Mapping Concurrent Programs to VLIW Processors

Hester Bakewell,* Donna J. Quammen, and Pearl Y. Wang
Department of Computer Science
George Mason University
Fairfax, VA 22030-4444

Abstract

In this paper, compiler techniques are presented for mapping a concurrent language to a VLIW processor. In particular, we examine techniques that map occam programs to one node of an iWarp processor array. These methods are of interest in situations where multiple tasks are to be placed on the same processor, e.g. when trying to reduce communications overhead, or when the number of tasks exceeds the number of processors. The techniques presented eliminate program specified communications in situations where the compiler can statically schedule these communications. These situations frequently occur in numerical applications. Compiler optimizations are discussed that pertain to packing techniques for long instruction words, register allocation, and scheduling. These compilation strategies for a single processor can be expanded to a compiler for a multiprocessor configuration.

1 Introduction

Modern parallel MIMD computers like the iWarp [4] frequently have two forms of parallelism: multiple processors and parallel functional units on each processor. The parallelism in the functional units is controlled by a *Very-Long-Instruction-Word* (VLIW). To be effective, a programming language and a compiler must be able to exploit both forms of parallelism. Toward this end, parallel languages and complex compilers have been emerging. Concurrent programming languages like occam [10] allow the programmer to express multiple streams of control and specify how these streams communicate. By investigating methods for compiling occam for the iWarp, we also learn how other concurrent languages with explicitly specified communications (e.g. Ada), can be compiled for VLIW parallel computers.

The occam language, formally defined by CSP [7], contains concurrent processes that communicate using one-to-one synchronized communications. This concept, in which the caller and the callee must name the communication path, is called explicit channel naming. The objective in this paper is to use the synchronized nature of occam communication to exploit the type of parallelism that is afforded by the iWarp's VLIW technology. Therefore, although occam was designed to code programs containing parallel tasks running on separate processors, this paper investigates how concurrent occam processes can use the registers and functional units of a single processor effectively.

VLIW computers are efficient for high-level languages only when a compiler is able to statically determine those instructions that can be executed in parallel. Current research into VLIW compilers has concentrated on decomposing large sequential programs using techniques such as trace scheduling, software pipelining, and hierarchical reduction of basic blocks [2, 3, 5, 8, 9]. However, a parallel language allows multiple tasks to run on a single processor. This can be an advantage when trying to reduce the overhead of interprocessor communications and when the number of tasks exceeds the number of processors. Although the VLIW compiler techniques mentioned above can be directly applied to sequential code within concurrent programs, this paper addresses additional methods that are directed at multitasking situations.

In the next section of the paper, a brief description of the occam language, the INMOS transputer, and the iWarp processor are presented. In the third section, a discussion is given of methods that can be used to map occam to the iWarp. We discuss how the parallelism of concurrent programs can be optimized for the VLIW computers. Efficient implementations of inter-process communications are considered as well as efficient register allocations.

*Hester Bakewell is now employed by General Electric Information Services

2 Background

Occam was developed simultaneously with the INMOS transputer. The relationship between the language and the architecture is explained by the following quotation: "A program running in a transputer is formally equivalent to an occam process ... a network of transputers can be described directly as an occam program" [11]. Both large grain and fine grain processing are supported by occam and the transputer.

The occam language is a block structured language. A block consists of either one statement or is built by combining several blocks into a construction. IF, CASE, WHILE, SEQ, PAR and ALT are used to form constructions. In particular, SEQ indicates a traditional sequential block, while the PAR instruction designates that the code blocks that are identified with it are to operate in parallel. The ALT indicates that two or more blocks are ready to run should they receive communication. When communication is received, only the blocks corresponding to that communication are allowed to run. (The ALT is similar to the select statement in Ada.) Further, constructions can be replicated. A replication that is used with PAR, for example, indicates the creation of a set of similar parallel occam processes. Communication in occam is specified by input and output statements that refer to channel names. For example, "chan1 ? x" indicates a request to receive data for variable x on channel "chan1," and "chan2 ! y" indicates an output of y's value onto channel "chan2."

There are several similarities between the transputer and the iWarp which suggest a natural extension of occam to the iWarp machine. In both the transputer and the iWarp, a collection of processors can operate in parallel and communicate through nearest neighbor links. In each architecture, every processor contains local memory and other functional units. The physical links between processors on the iWarp [4] and on the latest version of the transputer are handled by a communications unit that is separate from the actual processor. This allows the physical links to be viewed as virtual routes to arbitrary processors. In both architectures, the communication units are on the same chip as the processing units and run concurrently with program processing.

