A Coupled Multi-ALU Processing Node for a Highly Parallel Computer[1]

Stephen W. Keckler
April 12, 1994
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
skeckler@ai.mit.edu, NE43-619A

Many current parallel machines focus upon vector operations or coarse grained tasks. Performance can be improved, however, if tasks are parallelized using a hierarchy of granularities. Large blocks of code can be divided into coarse grained tasks; then each task can be scheduled to take advantage of fine grain instruction level parallelism. Instruction level parallelism techniques enable operations which have neither data nor control dependencies between them to execute simultaneously. Runtime scheduling mechanisms are necessary because some dependencies can not be discovered at compile time. Furthermore, runtime scheduling of multiple threads is especially important for multiprocessor tasks because even though they can exhibit high degrees of instruction level parallelism, they often have serial tail sections during which they consume few processor resources. The coupled architecture is intended to both provide runtime scheduling support and counteract the problem of serial operations within a thread, as it will allow each functional unit to independently associate itself with a thread of its choosing on every cycle.

To exploit instruction level parallelism, I propose a multiple functional unit processor that supports several active threads and interleaves them, not only in time, but also in space, based on their runtime processor resource requirements. This architecture will, at one extreme, behave like a Very Long Instruction Word (VLIW) [1, 2, 3] machine, while at the other perform as a process per element multicomputer. The location of this machine on that spectrum is determined dynamically on a cycle by cycle basis. Figure 1 demonstrates how the instruction streams for thread A and thread B have been scheduled at compile time, and how at runtime the tasks are interleaved to share processor resources. Each column in a thread's instruction stream represents an operation field for a single functional unit, and each row shows the operations which may be executed simultaneously. The empty boxes indicate that there is not enough instruction level parallelism to occupy all of the functional units.

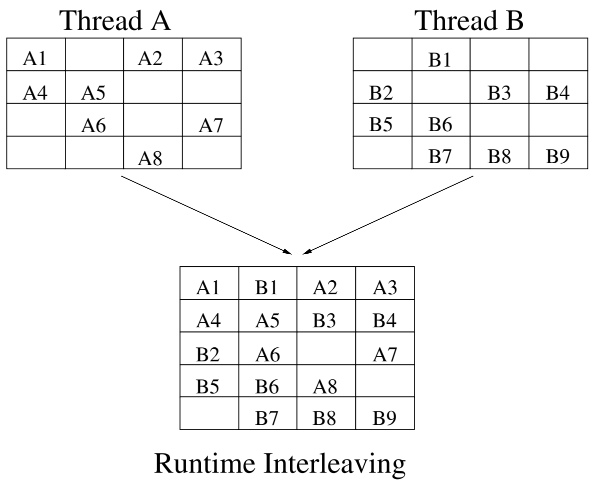

Figure 1: Interleaving of Instruction Streams

Each thread has its own instruction pointer and registers, and shares use of the functional units. The instruction pointer manages control flow, while data dependencies determine whether or not a given operation can be issued. Thus each

[1]This research is supervised by W.J. Dally and supported in part by a National Defense Science and Engineering Graduate Fellowship, by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-87K-0825, and by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation, IBM Corporation, and AT&T.

functional unit determines independently, through dynamic data dependencies, with which thread it will be coupled. Communication between threads takes place through a global shared memory. The presence or absence of data in a memory location controls synchronization between threads. Memory accesses that must wait for a synchronization operation are put to sleep so that additional memory request traffic is not generated. When a memory operation accesses a location, any idling memory operations wake up and begin executing again.

The innovation in this approach is combining compile time scheduling, in which a program fragment is scheduled as efficiently as possible, and runtime scheduling in which the operations from several threads are dynamically interleaved. Some of the new mechanisms required to support this level of parallelism include:

- Low latency context switching.
- The ability for multiple contexts to be active at once.
- Synchronization primitives in memory and registers.
- Low latency forking of threads.
- Interleaved memory to handle multiple simultaneous memory requests.
- Low overhead queueing for sleeping memory operations.

This architecture will perform well as a node in a massively parallel machine, as it will achieve good single thread performance, execute efficiently even when there is little instruction level parallelism within an individual thread, and tolerate arbitrary memory latency.

I have developed an experimental environment including a simulator and a compiler. The simulator is able to represent a machine with an arbitrary number of functional units, number of threads, and configuration. The compiler will be used to generate code for benchmark programs, and to explore such issues as partitioning programs into threads, detecting dependencies, scheduling instructions, and generating synchronization points. With the experimental system, I intend to investigate the viability of the coupled architecture and compare its performance to that of other approaches which exploit instruction level parallelism.

References

[1] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE Transactions on Computers*, vol. 37, pp. 967–979, August 1988.

[2] R. P. Colwell, W. E. Hall, C. S. Joshi, D. B. Papworth, P. K. Rodman, and J. E. Tornes, "Architecture and implementation of a VLIW supercomputer," in *Proceedings of Supercomputing '90*, pp. 910–919, IEEE Computer Society Press, November 1990.

[3] J. R. Ellis, *Bulldog: A Compiler for VLIW Architectures.* Cambridge, MA: MIT Press, 1986.

[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera computer system," in *Proceedings of the International Conference on Supercomputing*, pp. 1–6, June 1990.

[5] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th ACM Symposium of Principles of Programming Languages*, pp. 207–218, ACM, 1981.

[6] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188, ACM Press, April 1991.