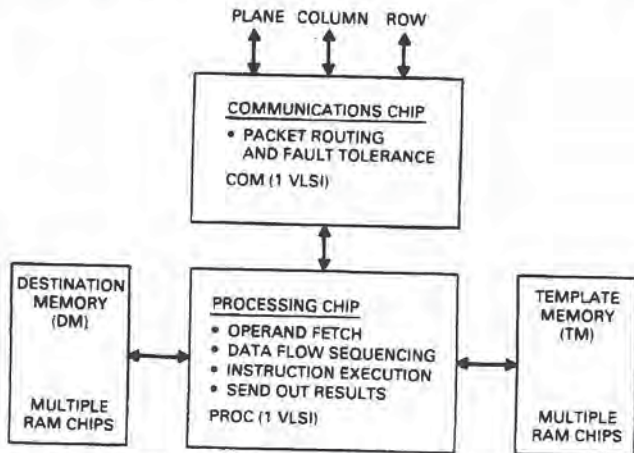

Fig. 12. A PE in the HDFM.

The COM chip handles all the communication functions and interfaces the actual PE with the three-bus communication network. It implements the necessary "store-and-forward" and performs in addition a buffering function in order to even packets rates. Note that the chip pin-outs requirements limit the number of outside busses to 3. The PROC chip is the actual PE which contains three pipelined stages: 1) instruction/operand fetch and data-flow firing rule check, 2) instruction execution, and 3) result token formatting. It can be easily represented schematically (Fig. 13). Tokens arriving from the COM chip are first checked to determine whether they complete an instruction packet or not. Ready instructions are then dispatched to the exe-

MICROMACHINES
COM: COMMUNICATIONS/FAULT TOLERANCE
TMC: TEMPLATE MEMORY CONTROLLED
ALU: ALU/MICROPROCESSOR AND MICROMEMORY
DT: DESTINATION TAGGER
DMC: DESTINATION MEMORY CONTROLLER

MEMORIES
DM: DESTINATION MEMORY
TM: TEMPLATE MEMORY
FDM: FIRE DETECT MEMORY

QUEUES
FCQ: FROM COMMUNICATION QUEUE
FBQ: FEEDBACK QUEUE
FQ: FIRING QUEUE
RQ: RESULT QUEUE
DQ: DESTINATION QUEUE
AIQ: ASSOCIATED INFORMATION QUEUE
TCQ: TO COMMUNICATION QUEUE

Fig. 13. The PROC chip in the HDFM.

cution unit. The results are formatted into tokens by the Destination Tagger and sent to the COM chip or back to the input Token Unit as the case may be. The Template Memory Controller (TMC) enforces the data-flow rules of execution and checks the completion of an input set before it sends a complete instruction packet to the ALU for execution. Results are sent by the ALU to the Destination Tagger (DT). In collaboration with the Destination Memory Controller (DMC), this unit associates the data values produced by the ALU with their proper destination address (which can be found in the Destination Memory DM). Note that in the Cell Block architecture, the templates are wholly stored in one location. In this architecture, the templates are, instead, split in two portions: the op. code and input operand portion stored in the Template Memory TM, while the corresponding result pointers are stored in the Destination Memory. The rationale for this design decision can be found in two points: first, this allows a better space management since the destination list of any template may be of undeterminate length. Second, if the whole template were to be stored in the Template Memory, the Destination pointer information would have to be propagated through the ALU before it could be used only in the last stage of the processor. This would prove particularly inefficient since the whole processor must be integrated on a single chip, thereby multiplying the area necessary for busses. Simulation of radar processing algorithms has demonstrated that each PE capable of a throughput of 2–4 MIPS while a 64 PE could produce throughputs of 64 MIPS.