The transputer emphasizes concurrency, and therefore has a micro-coded scheduler that is used to handle virtual concurrent processes on the same processor, and a small register file that facilitates register copying. In contrast, the iWarp was designed for systolic processing and has a VLIW and a large register set that facilitates instruction parallelism.

3 Compiling occam to the iWarp

The hardware differences of the transputer and the iWarp differentiate an occam compiler aimed at the iWarp from one targeted to the transputer. In an occam program, a task switch can be as common as a procedure call. Because task switching normally requires the copying of the register file, the limited size of this file in the transputer helps minimize the context switch overhead. This promotes the execution of occam programs containing many small interacting tasks executing on the same processor. The iWarp, however, contains 128 registers. Copying all of them at each task context switch would greatly increase the overhead of an occam implementation, but restricting the implementation to a fixed subset of the registers would not fully utilize the architectural features of the iWarp.

The transputer's micro-coded scheduler handles on-chip concurrency. This scheduler is triggered by any request for communication, regardless of whether the communication is on-chip (between two concurrent occam processes) or off-chip (between two parallel occam processes), and by a clock, since round-robin scheduling is implemented. In the iWarp, a scheduler would have to be implemented in software. This implies that the scheduler acts as a separate task that not only gains control after any tasks give up control, but also is invoked by any interrupts. In addition, it is the responsibility of the scheduler to decide the order of task execution.

Unlike the transputer, the iWarp is a VLIW computer that is capable of handling up to nine instructions in one cycle. The compiler must be capable of determining what instructions can be executed in parallel and of scheduling (or packing) these instructions into a single cycle. For occam, this scheduling should take into account the existence of concurrent processes and any communication between these processes.

In the remainder of this paper, these issues are examined in the context of compiling occam programs to a single iWarp processor. Several compiler optimizations are presented that pertain to the packing of long instruction words, register usage, and scheduling. Although the focus of this paper is on compilation to a single VLIW processor with multiple parallel functional units, the techniques presented here can be expanded to an array of such processors.

3.1 Flow Graphs for Concurrent Tasks

A frequent first step in any compiler is the determination of the flow of control across basic blocks and between procedures. A basic block is a code segment that contains a sequence of consecutive statements. For occam (and other parallel languages), this involves the

development of a traditional directed acyclic graph for each task, where a task is a set of one or more procedures with a single thread of control. The basic blocks within all occam processes in a program need to be identified, and this may require the expansion of replication statements to statically determine channels and constants.

The flow of control across basic blocks is also dependent on any inter-process communications that are specified in the program. Because occam channels are statically coded within a program, it is straightforward to determine where one-to-one synchronous communications occur. In Figure 1(a), a sample graph is shown. The nodes are the basic blocks of two loops, and the dotted line indicates synchronous communication from "Task A," who sends the value of the variable x, to "Task B," who receives it in the variable y. The channel used in this case, is called "comm". The same notation can be used to indicate communications in a procedural flow graph. An example of this is given in a later section (Figure 3). Both types of flow graphs, along with a parent/child task hierarchy, provide information that can be used to perform optimizations related to task scheduling, register allocation, and instruction word packing. These optimizations are described in the remainder of this section. To perform them, one needs to determine which basic blocks can, must, or should never be active at the same time.

3.2 Scheduling of Long Instruction Words by Task Elimination

A VLIW computer has multiple functional units (resources) that can be executed simultaneously. Lam and others [2, 5, 8, 9] have developed techniques for extracting candidate instructions for the long instruction word from sequential programs. These instructions are ones that can be executed in the same cycle. The extraction methods include: trace scheduling, loop unrolling, loop software pipelining, and hierarchical reduction of basic blocks.

A single basic block rarely contains enough instructions to effectively make use of a long instruction word, especially since the execution order of the code in a basic block is usually governed by data dependencies. Hence, in all of the above techniques, several basic blocks, or instances of the same basic block (in the case of loops) are determined to be executable as a "sequential thread". Therefore, the code from all of these basic blocks is used to perform the packing of the long instruction word. The "sequential thread" is determined in light of a flow graph with conditional paths, by either duplicating code for both sides of a conditional, or by adding code to undo the effect of incorrectly executed instructions.

