Microflow: A Fine-Grain Parallel Processing Approach

Jon Solworth
Alexandru Nicolau

TR 85-710
November 1985

Department of Computer Science
Cornell University
Ithaca, NY 14853
# Table of Contents

1. Introduction ........................................................................................................... 1
2. Overview of fine-grain parallelism ......................................................................... 2
   2.1. Other fine-grain approaches ............................................................................. 2
   2.2. Systolic processors ......................................................................................... 2
   2.3. LIWs and VLIWs ............................................................................................ 3
2.4. Dataflow ............................................................................................................ 3
2.5. Microflow: parallel execution concepts ............................................................ 4
   2.5.1. Threads ....................................................................................................... 5
   2.5.2. Weaves ....................................................................................................... 7
   2.5.3. Microflow: the architecture ...................................................................... 8
   2.5.3.1. Message passing vs. shared memory machines .................................... 9
3. Software ................................................................................................................ 13
   3.1. Goals for a software support system ............................................................... 14
   3.2. Software design ............................................................................................. 15
   3.3. Compilation .................................................................................................... 16
   3.3.1. Comparison of approaches ....................................................................... 16
   3.3.2. Microflow compiler structure ................................................................... 18
   3.3.3. Global transformations optimizations ...................................................... 21
   3.3.3.1. Loop quantization ............................................................................... 21
   3.3.3.2. Symbolic unwinding dependency-graph ............................................ 23
   3.3.4. Compiler technology ................................................................................. 25
   3.3.5. Fortran extensions ..................................................................................... 25
3.4. Interpreter .......................................................................................................... 26
4. Proposed research .................................................................................................. 26
5. The Cornell research environment ......................................................................... 26
5.1. Computing Facilities ......................................................................................... 27
1. Introduction

The most important decision in choosing a parallel architecture is the granularity of parallelism; from this decision, the dominant characteristics of the architecture are determined. Ideally, the highest performance architecture is a fine-grain one, since all the parallelism can be exploited at this level. In practice, however, the runtime overhead at the fine-grain level is often several times larger than the theoretical performance speedup. Hence, many architectures employ coarse-grain parallelism, reducing overhead at the price of reducing parallelism. We describe a new architecture, Microflow which uses a combination of compile-time micro-scheduling of operations and Dataflow techniques. By scheduling fine-grain operations at compile time, Microflow avoids the high runtime overhead of control without loss of effective parallelism.

The Microflow architecture provides new opportunities for exploiting parallelism in ordinary programs. Because of the fine granularity of schedulable work, Microflow is able to expose parallelism not attainable with other parallel processors. However, the increased amount of control can easily become a nightmare on large codes —compiler technology is required to bridge the gap between the underlying machine and a source level specification. Some investigators, [Dennis80], [Arla85], have proposed Dataflow architectures as capable of exploiting large scale fine-grain parallelism. However, as we shall show latter, the overheads inherent in Dataflow computers range from 1-2 orders of magnitude larger than those in Microflow. This means that there would have to be 10-100 times the hardware resources to get the same speedup, or for algorithms with limited parallelism Microflow will perform the computation 10-100 times faster.

We propose the development of a software support system for fine-grain parallelism exploitation. This system will provide extraction of massively fine-grain parallelism on ordinary programs, and provide source-level deterministic-execution extensions to these programs. To evaluate the effectiveness of the software support system, we will write an
**interpreter** to measure and improve the dynamic behavior of this system.

The first half of this proposal is an overview of the issues in doing fine-grain parallel processing. The various approaches are described, and an overview of the Microflow architecture and concepts are discussed. The second half contains the proposal of issues to be investigated.

2. **Overview of fine-grain parallelism**

This section contains an overview of fine-grain parallelism, including other fine-grain approaches, Microflow's fine and coarse grain parallelism runtime modes (threads and weaves) and a discussion of the architecture of Microflow.

2.1. **Other fine-grain approaches**

Historically, fine grain parallelism has usually been applied to either small scale parallelism, in which case control can be dynamic, or to large scale parallelism, in which case the control is static. An example of the former is the CDC 6600 scoreboard, an example of the latter is systolic arrays. Dynamic techniques such as those used in the CDC 6600 scoreboard [Thornton70], are limited to speedups of a factor of 2-3, and effective use of about 10 functional units. Attempts to use more functional units would require a larger scoreboard and hence slow down the basic cycle time of the machine. Because these functional units would usually be idle, the effect of increasing the parallelism would be to slow down the computation rate. Note that this is a limitation of the solution domain, not of the problem domain.