F. The NEC μPD7281 [12]

The NEC μPD7281 is a single-chip digital signal processor. Its most important application is image processing. Some immediate applications include image restoration, enhancement, compression, and pattern recognition. It is based on a data-flow model of computation and implements such complex operations as multiplication in the basic instruction set. More specifically, its primitives are designed for an efficient execution of image processing algorithms. The use of data-flow principles of execution increases the programmability of the machine and renders

the multiprocessor architecture invisible to the programmer. Another characteristic of the μPD7281 is that it can be *cascaded* with several other identical chips in a *ring* architecture. Indeed, the architecture of the μPD7281 enables the design of multiprocessor systems for improved performance. By cascading several such PEs, a high degree of pipelining can be observed. In addition to the high-level organization of the chips, each chip is itself organized in a ring architecture (Fig. 14) which operates in a pipelined fashion.

Constants may be stored in the Data Memory for storage during execution. The program is represented in both the Function Table and the Link Table. In the Function Table, the actors themselves are stored. Similarly, the Link Table contains a representation of the arcs between the actors. After the initial loading of the PEs, when a token enters a μPD7281, it is first checked to determine whether this PE should process it. If not, it is directly transferred from the Input Controller (IC) to the Output Controller (OC) where it is forwarded to the next processor along the chain. Otherwise, it can be sent to the Link Table for processing. In the Link Table and the Data Memory, it is matched with other tokens before it can be sent for actual processing. The Link Table always contains the first of the two operands that arrive. The Address Generator and Flow Controller are used to generate addresses of constants. Note that after actual processing of a data-flow actor in the Processing Unit, the resulting token is again processed by the Link Table of the same Processing Element. When the ultimate consumer of the token is allocated to another PE a special output instruction is executed in the queue so that the token can be switched to the Output Controller. Overall, this circular pipeline contains seven segments and can deliver a maximum throughput of one instruction per cycle. The primitives of the μPD7281 are oriented towards image processing applications:

- CONVO (Convolve) which can be used to perform cumulative operations such as
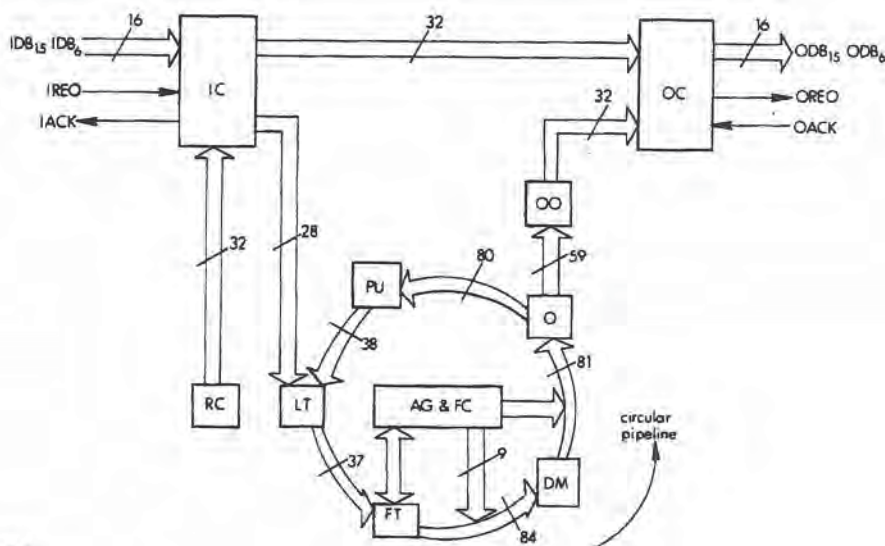
$$\sum_{i=1}^{n} A_i B_i$$
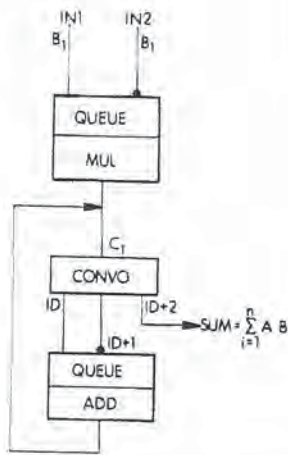

Fig. 14. The NEC μPD7281.

Fig. 15. A convolver actor.