In an occam compiler for the iWarp, the techniques used for sequential programs can be applied to the code within a single task, if care is taken not to cross communications points. Because of the presence of concurrently executing multiple tasks, however, it should be possible to find basic blocks from different tasks that can be packed together to fill the long instruction word. This is of particular interest since basic blocks from different tasks should have no data dependencies except for those defined by inter-task communication. Hence, there should be fewer packing constraints in code from different tasks, as compared to code from basic blocks of the same task.

If parts of concurrent tasks were to be executed in this fashion, then they would actually be executing in parallel (capitalizing on the parallelism of the multiple functional units), and not concurrently. This may reduce the overhead incurred when scheduling concurrent tasks. Traditionally, the task scheduler must be invoked when concurrent tasks interact, as in the case with occam communication events. The dotted line in the flow graph shown in Figure 1(a) indicates an occam communication synchronization. Without optimization, the task scheduler would be invoked after the first task (either A or B) reaches the communication point. That first task will wait there until its partner task also reaches the communication point. Then the scheduler determines which of the two tasks will continue first, postponing the scheduling of the second task until later. Each of these scheduling points will most likely incur the overhead of a task context switch. Eliminating even some of these task context switches would enhance execution efficiency.

Because the occam language promotes synchronized communication between concurrent processes, it can be assured that some basic blocks from two communicating tasks can always be executed at the same time. These blocks, therefore, can be statically scheduled into a long instruction word. The blocks would be those basic blocks that follow the communication, and would include basic blocks from both tasks involved in communications. The identity of these basic blocks is derivable from the flow graphs. In the example described in Figure 1(a), basic blocks B_3 and B_4 would qualify, since they will always be ready to execute after a successful communication. This implies that the code from these two blocks can be statically scheduled together into the same instruction words, as is indicated in Figure 1(b). The techniques described for sequential languages can now be applied to map this single basic block into long instructions. Note that only one basic block needs to be scheduled after the communication, so that this optimization also reduces the task scheduling overhead.

Further optimizations are possible for the flow graphs described in Figure 1 if the compiler can statically bound the execution time of all basic blocks in the two loops. In this case, the need for the synchronized

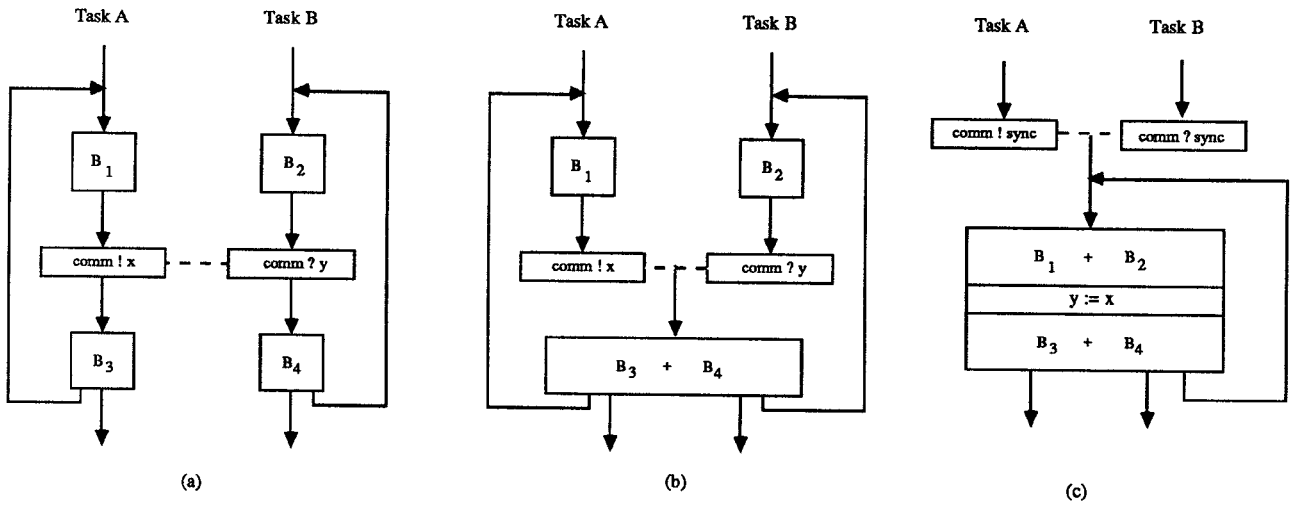

Figure 1: Eliminating synchronized communication

transmission of the second x can be completely eliminated. This can be accomplished by recognizing that if the compiler can guarantee a synchronized entrance to blocks B_1 and B_2, and also that the execution times of these two blocks can be statically bounded, then the compiler can also guarantee a synchronized exit from both blocks. No-op's can be inserted if one block is larger than the other. This assures a synchronized arrival for both sides of the communications, therefore eliminating any need for communications or scheduler overhead, and changing the communication into a simple assignment statement.