2.2. **Systolic processors**

Systolic arrays offer much larger speedups than dynamic techniques, but are controlled statically. In systolic arrays, flexibility of control is traded for higher performance. the result is a class of architectures which run certain extremely important algorithms very efficiently but which are not very general. These architectures are well suited to those
problems which on a serial machine spend an enormous percentage of their time executing one (or a series of) algorithms which can be cast in a structure of very local and regular communication. Even the most flexible of these architectures, WARP [AKMS85], is a very specialized machine. We believe that successful systolic architectures will be used as back-end devices on general purpose parallel processors.

2.3. LIWs and VLIWs

Another approach for effective exploitation of fine-grain parallelism is Long Instruction Word machines such as the FPS-164 [Charles81] and VLIWs such as ELI [Fisher83]. On these machines, instruction issue is statically scheduled at code generation time, enabling multiple operations to be issued on the same cycle and removing the need for hardware interlocks. LIWs are difficult to program in assembly language; VLIWs (with instructions containing over 500 bits) are almost impossible. Such architectures require sophisticated compiler technology to statically schedule operations. These techniques have been successfully employed in scientific code for parallelism on the order of 5-10 times in the ELI project [FERN84]. However, a total reliance on static scheduling reduces the effectiveness of parallelism detection, and requires extremely time-consuming compilations.

2.4. Dataflow

We have discussed the basic tradeoff in fine-grain parallelism — flexibility of control versus degree of parallelism. An exception to this dichotomy is the Dataflow architectures. These architectures enable the use of massive fine-grain parallelism. However, Dataflow architectures are burdened with very high overheads of several types, and cause an efficiency loss of at least an order of magnitude. These overheads may be attributed to the following factors:

- **Instruction Ratio.** Dataflow requires 3-5 times as many instructions to be executed as on a conventional von Neumann machine Arla85.
\textbf{Communications Load.} Dataflow places an enormous load on the data paths—this is a factor of 3-4 times higher than on conventional architectures, for example [GuKiWa85]. This increased communication is especially problematic for VLSI implementations which are presently I/O limited, even for conventional microprocessors [Goodman83].

\textbf{Copy Semantics.} The functional programming model requires expensive copy semantics. There is an added cost in the execution of these languages which is \textit{proportional to the size of the data structures}. This problem is especially acute when dealing with large arrays [GPKK82], as typically arise in scientific processing.

\textbf{Dynamic Scheduling.} Dataflow is dynamically scheduled. Such architectures require long pipelines—to effectively use long pipelines the amount of parallelism needs to be much larger than the number of parallel execution units. Thus there is an inherent constant factor loss of parallelism in Dataflow architectures. This constant has been estimated to be a factor of 10 [Seitz85].

\textbf{Critical Paths.} Even though the parallelism available may be high, Dataflow machines may spend most of their time executing dead-end paths or code which is not on the critical path [GPKK82].

Note that the overheads of Instruction Ratio, Communications Load and Dynamic Scheduling are multiplicative. Copy semantics and critical paths are application dependent, and at best may require careful rewriting of the application (and limiting parallelism\footnote{Because the copy semantics of functional languages often require more intermediate values especially in scientific code than there is main store the parallelism has to be reduced so that the store does not overflow.}). In addition, copy semantics can increase the granularity in data structure modification to the coarse-grain level. In succeeding sections, we shall show how Microflow addresses and reduces each one of these overheads.

In this proposal, we examine a new mechanism for massive parallelism, which relies extensively on sophisticated compiler technology to drastically reduce runtime overhead. The components of the Microflow approach fall into four categories: Fine-grain parallelism \textit{(threads)}, coarser grain parallelism \textit{(weaves)}, architectural design and compilation techniques (PS). We show that the Microflow approach can solve the performance problems of Dataflow.