(Fig. 15). Note that this operation is not strictly speaking a data-flow actor in that the summation implies a "state" of the actor. Formally, it would correspond to a "macro-actor" [20] which includes a graph of several elementary data-flow actors.

- ACC (Accumulative Addition Instruction).
- Bit manipulation, data conversion instructions, etc.

The multiplication of a 3×3 matrix by a 3-element vector is illustrated in Fig. 16. It is assumed that the A matrix has

$$\begin{bmatrix} A_1(1) & A_2(1) & A_3(1) \\ A_1(2) & A_2(2) & A_3(2) \\ A_1(3) & A_2(3) & A_3(3) \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix}$$
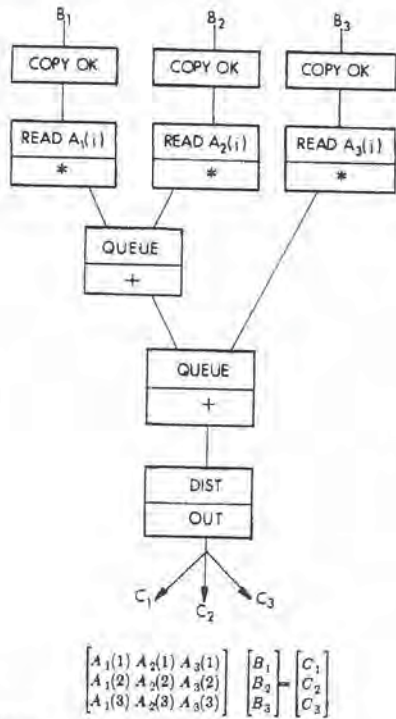
Fig. 16. Matrix vector multiplication.

already been allocated to the data memory. Each element of the vector B is received and is replicated three times by the COPYBK actors. Note that one of the internal parameters of the actor is 3, the number of required replications. The *multiply* actor is coupled with a *readA$_i$(j)* actor so as to perform the multiplication of B_i with $A_i(j)$. The results from the three multiplications are matched and accumu-

lated to produce the result C_j. Note that the queue actors enable a better pipelining of the successive computation waves. They allow an execution similar to the model presented by the Acknowledgment scheme (Section III-B1). However, while in the Acknowledgment scheme only one token is allowed at any time on any single arc, this model of execution allows as many tokens per arc as the size of the queues. Benchmark evaluations have shown a near linear speedup with increasing numbers of chips from 1 to 3:

Algorithm	1 PE	3 PE
512×512 binary image rotation	1.5 ms	0.6 ms
512×512 binary image $\frac{1}{2}$ shrinking	80 ms	30 ms
512×512 binary image smoothing	1.1 s	0.4 s
512×512 binary image 3×3 conv.	3.0 s	1.1 s
64 stage FIR filter (17 bits)	50 μs	18 μs
$\cos(x)$ (33 bits)	40 μs	15 μs

G. The USC TX16 [19]

The TX16 is based upon the Inmos Transputer. The Inmos Transputer has been heralded as the first of a new generation of microprocessors. Indeed, while conventional microprocessors are interfaced with the external world through a single memory bus (address, data, and control), the Transputer possesses in addition four serial communication links. Each of these communication links allows point-to-point transmissions between two Transputers. This architecture is reflected at the language level: the arrival of data on a link will trigger a process inside the receiving Transputer.

The programming language **Occam** allows the presence of several different processes while only one is active at a time. Message transmission with other processes is based upon the synchronous principles of CSP [26], [10]. This means that when the active process must communicate an intermediary result with another process (possibly located in another processor), the active process is held until the other process has been found to be ready for the transmission. While the process is held, it is stacked into the inactive process queue. Another ready process is then activated until it either terminates or is itself hung because of a required transmission. This low-level context change mechanism compares favorably to the busy–wait model found in conventional multiprocessor systems. Instead of idling a processor while waiting for an intermediary operand to arrive, the system allows context switching to another ready process.