A synchronized entrance by both tasks into the beginning of the loop can be guaranteed because it is assumed that the combined block $B_3 + B_4$ has a bounded execution time, and it is known that the entrance to this block was synchronized by the first communication of x. This implies that after the first communication, the total loop can be reduced to a single block. This first communication is now being used to synchronize the entire loop. To simplify things, a new synchronization point can be added as shown in Figure 1(c), to replace the first transmission of x in the loop. Once both tasks reach this point, the total loop can be executed without the intervention of the scheduler. Both tasks would be executing in parallel using only the functional units.

If an occam program contains an ALT construct, the technique of eliminating the communication point which reduced Figure 1(b) to Figure 1(c) cannot be used. This is because resolution is nondeterministic and will occur at runtime. However, the potential for parallel basic block execution of an ALT construct with another occam construct (following a paired communication) does exist. In the example shown in Figure 2(a), an occam ALT construct is shown. A sequential code segment A is to be executed if an input value is received on channel "Chan1." Otherwise, the sequential code segment B is to be executed. The IF construct in the example contains three sequential branches, two of which contain matching communications events on this channel. The constructs can be paired, as shown in Figure 2(b), to indicate the potential for parallelism. Code segment A is duplicated for the two branches of the IF construct. The code following the output communication can be scheduled in parallel with the code following the corresponding input statement. This approach is similar to that used by Lam [9] to handle sequential *if* statements.

The method presented in this section of combining basic blocks after communication points can be used for any communication by two tasks on the same processor. The technique of further reducing basic blocks and eliminating communication points can be generalized to any situations where the time between the synchronization point and the communications can be calculated. Other situations where this is possible include systolic or multiple processes where the off-chip input is known to arrive at regular time intervals, and in the case where statements spawn sets of similar tasks at the same time, thus creating a known synchronization point. The technique is not possible when processes are dependent upon asynchronously generated communication. The advantage of eliminating tasks is that the runtime involvement of the scheduler can be avoided, which implies that several task context switches do not need