\section*{2.5. Microflow: parallel execution concepts}

The Microflow approach to parallel processing is to produce semi-static \textit{(compile-time scheduling of instructions}. The schedule is broken down into fine-grain \textit{threads of execution}
and process creation entities called *weaves*. Threads are schedules of data movement (communication) and data manipulation (computation), which once created run to completion. Threads do not denote the execution of a single instruction stream; rather they represent the largest, statically-scheduled fine-grain computation which can be efficiently produced by the compiler. It is the compiler's job to break up the program into threads: multiple threads of the same program can be executed in parallel if no data dependencies exist between them. Since the threads are parallel schedules, they involve multiple processing nodes—conceivably involving the entire Microflow Computer. We expect that for efficient execution, thread sizes should be on the order of hundreds of instructions (summed over all processors).

2.5.1. Threads

Because threads are statically scheduled, in-place manipulation of data structures can be applied (side effects) without ambiguity. There have been many objections to side effects, particularly since they do not produce pleasant semantics for program verification. However, side effects are a crucial part of 30 years of computer science algorithm development and are critical to the efficiency with which current algorithms run. Hence, one inherent overhead of Dataflow, copy semantics, is eliminated in Microflow. The argument of fine-grain efficiency for Microflow boils down to the static scheduling (low runtime overhead) of operation issue and the freedom to use (possibly selective) side effect inducing programs.

Threads are the base mechanism for achieving fine-grain parallelism. Because of the fine unit of execution, the overhead of scheduling operations in threads must be handled at compile time. Higher level parallelism is inherently more dynamic, and hence less suited for static scheduling. There are several reasons for this. First, it is more difficult for a compiler to determine parallelism at a higher level. Second, the overhead of scheduling dynamically is, as a percentage of running time for the task scheduled, much lower at
higher levels since the tasks are much larger. Third, the cost of making an overly cautious (compile-time) decision is more expensive. Hence, threads will be combined together dynamically to produce the computation. This combining of threads is called the *weave*.

To clarify the notion of threads, we will give an example of a simple single thread computation, and its implementation on Microflow. The problem, to sum the elements of the array would be written in FORTRAN as shown in figure 1.

```
S = 0
DO 100 I = 1, N
100       S = S + A[I]
```

Figure 1 – FORTRAN code for summing

As it stands, compiler optimization would be unable to improve the performance of the above computation without use of the associative law. However, if associativity is allowed, our symbolic optimization techniques will restructure this computation as a summing tree. Each leaf of the tree will sum \([N/P]\) elements (\(P\) is the number of processors), followed by a log \(P\) summing tree (see figure 2). Because of Microflow’s network architecture (Cube Connected Cycles), trees are easily embedded within it. The time to perform this computation would be \(O(N/P + \log P)\), which is optimal.

The FORTRAN code in figure 1 is transformed to the parallel version shown in figure 3. The array \(A\) is high order interleaved, so that \(N/P\) consecutive array elements are located at each node. Computations at relevant nodes is specified by the keyword \(At\). At each \(p\) satisfying the condition on \(at\), the code is executed. Note that although we have written the code parameterized on \(p\), the compiler will typically fold in the constants and produce simple straight line code for each processor (except for the summing loop).
At $p$ In $\{0..P-1\}$ do
    $s_p := 0$;
    For $j := \rho^p(N/P) \cdot (p+1)^p(N/P) - 1$ do $s_p := s_p + A[j]$; end;
end;
For $(i=0..\log(P))$
    At $p$ In $\{0..P-1\} \mid \text{bit}(p,i)$ do
        send($s_p, p \& \text{loworder\_ibits\_cleared}(i), \text{var}$);
    end;
    At $p$ In $\{0..P-1\} \mid \text{bit}(p,i)$ do
        $s_p := s_p + \text{var}$;
    end;
end:

Figure 3 – Microflow message passing version

2.5.2. Weaves

The weave is the dynamic coordination of the threads. Multiple threads (based on dependency analysis) can be run in parallel. These threads may be initiated simultaneously. However other thread's semantics require serial executions. In this case the execution of one thread is initiated by the tail instructions in the preceding thread.
Both of the above cases are trivial examples of weaves. Two examples which are more interesting are threads which conflict only for an initial segment of the thread and threads for which it is not possible to determine at compile time whether they can be run in parallel. In the first case, the compiler will place the initiating code for the second thread after the initial segment in the first thread. In the second case, the compiler will try to determine necessary conditions for the threads to be executed in parallel. These conditions are a tradeoff between very simple tests (cheap to compute) versus tests which exactly evaluate the necessary conditions for parallelism (squeeze out the maximum parallelism). The generation of these tests is completely under the control of the compiler.