The system consists of 16 interconnected Transputers interconnected in an ILLIAC-IV topology. The four links of each Transputer are used for scalar data communications and for interprocess synchronization messages. Each PE owns a single bank of the memory system (Fig. 17). A processor can directly access its own memory bank through the external memory bus of the Transputer (*local access*). A *remote access* can be made into the bank owned by another processor. In this case, the Bus Controller formats the request from the PE into a packet and takes control of the bus. The request is then forwarded to the destination PE. When the request is a read request, a response will be sent in the same fashion back to the originator.

The data-flow language SISAL (Section III-D) was chosen as the high-level interface for the TX16 because the data-flow principles of execution can be directly mapped into Occam.
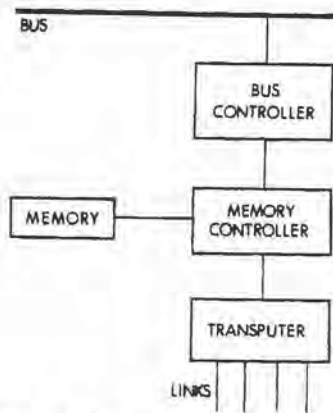
Fig. 17. A Processing Element in the TX16.

The converse is not true, however, since it is possible to design unsafe Occam programs which would have no corresponding part in the data-flow world. This mapping is made possible by the fact that both programming approaches rely upon the principles of *scheduling upon data availability*. Several numerical algorithms have been tested on a simulated machine and have demonstrated a near linear speedup for the size of the machine considered. It should be noted that this was obtained without the intervention of a sophisticated high-level language compiler. Instead, a data-flow language was used to provide the *programmability* needed. Indeed, for the same programming effort, a higher speedup would be obtained by the data-flow approach.

H. Comparison of Data-Driven Architectures

The different architectures presented in the above sections all represent different approaches to the problem of specifying scheduling in multiprocessor systems for digital signal processing applications. They each fit a different niche in the realm of problems encountered in this domain:

• The *systolic method* efficiently and cheaply implements parallel algorithms on potentially large numbers of simple processing elements. However, the design of the algorithm on the array of processors remains fixed and constrains the system to consistently solve the same problem.

• Architectures such as the *WAP* and the *ESL polycyclic processor* possess a greater degree of programmability. The WAP notably has no global synchronization mechanism since it relies upon the notion of a computational wavefront.

• The *data-flow multiprocessors* which we have described (the HDFM, the MIT tagged token data-flow machine, and the USC TX16) offer much more flexibility in that their scheduling is in a larger part decided at compile time. They possess no notion of central control and can deliver maximum parallelism in very complex algorithms without any intervention from the designer, programmer, or compiler. Data-flow machines find their applications in two cases: 1) in problems which involve large amounts of heuristics and decision making, or 2) in applications which require frequent reprogramming, thereby requiring the *high programmability* characteristic of data-flow systems. The data-flow interpretation model also presents the crucial advantage of *scalability* in that the same programming

method of a given algorithm can be used, regardless of the size and topology of the target machine. Finally, the *programmability* afforded by this approach translates into a higher performance for a given amount of programming effort.

V. A DATA-FLOW ARCHITECTURE WITH MULTIPLE LEVELS OF RESOLUTION

The data-driven model of execution has thus been demonstrated to provide a very efficient programming environment for the parallel execution of programs. We now show how the concentration of the model on small atomic operations can lead to many runtime inefficiencies. We examine the performance of a multi-level architecture.

A. The Multi-Level Approach

It has been observed [17] that the data-flow model of execution was often applied at too low a level and imposed much overhead at runtime. For instance, as was demonstrated in Section III, the description of a simple loop under the principles of the U-interpreter can impose a large number of overhead actors such as *D, L,* etc. For each loop, a minimum of five actors must be included. In addition, since we are in a data-driven environment, each datapath in the same loop (for instance, the index and the iterated variable) must "own" their own set of iteration actors, thereby multiplying the overhead. Let us consider a simple vector operation (Fig. 18(a)) which would be translated into the graph of Fig. 18(b). This is obviously a large overhead. Indeed, the data-flow interpretation mode should be used to uncover at runtime parallelism which would be difficult or impossible for a compiler to detect. Here, a relatively easy compiler intervention would quickly detect and exploit the parallelism available in the vector operation while the data-flow constructs would impose a large overhead.