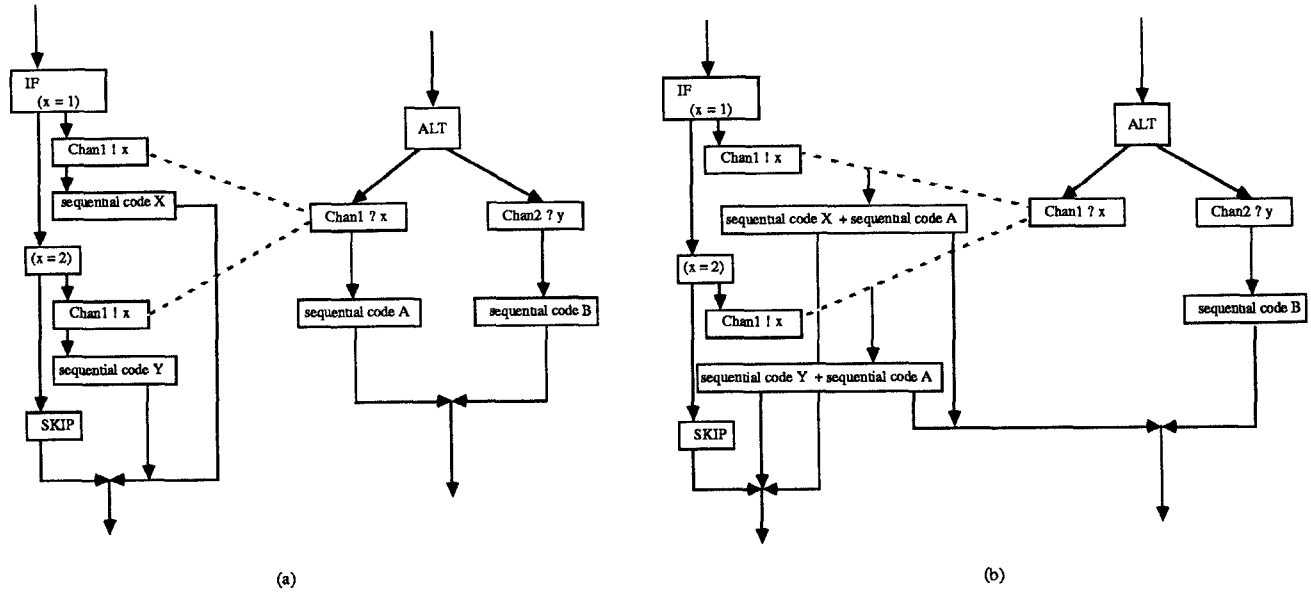

Figure 2: Matching communication in an ALT

to be performed. This addresses one of the claimed disadvantages of concurrent programming languages, i.e. the overhead of communications when using a scheduler. These techniques of graph reduction can be applied recursively if there are multiple communication synchronization points. The instructions in the newly formed basic blocks of the final graph are again candidates for instruction scheduling. It is assumed that the compiler can resolve discrepancies such as loops of different sizes.

3.3 Register Sets

The use of a VLIW suggests a need for a large register set that supplies operands to the multiple instructions. To reduce the overhead of a task context switch, concurrent processes would ideally use distinct register sets (removing the need to save and restore data housed in registers when a task switch occurs). The large register file on the iWarp provides the opportunity to do this. The naive approach for this partitioning would be to use a simple segmentation of the register set. However, this may cause undue register copying when there are many parallel processes, and when more than one process must be allocated the same set of registers. A more optimal approach would be to examine the flow graphs describing the procedural call flow, the control flow across basic blocks, and the parent/child spawning hierarchy. The information derived from these can be used to infer when tasks, or even parts of tasks, will actually interfere with each other.

Current compiler technology for sequential programs examines inter-procedural control flow when allocating registers [12, 13, 14]. This is done after intra-procedural allocation occurs [6]. A common approach is to examine the procedural call graph and then allocate registers, starting at the leaf procedures and continuing down the call graph to the root. Consider the procedural call graph for Task A in Figure 3. From this graph, it can be determined that procedures E and F can share the same registers because they will not both be active at the same time, but procedures $main_A$ and B cannot share registers because they will both need to be allocated at the same time. By noting that the procedures D and X communicate, it can be similarly determined that procedure chains $main_A$, B, D and $main_B$, V, X will be active at the same time and therefore, should not use the same registers.

To extend Mulder's approach to a concurrent environment, the call chain of each task is split into three sections: the chains above the communication (group 1), the chains below the communication (group 2), and those parts of the chains below the communication that are deallocated at the point of the communication (group 3). A fourth group (group 4) is also identified that contains those variables from groups 1 and 2 that would be used before and after the communication. Ta-

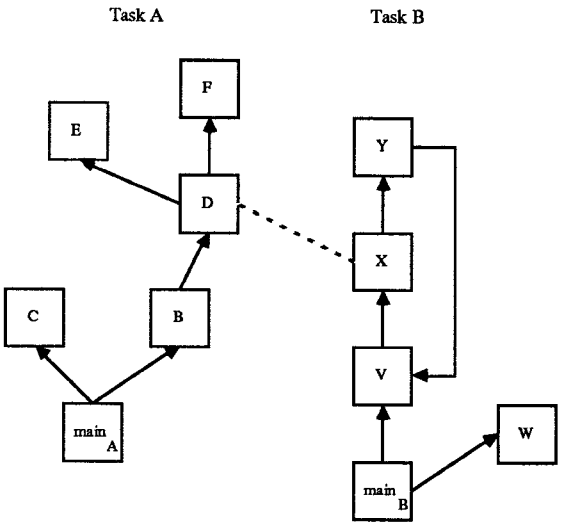

Figure 3: Procedural call graph with communications

ble 1 shows the procedures from Figure 3 that fit into these groups. Notice that the procedures that communicate are contained in groups 1, 2 and 4. In these procedures, the basic block graph will be used to make the determination of which registers are allocated for which group. It should also be noted that when using the techniques developed for sequential programs, group 3 will usually not be allocated registers that will in any way effect the register allocation at the point of the communication.