The mechanism described above for creating weaves would also support multiprogramming, with the operating system ensuring that separate programs do not share thread numbers.

Threads are the primary mechanism for drastically reducing the dynamic scheduling in Microflow. Additionally, threads are shown as a mechanism for reducing or eliminating copy semantics. We shall see in the following sections how each of the other overheads is reduced.

2.5.3. Microflow: the architecture

The Microflow architecture consists of two components: the network and the processing nodes. The network topology is cube connected cycles [PrVu81]. A member of the cube connected cycle family can be described with two parameters: the number of elements in a cycle \( n \) and number of cycles \( r \) in the graph. A subset of the cube connected cycles is the hypercube, for which \( n \) equals one. However, cube connected cycles in which \( n \) and \( r \) are chosen appropriately have the same asymptotic bounds on well known algorithms as the hypercube and the additional very desirable property that each node has constant number of connections instead of the logarithmic number of the hypercube. Since the network nodes will in the long run be pin-limited, cube connected cycles will provide higher performance.
than the hypercube. Below is a picture of a cube connected cycle with \( n=4 \) and \( r=16 \) - a 64 processor ensemble.

![Diagram](image.png)

Figure 4 – Microflow computer network (cube connected cycles) containing 64 processors.

2.5.3.1. Message passing vs. shared memory machines

One architecture, the Cosmic Cube [Seitz85] has explicit message passing as its programming (and architectural) construct for synchronization. Because in the most general case, performing random accesses, the Cosmic Cube must invoke the processor at the memory cell end (this is a characterization of pure message passing systems), the parallelism is at a coarser level than in shared memory systems. The communications on the Cosmic Cube is slow because it is done in software, hence much of the parallelism detected by compilation techniques would be unusable due to overheads. The programming model requires explicit parallelization by the user. This makes it very difficult to take advantage of any parallelism other than the most important inner-loops. Moreover, as the Cosmic Cube scales up, both the latency and the percentage of code parallelized becomes more important. For these reasons, Microflow is applicable to a wider range of applications than the Cosmic Cube.

The NYU Ultracomputer has an omega network [GGKMRS84]. Our topology is a cube connected cycle; The Ultracomputer contains processors only at the network endpoints, while in Microflow every node has a processor. Since Ultracomputer communication nodes
are pin limited. Microflow obtains a $\log P$ advantage over the Ultracomputer in processing capability for a given size network. The shared memory model of the Ultracomputer requires a round trip for every memory access doubling the time (and traffic) over the most favorable message passing approach. The cost of memory accesses is reduced under load absorption techniques such as Fetch&Add and combining [GLR83]. Furthermore, efficient techniques exist for combining, see [DKSS85]. The shared memory model also results in busy waiting for synchronization.

There are strong advantages to both message passing and shared memory models. Message passing reduced traffic and provides a synchronization mechanism. Shared memory results in fine granularity for unpredictable memory requests and enables load absorption techniques such as combining. Microflow incorporates both models.

2.5.3.2. Node design

We have completed a preliminary design for a Microflow node. This design shows its feasibility; however we will need dynamic analysis of Microflow before the architecture can be tuned enough to implement our design in hardware.

Each node contains both routing and processing functions. A schematic of the routing of the node is shown below:

The principal routing function consists of taking network packets from the cube connection, the input cycle connection or the processor, and routing them either to the cube connection, output cycle connection, the processor or the memory. Note that there are many parallel paths in this scheme. In addition, a message might be destined for or generated by the processor at that node. Routing and processing are independent functions, so if a message is not going to the processor or memory at NODE, then processor, may execute instructions.
Figure 5 – Network node.

Note that a value in processor_i's register can be sent to processor_j's register without that value being stored in memory. Thus, we expect that for adjacent processors nodes, processor to processor communication will be faster than a local memory access.

The processing at a given network node is as follows. At each node there exists a processor. The processor time shares between multiple contexts using windows. There is a window for each active thread; each window contains general purpose registers and an instruction counter (which is local both to the thread and the node). Associated with each register is a presence bit, indicating whether a valid value is stored in that register.