This shows that the data-flow principles of execution can be advantageously applied with a higher granularity as it has been demonstrated [20], [42]. It is indeed intended to retain much of the distributed concepts introduced in the tagged token data-flow machine [22]. The architecture we consider comprises a large number of independent PEs which can communicate over a packet-switched interconnection network. The size and structure of the individual PEs, however, should match the higher granularity envisioned in this project and would implement powerful primitives such as complex vector operations.

The architecture of the machine is to be organized in a hierarchical fashion. It respects at the higher level the data-flow principles of execution but comprises powerful PEs at the lowest level. The PEs are to be vector processors. The advantages brought by this approach are several-fold:

• The principles of data-flow are maintained at all levels of execution which implies the same programming model. (Indeed, a vector operation can easily be detected in a higher level data-flow language.)

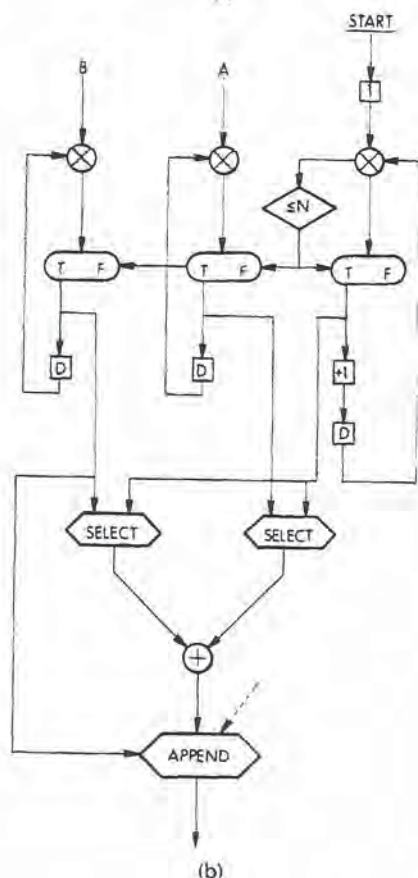• A continuous succession of more powerful but conversely more tightly coupled levels is implemented.

• The increase in performance brought by higher granularity can be directly implemented on this hierarchy of levels with increasing communication costs.
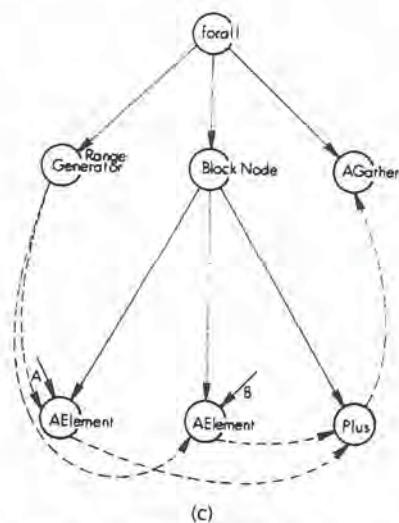
FORALL i in [1,N]
c:=a(i)+b(i)
returns array of c
ENDFOR

(a)

(b)

(c)

Fig. 18. (a) A vector addition in SISAL. (b) U-interpreter graph of a vector addition. (c) IF1 representation of a vector addition.

B. The Elementary Processing Element

In order to apply the data-flow principles of execution at a higher level of granularity, it appears that the individual PEs should be dealing with complex constructs which are more representative of the application under study. In the case of large signal and data processing problems, the input data which are processed are usually received at a high rate. Each datum undergoes the same operation and interacts only slightly with other data elements. This is the case for image processing applications in which *local* transforms are usually undertaken. This demonstrates that signal processing problems usually entail a low level which corresponds to *vector operations*. However, the highest level of operations includes such constructs as conditional, decision making, etc., for which the data dependencies cannot be identified clearly at compile time as they can be in a vector operation. Instead, *runtime dependencies* must be detected in order to provide safety of execution and adequate scheduling of our multiprocessor organization.

