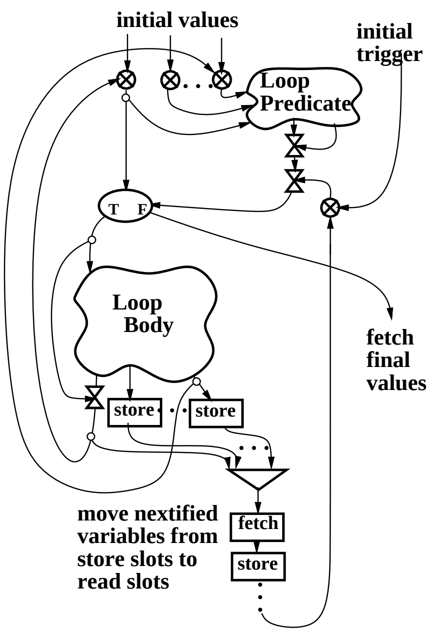

Figure 13: The frame-based-variable sequential loop schema with the 2-frame-slot-per-nextified variable approach to take care of WAR hazards.

6. MERGE SORT generates an array of n random numbers and then proceeds to sort it using the merge sort algorithm. The program is recursive but uses a loop to perform merging.

7. STATS generates an array of size n, fills it with random numbers and then finds the minimum, maximum, and average of the values in the array.

8. FFT computes the Fast Fourier Transform of a degree n polynomial. It takes $O(n \log n)$ time.

6.2 The Results

Table 1 shows the results. Due to the lack of space,the data is highly compressed. For each program, three pairs of normalized numbers, each corresponding to using a new sequential loop schema during compilation, are reported. The two numbers in each pair are separated by a slash "/". The first, *token-count*, corresponds to the total number of tokens processed by the Monsoon processor and can be thought of as the useful work done. The second corresponds to the elapsed time for running the program, and is call the *critical path length.* The difference between the two numbers is due to Monsoon's 8-way interleaved pipeline which requires eight fold parallelism at any time to keep a processor busy. When there is not enough parallelism, idle cycles are incurred, which are included in the critical path length but not the token-count. All the numbers are normalized, using the token-count and critical path length of the same program compiled with Culler's schema as basis.

While critical path length is important if we are interested in how fast each program runs on its own, the token-count matters more here as most of these benchmarks are

Program	Self-gating	Gated-store-Fbv	2-slot-Fbv
MATRIX MULTIPLY	0.68/0.71	0.53/0.55	0.53/0.55
BLOCKED MM	0.69/0.70	0.51/0.53	0.51/0.53
SIMPLE	0.96/0.96	0.94/0.95	0.96/0.96
PARAFFINS	0.88/0.89	0.89/0.90	0.88/0.89
FIB	0.71/0.75	0.35/0.80	0.41/1.06
MERGE-SORT	0.92/0.93	0.88/0.91	0.91/0.94
STATS	0.79/0.82	0.66/0.74	0.66/0.75
FFT	0.90/0.92	0.87/0.89	0.88/0.91

Table 1: Comparison of various Sequential Loop Schema. Each box contains 2 numbers: normalized token-count/ critical path length. Normalization is with respect to the run time of the respective programs compiled with Culler's schema.

small toy routines that we expect to find as inner loops of larger programs. Real programs are expected to be larger and have several instances of the inner-loops executing at the same time which together provide the required parallelism. We should, nevertheless, try to keep the critical path lengths closed to the token-counts so that the amount of parallelism that has to be provided by other code running at the same time is kept low.

The token-count of every program is lower when compiled with any of the three new sequential loop schemata than with Culler's schema. In addition, the fbv schema with gated-stores gives the best run time in almost every cases. Improvements ranges from 6% to 65%. The low improvement for SIMPLE can be explained by noting that SIMPLE contains loops with very few nextified variables, and also fairly large loop bodies. This means that the loop overhead was relatively small to start with, and hence there is little scope for improvements.

The critical path length of the benchmark programs shows similar improvement. The only exception is FIB compiled under the 2-slot fbv schema where the critical path is actually *longer* than the same program compiled under Culler's schema. This example illustrates the short-coming of the 2-slot fbv schema that we mentioned earlier, namely that of a long single thread where data is move between the write and read slots. Gated-store fbv is still the best in most cases. In other cases it is also not too far off from the best timing.

Based on the run time for our benchmarks, the gated-store fbv sequential loop schema emerges as the best sequential loop schema among those that we have studied, having both low token-counts and critical path lengths.