The mode of processing is a combination of Dataflow and traditional sequential execution techniques. For each window, there is the next instruction to be executed on the data. These instructions are register-to-register instructions. The instruction waits until its register operand presence bits are turned on, at which point the instruction fires. A sequence of instructions continues to fire until some register is needed which is not yet available (or the thread ends and the window is released). At this point another window is selected for processing.
There are obviously a fixed number of windows per node. Since the number of active threads varies dynamically, we can either have more threads than physical windows —this would be a virtual window system —or explicitly limit the number of active threads available at a time by operating system control.

Figure 6 — Processor architecture.

For each window, an instruction counter is associated with it. Dataflow advocates hail their method as "instruction counterless" programs; however, as already discussed, at the finest grain of parallelism the overhead is typically not worth the parallelism. hence "low level" instruction counters are a performance enhancement since they allow caching techniques to be used. This allows a number of instructions to execute sequentially, thus enabling high performance (short pipeline) pipelining and for this case eliminating the high cost dynamic scheduling of Dataflow. This enables Microflow to reduce the communications load appreciably over both Arvind and Dennis models of Dataflow.
Finally, the use of register-to-register instruction sequences will reduce the instruction ratio, since the high instruction ratio of Dataflow is due, in part, to the extra movement of data. Whenever possible, Microflow schedules the operations on data at the site of the data.

In this section, the Microflow Architecture has been shown to reduce the communication load by processing the data locally and reduce instruction counts by reusing registers rather than copy data for each time it is needed.

3. Software

Our primary focus is on science and engineering applications. This has been the traditional market for super computers and it is estimated that by increasing computational rates by several orders of magnitude dramatic advances in fundamental fields would be made.

The scientific community needs not only high performance parallel processors, but a means of using them. Ability to run large, existing FORTRAN codes (with slight modification) is critical to the migration of these users. We believe that techniques which require scientists and engineers to completely recode their programs (such as Dataflow) are unfeasible unless there are large and certain performance gains. Because Dataflow is such a large departure from von Neumann machines, it is difficult to predict the resulting performance gains, if any.

Other techniques, such as required by Ultracomputer and Cosmic Cube, involve restructuring code (by hand) around message passing or operating systems primitives. These techniques require significant amount of expertise. With parallel processors likely to become the standard scientific engine in the 1990s (at costs comparable to today’s large minicomputers), these machines must be made more accessible.

While our primary interest is in science and engineering, Microflow is a general purpose parallel processor. We believe that Microflow will be effective across a larger range of
problems than, for example, today's vector machines. In addition to applications included in
the science umbrella (such as CAD/Graphics/Signal processing/Numerical codes) we believe
that these machines will also execute high-performance databases and AI applications.

The strategy for the use of such machines is then to provide access to the machine at
three levels of increasing capability/difficulty.

(1) Plain vanilla FORTRAN. Codes written in ANSI FORTRAN should run with speedup
and without modification on the Microflow Computer.

(2) Extended FORTRAN. FORTRAN extended with deterministic extensions. That is
intermediate computations have well defined states and computational results are not
order dependent.

(3) Concurrent programming. All features of the machine are exposed to the program-
mer. This mode is primarily for systems work, for code generation and for the run-
time library although sophisticated scientific users will occasionally write critical sec-
tions at this level.

3.1. Goals for a software support system

The software support system for a parallel processor consists of its compiler, resource
manager, and associated instrumenting and debugging tools. Because of the fine granular-
ity of parallelism available in Microflow, extensive reliance will be placed on the compiler
to exploit this parallelism, and to decide on the run-time compile time split of parallelism
scheduling and synchronization.

- **Effectiveness.** If parallel processors are to become widely usable, they must provide
attainable high performance and they must not be vastly more difficult to program
than today's von Neumann machines. To encourage use by the scientific and
engineering communities, these processors should support FORTRAN, which on suit-
able problems would attain significantly greater performance than is possible with
conventional architectures.

- **Efficiency** The software must be able to exploit substantially all of the power of the
underlying machine.
• **Tunable.** It must be possible to tune the application to run efficiently on the architecture. There are two software issues in tuning the application. First, performance bottlenecks must be detectable in the the applications code in a way which is comprehensible in terms of the source program. Second, selected portions of the code should be modifiable without rewriting much more than those sections. These modification should be possible at a variety of levels of increasing efficiency (and difficulty).