The individual PE is organized around a vector processor but also includes the capability to perform scalar operations as needed. The vector architecture can remain unspecified for the purpose of this section but could be an SIMD processor, a pipelined vector architecture, etc. For proper I/O function (i.e., communication with other PEs), the Processing Element is separated in the actual Processing Unit and the Communication Unit. The Processing Unit implements the actual vector functions while the Communication Unit is responsible for transferring data packets to/from the communication network and for the forwarding data packets to other processors.

C. The Software Environment

The elementary principles of execution are based upon an application of multi-level data flow. It was earlier demonstrated that the high granularity would considerably and positively affect the performance of a data-flow system. However, it is also known that the high-level, statically scheduled data-flow programming methodology could be used to design extremely powerful vector processors. The architecture we have described comprises therefore two levels: sophisticated processors are connected into a second hierarchy. This hierarchy also exists on a software point of view: program constructs must be partitioned together in order to best utilize the characteristics of the architecture.

We have chosen for our high-level programming interface the paradigm provided by SISAL (Streams and Iterations in a Single Assignment Language) as introduced by McGraw and Skedzielewski [38]. As shown in Section III, SISAL is a high-level language, the syntax of which resembles Pascal. It is different from conventional languages, in that it contains none of the side-effects associated with the usual programming approaches. An example of the specification in SISAL of the addition of two arrays *A* and *B* was given in Fig. 18(a). The existing compiler provides an Intermediate Form output (IF1). Not only does this output include the data-flow graph necessary for the runtime detection of data dependencies (called Data-Flow Graph DFG), but it also includes program structure information (Program Structure Graph PSG). As an example, the IF1 representation of the SISAL program in Fig. 18(a) is shown in Fig. 18(c). It shows the Program Structure Graph (PSG) in solid lines, while the actual Data-Flow Graph (DFG) is represented by dashed lines. The *forall* pseudo-node belongs to the PSG and is the head of a three-pronged tree: the left-most node contains the RangeGenerator actor which produces the index *I* from 1 to *N* (see SISAL program). The middle pointer is the actual

body of the loop. The right-most node *gathers* the results from the body of the loop. Note that in the DFG, the index is received by both array selectors (*AElement*) which receive *A* (respectively, *B*), and *I* and produce *A(I)* (respectively, *B(I)*). The *Plus* operator adds the two. Partitioning can easily be done along the edges of the PSG, provided a *cost matrix* is kept in order to easily assess the communication costs among the modules so isolated. An immediate heuristic comes to mind concerning the system under study: since the atomic vector processing unit is so well tuned to the scheduling of Generalized Vector Computations (GVCs), the partitioning process should examine the PSG from its leaves until it encounters a FORALL pseudo-node. A partition which would comprise the whole subgraph can thus be created. In addition, it should be noted that beyond this relatively simple partitioning approach, several optimization methods have been implemented. For example, nested loops can be exchanged or combined, code could be *hoisted* when data dependencies allow. These and other strategies have also been described in [29].

D. Applications

Kalman filtering [45] can be chosen as a representative example of some signal processing algorithms. It maps remarkably well on our architecture because of the multiple levels of hierarchy which are embedded in the algorithm itself. Indeed, the entire system could be described at the low level used by "conventional" data-flow architectures (Section IV). However, it should be immediately noted that most of these low-level operations can be grouped into higher order tasks. For example, the computational block which corresponds to covariance matrix estimation implements a complex matrix inversion. This algorithm itself entails repetitive applications of transpose operations.