7 Conclusion and Future Directions

We explored various ways of implementing sequential loops efficiently on dynamic dataflow machines that perform matching with frame memory. Several other good schemata were not examined due to lack of space. We will quickly point out a few of them, although we believe that what we have presented in this paper are at least as good. One way of reducing the sequential loop overhead of the 2-barrier approach is to use 2 copies of code, with a barrier between each copy. In addition, the gates in the barriers can be removed by simply gating the switch predicate instead. We believe that the self-gating sequential loop schema is as efficient as this schema for most cases, given that most nextified variables are strict in their current values. By using the self-gating schema, we avoid the increases in code and frame sizes that accompany duplication of code. The 2-slot fbv can also be improved by duplicating code. This avoids the explicit moves between the two slots by having each copy use opposite slots for reading and writing.[5]

This work also pointed out very clearly the shortcomings of pure dataflow execution. The comparison of pure dataflow execution with RISC style code in Section 5.1 is applicable to dataflow execution in general. The overheads of switching, fanout and matching bubbles is *inherent* in the pure dataflow execution model. These problems have already been noted by other researchers, and means of overcoming them have been suggested by Iannucci[10], Culler[19], Traub and Papadoupolos[18, 20]. In general, a fundamental change in the way data is communicated, from *only* via tokens to using memory as well is needed. This together with *threading* by the compiler allows redundant synchronization to be removed.

The *T project[13], jointly undertaken by our research group at MIT and Motorola, will be investigating and testing these approaches on a real machine. Some interesting open questions remain, including what sort of memory hierarchy, instruction set and scheduling will efficiently support this style of execution. It appears doubtful that the traditional RISC style three-address register-to-register instruction sets with generic general-purpose registers is a good match for this style of execution due to the expected high frequency of thread switch.

Acknowledgements

Funding is provided in part by the Advanced Research Projects Agency of the Department of Defence under Office of Naval Research contract N00014-89-J-1988.

References

[1] B. S. Ang. Optimization of Loops for Dynamic Dataflow Machines. Master's thesis, MIT, EECS, Laboratory for Computer Science, December 1992.

[2] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[3] D. T. Chiou. Activation Frame Memory Management for the Monsoon Processor. Master's thesis, MIT, EECS, Laboratory for Computer Science, September 1992.

[4] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. The SIMPLE Code. UCID 17715, Lawrence Livermore Laboratory, February 1978.

[5] D. E. Culler. *Managing Parallelism and Resources in Scientific Datalfow Programs*. PhD thesis, MIT, EECS, Laboratory for Computer Science, June 1989.

[6] D. E. Culler and G. M. Papadopoulos. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, 1990.

[7] V. G. Grafe and J. E. Hoch. The Epsilon-2 Multiprocessor System. *Journal of Parallel and Distributed Computing*, 10(4):309–318, 1990.

[8] J. E. Hicks. Id Compiler Back End for ETS and Monsoon. CSG memo 310, MIT Lab for Computer Science, Cambridge, MA, June 1990.

[9] K. Hiraki, K. Nishida, S. Sekiguchi, T. Shimada, and T. Yuba. The SIGMA-1 Dataflow Supercomputer: A Challenge for New Generation Supercomputing Systems. *Journal of Information Processing*, 10(4):219–226, 1987.

[10] Robert A. Iannucci. *Parallel Machine, Parallel Machine Languages: The Emergence of Hybrid Dataflow Computer Architectures*. Kluwer Academic Publishers, 1990.

[11] Y. Kodama, S. Sakai, and Y. Yamaguchi. A Prototype of a Highly Parallel Dataflow Machine EM-4 and its Preliminary Evaluation. *Proceedings of InfoJapan*, 10(4):291–298, 1990.

[12] Venkat Natarajan, Derek Chiou, and B. S. Ang. Performance Visualization on Monsoon. *Journal of Parallel and Distributed Computing*, May 1993. (To appear).

[13] R. H. Nikhil, G. M. Papdopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th ISCA, Gold Coast, Australia*, May 1992.

[14] R. S. Nikhil. Id Reference Manual, Version 90.1. CSG memo 284-2, MIT Lab for Computer Science, Cambridge, MA, September 1990.

[15] R. S. Nikhil and Arvind. Id: a Language with Implicit Parallelism. CSG memo 305, MIT Lab for Computer Science, Cambridge, MA, February 1990.

[16] G. M. Papadopoulos. Program Development and Performance Monitoring on the Monsoon Dataflow Multiprocessor. In *Proceedings of the Workshop on Instrumentation for Future Parallel Computing Systems*. ACM Press, 1989.

[17] G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. Research Monograph in Parallel and Distributed Computing. MIT Press, 1992.

[18] G. M. Papdopoulos and K. R. Traub. Multithreading: A Revisionist View of Dataflow Architectures. In *Proceedings of the 18th ISCA, Toronto, Canada*, May 1991.

[19] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Proceedings of the 5th FPCA, Cambridge, MA*, pages 50–72, 1991. (Springer-Verlag LNCS 523).

[20] K. R. Traub. Multi-thread Code Generation for Dataflow Architectures from Non-Strict Programs. In *Proceedings of the 5th FPCA, Cambridge, MA*, pages 73–101, 1991. (Springer-Verlag LNCS 523).

[5] I would like to thank one of the referees for pointing out these possibilities.