• **Robustness.** We expect massively parallel machines to be multiprogrammable. The reasons are similar to the those for uniprocessors: problem sizes will differ greatly, the amount of resources the job will use varies over time, the code must be debugged, and the computer must be economically justifiable. Given the multiprogrammability of these machines, it is impossible to predict even for simple problems, such basic parameters as network load, memory latency, computational rates at the processor nodes, or I/O loads. Successful systems must perform robustly; the performance should not vary in unexpected ways when the machine load changes.

• **Debugging.** Parallel program executions, by their nature cannot simply be viewed as (single flow of control) state transformations, as is the case with traditional von Neumann computers. For the execution of FORTRAN, it is possible to sequentialize the code; however, this would cause significant degradation of performance. It should be possible to enforce sequentiality for only parts of the computation. For example, it should be possible to run only certain iterations of a loop sequentially.

• **Fault tolerance.** A massively parallel processors will have failures involving small fractions of the computer. Intermittent and hard failures must be masked, and recovery mechanisms provided.

### 3.2. Software design

Of all the criteria enumerated in the previous section, clearly the most difficult to meet is that of effectiveness. We intend to address this problem through a combination of compilation techniques coupled with efficient runtime management. Although the problems involved in compiler optimization are in principle either NP-hard or unsolvable, good success has been achieved in practice. What has proved more difficult is to perform these optimizations in reasonable time [Ellis84].

For traditional architectures, optimization techniques would provide a performance increase proportional to a small constant factor (typically 2-4 times). If the compilation run was deemed too expensive, the code could be run without it. However, for massively parallel machines the speed-up is proportional to the number of processors, which might number in the thousands. Hence, large scale programs will be unrunnable without the compiler optimizations. We are especially interested in methods which perform the bulk of the paral-
lization independent of the number of processors. Another alternative is to have the compiler itself run in parallel or incrementally.

The issues of debugging and tuning are highly interrelated in the case of parallel machines. Clear notions of deterministic state transformation, especially in terms of the source code are necessary for debugging the code. For these reasons, and those of repeatability, the extensions to Fortran must result in deterministic intermediate results, even in the presence of bugs. For example, when a divide by zero interrupt occurs on a computation at line i (assuming straight line code), the state of the program (variable values) should be as though all instructions at line \( j < i \) have been executed, and that no instructions from line \( j > i \) would be executed. Since this is likely to limit parallelism, this property would hold only while debugging.

The issues of Robustness and Fault Tolerance are combined architectural/software issues. Two fundamental problems need to be addressed for performance, thrashing and deadlock. To prevent thrashing, runtime load balancing will be required. Sophisticated programs can be self adjusting, but in general the operating system must handle load balancing. Deadlock also can be handled at the architectural level (dual networks) or at software level (priorities, deadlock breaks, etc.). For these problems, combined software/architectural features can be studied using the interpreter.

3.3. Compilation

3.3.1. Comparison of approaches

The generation of hand-crafted code for efficient execution on machines containing hundreds or thousands of processors is a tedious task. For some important problems, new algorithms carefully designed for parallel execution will be developed, often tailored to a particular architecture. However, these algorithms are difficult to develop and implement – the problem must be of sufficient generality, interest and regularity to compensate for the
considerable effort. Even if these algorithms are parallelized, complex application codes will probably not run with large speed-ups. Compile-time program transformations are required to achieve efficient execution on parallel architectures.

Important work in the exploitation of source level parallelism has been undertaken at the University of Illinois at Urbana, where a system called Parafrase was developed [KKPLW81]. It relies on extensive global data-dependence analysis (Pi-Partitioning [Tarjan72]) to identify vectorizable or independently executable blocks of code. A memory disambiguation mechanism [Banjeree79,KKPLW81], is used to remove phantom dependencies from the data dependency graph, while numerous source level transformations (e.g., loop fusion, recurrence solving, wavefront method) [ChKu75,GPKK82] are used to enable vectorization and/or partitioning into independent blocks.

Another major effort in vectorizing of Fortran code has been undertaken at Rice by Ken Kennedy and his group. They have developed a variety of techniques which, like Kuck's, perform source level manipulation of code to enhance the potential for vectorization. Their system, called the Parallel Fortran Converter [AlKe82], uses an integrated data and control dependence graph as described in [AKPW83,Kennedy80], to determine the applicability of the various transformations, for example, loop interchange [AlKe84], and other reordering transformations (e.g., loop fusion, strip mining) [AKPW83].