As an illustrative example of the matrix operations which can be directly mapped onto our architecture, we have chosen the multiplication of two matrices. The SISAL code which corresponds to matrix multiplication is shown in Fig. 19(a). The corresponding IF1 output is shown in Fig. 19(b). In the graph, actors "RangeGenerator1," and "RangeGenerator3" broadcast index values *i* and *k* to the actors "AElement1" (Array Element select) and "AElement2," respectively. Once the actors "AElement1" and "AElement2" have received the index values, they forward the pointers *A[i, *]* and *B[k, *]* to the actors "AElement3" and "AElement4." "AElement3" and "AElement4" are also waiting for the index values *k* and *j* which are sent from the actors "RangeGenerator3" and "RangeGenerator2," in order to generate the elements *A[i, k]* and *B[k, j]*, respectively. The "Times" actor receives the two elements *A[i, k]* and *B[i, j]* and sends the product to the "Reduce" actor which accumulates the received data and forwards the result to actors "AGather1" as well as "AGather2" to form a two-dimensional array. The allocator analyzes the PSG and determines that the lowest level consists of a vector operation which can be easily assigned to a single processor for execution. However, it should be noted that for performance improvement reasons, the allocator *optimizes* the allocation of actors. Simple vectorizing compiler techniques can be applied to optimize the mapping of this application on our hybrid multiprocessor system.

```
type OneDim = array[ integer ];
type TwoDim = array[ OneDim ];

function MatMult( A, B: TwoDim ; M,N,L : integer
returns          TwoDim)

for i in 1,M Cross j in 1,L
    S :=
        for K in 1,N
        R := A[ I,K ] * B[ K, J ]
        returns value of sum R
        end for
returns array of S
end for
end function % MatMult
```
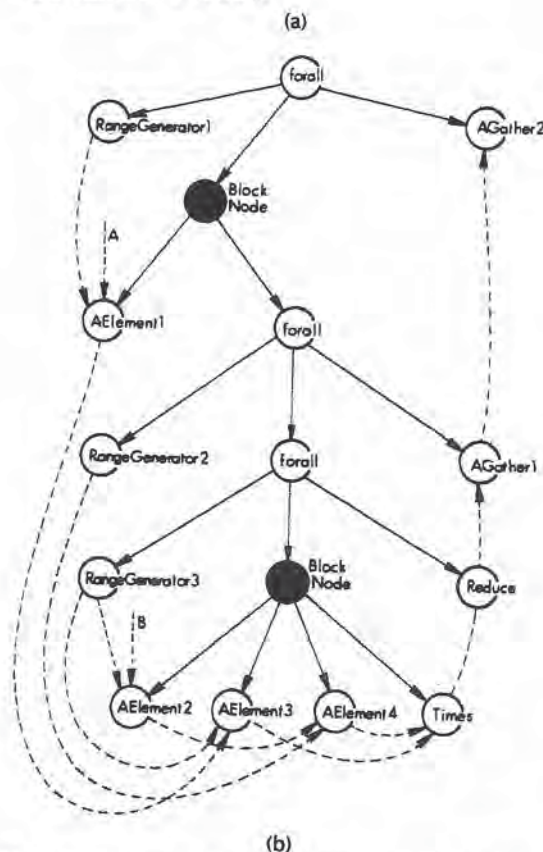
(a)

(b)

Fig. 19. (a) Matrix multiplication in SISAL. (b) Matrix multiplication in IF1.

VI. Conclusions

In this paper, we have demonstrated that data-driven principles are particularly well suited to the determination of the schedulability of operations in signal and data processing problems on multiprocessor architectures. The degree at which these principles are applied determines, for a large part, the domain of application of the system. When the various processors in the architecture are organized in a systolic array, the executability of instructions has been determined directly by the designer and the design remains frozen. This means that the application is fixed. However, comparatively high computational throughputs can be obtained from such organizations. When operation scheduling is decided by compiler intervention, systems such as the ESL polycyclic multiprocessor or the Wavefront Array Processor can be designed. These offer more programmability than systolic arrays. They also offer the potential for scaling up without a complex overall redesign. However, complex data-dependent operations cannot be easily implemented on these architectures.