GROUP	TASK A	TASK B
1	D′, E, F	X′, Y
2	$main_A$, B, D″	$main_B$, V″, X″
3	C	W
4	D‴	V‴, X‴, Y‴

Table 1: Separation of call chain for flow graph in Figure 3

Pseudo-registers for each of these groups are allocated using the approach defined by Mulder. The goal is to find those tasks that communicate with each other (and therefore will be set ready to execute at the same time) and allocate their registers to be non-interfering whenever possible. Therefore, group 1 of task A and group 1 of task B will be set to be non-interfering as well as groups 2 of both tasks. The non-interference can be accomplished by first allocating registers to one task, and then allocating the group 1 registers of any communicating task in the group 2 registers of the first task. Although a perfect solution is not always possible, situations that occur in loops and those described in section 3.2 can be prioritized. In sample programs that were examined, it was found that long procedure chains were infrequent and the scheme could be easily applied.

This approach will require the copying of registers just prior to a communication. Since it cannot be determined when asynchronous communications will occur, the point when a task reaches a request for communication is a natural time to switch register sets. Group 4 is the only group that should be in registers at the point of a communication.

By linking the registers together, a co-dependency is established between those tasks. The scheduler must be aware of these co-dependencies. The fact that the goal of the register allocation scheme is to avoid copying at the point of a task context switch means that there is not an automatic purging of the register file (or part of it) when a task is removed from the CPU. However, register allocations can be shared between tasks, and therefore some swapping of data must be instigated if a task that does not currently own the registers is scheduled to execute. This must be handled by the scheduler. To facilitate this, the compiler informs the scheduler which tasks are co-dependent in execution, and which are anti-dependent because they have both been allocated the same registers. The scheduler tries to schedule co-dependent tasks that own their respective register

allocations. If the scheduler decides to allocate a task that does not own its registers, the scheduler first saves the registers. The runtime overhead of this approach was not extensive for the example programs examined, as there was little sharing of registers.

4 Conclusion

The optimization method described in this paper is similar to that of Zaafrai, Dietz and O'Keefe [15] which uses predictable execution times of code segments in two communicating parallel tasks to eliminate the need to check if a communications point has been reached. For VLIW processors, this parallelism can be on a single processor. The technique is not restricted to occam, but can be applied to any concurrent language where communications can be specified, and synchronization points can be statically determined, such as Ada. By eliminating predictable communications between parallel processes on a single iWarp processor, one of the major criticisms of using tasking languages can be minimized. The runtime overhead of the scheduler can be eliminated when sequential processing is possible.

The techniques for reducing task communications were successfully applied to several programs [1]. In the case of a matrix multiply program, all task context switches were completely eliminated. The resulting code was equivalent to that of a sequentially coded program. In a signal processing application, the communications was significantly reduced, but could not be completely eliminated due to ALT statements. In these and other exercises, it was demonstrated that the instruction word scheduling and register allocation schemes described in the paper offer potential reductions in the overhead associated with executing concurrent programs on a single processor. In multiprocessor situations, the technique could be directly incorporated into existing methods for mapping concurrent tasks onto several processors. An ongoing research project is concerned with the determination of mapping strategies that more closely integrate these single processor optimization techniques with load balancing issues.

References

[1] H. S. Bakewell. *Compiling occam to the iWarp.* 1990. MS project at George Mason University.

[2] Robert Cohn, Thomas Gross, Monica Lam, and P.S. Tseng. Archictecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor. *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, 1–13, April, 1989.

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

[4] S. Borkar et al. *iWarp Macroarchitecture Specification.* intel Corporation, April, 1989.

[5] Joseph A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, c-30(7):478–490, July, 1981.

[6] J.L. Hennessy and F. Chow. Register Allocation by Priority-Based Coloring. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, 1984.

[7] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, New York, 1985.

[8] Monica Lam. A Systolic Array Optimizing Compiler. *Ph.D. Thesis*, Carnegie Mellon University, May, 1987.

[9] Monica Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machine. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, 23(7):318–323, July, 1988.

[10] Inmos Limited. *Occam Reference Manual.* Prentice Hall, New York, 1988.

[11] Inmos Limited. *Transputer Databook.* INMOS, Ltd, Bath, 1988.

[12] H. Mulder. Data Buffering: Run-Time versus Compiler-time Support. In *Proc. of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 144–151, 1989.

[13] D. W. Wall. Global Register Allocation at Link Time. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, 264–275, 1986.

[14] D. W. Wall. Register Windows vs. Register Allocation. *Proceedings of the ACM SIGPLAN '88 Symposium on Compiler Construction*, 1988.

[15] A. Zaafrai, H. G. Dietz, and M. T. O'Keefe. Static Scheduling for Barrier MIMD Architectures. *1990 International Conference on Parallel Processing*, II:187–194, August, 1990.