Some of the above techniques (e.g., recurrence solving) may be adapted for use in the context of Microflow. However, there are no currently available techniques for parallelizing code for thousands of processors. The above-mentioned techniques exploit only source level, regular parallelism (i.e., loops relatively free of conditional jumps, and in which dependencies are uniform—that is, do not change from iteration to iteration). In particular, Parafrase does not schedule individual low-level operations, and thus is limited in its ability to exploit fine-grain parallelism. While the Alliant compiler [Alliant85] provides a means of synchronization on operation level dependencies, still the code is not reorganized to take
advantage of parallelism at the operation level. As a result, their fine-grain techniques are limited to exploiting parallelism explicitly available in innermost loops through a combined hardware/software interlock scheme. This solution does not scale. The inadequacy of coarse source-level parallelism exploitation is illustrated, for example, by Heuft and Little [HeLi82]. They show that the dynamic behavior of a loop coupled with fine-grained parallelism exploitation may provide significant speedup over the above methods. They point out that both the parallelism and efficiency (processor utilization) obtained by Kuck et. al. may be dramatically improved upon using Dataflow machines. We show, latter in this paper that the same parallelism can be achieved with our techniques, without incurring the runtime (dynamic) overhead of dataflow machines.

Trace Scheduling was first proposed as a compaction technique for horizontal microcode and was later applied to code generation for fine-grained small-scale parallelism in VLIW architectures [Fisher83]. In Trace Scheduling, directed acyclic paths (traces) through the original (sequential) code are selected and compacted as if they were basic blocks. Trace Scheduling is limited since traces can only be picked in acyclic code and innermost loop bodies. Furthermore, the notion of traces does apply to nested loops or independent loops/modules, trace scheduling cannot be used for massive parallelization. The reliance on a single heuristic for trace-selection and the monolithic structure of the transformation (one full trace at a time – traces cannot be combined) also limit the applicability of trace-scheduling.

3.3.2. Microflow compiler structure

Percolation Scheduling (PS) [Nicolau85] is a compiler optimization project that has been underway at Cornell for a year and half. This project grew out of the Trace Scheduling project at Yale and uses as a base the substantial data analysis section of the Bulldog compiler [Nicaus84,Ellis84].
PS is a compilation technique for parallelism exploitation based on a layered system of transformations. The base layer of PS consists of four transformations. These are simple to understand and implement, and are independent from any superimposed heuristics. On top of this base, high-level transformations are implemented. The base transformations map program-graphs into program-graphs, allowing incremental compilation. In addition, the transformation process lends itself to fine-grained parallel execution, due to the atomic and local nature of the base transformations.

Preliminary evidence shows significant speedups obtained by PS for existing machines, even in cases where previous approaches do not apply. However, much work remains to be done if the massive parallelism offered by the Microflow model is to be fully exploited. Compile-time scheduling, as currently done in PS, would be too time consuming on the massive scale necessary for Microflow. While PS can still be used in a hierarchical fashion to micro-optimize and tightly schedule code on a neighborhood of processors (e.g., spreading the body of a thread on adjacent processors), more high-level transformations are needed to perform the global partitioning of the programs (into threads), introduce synchronization points (weaves), and set-up communication paths (schemas) that take full advantage of a cube-connected cycles interconnect.

We propose the development of high-level transformation and mapping algorithms for Microflow, which when combined with PS, restructure computations (at compile-time) for massive fine-grained parallelism exploitation.

The overall organization of the system is shown below:
• **Fortran (with Extensions).** The front-end to our system will be a Fortran parser, augmented to handle our language extensions. Its output will be a program-graph, in which each node contains exactly one RISC operation.

• **Program Analysis.** Standard optimizations (e.g., dead-code and common-subexpression elimination) as well as enhanced flow-analysis are done here. The collected information annotates the simplified program-graph.

• **Global Transformations.** Global program restructuring such as loop quantization and symbolic unwinding are performed at this stage. Then the other layers of PS may be invoked interactively with the resource manager, to obtain an actual mapping to the Microflow architecture. Throughout this process, data-dependency is performed incrementally. The result is a symbolic parallel program graph, presenting a partial order on the code. In conjunction with the resource manager data is mapped to memories, threads and weaves are determined and communication schemas selected.

• **Resource Manager.** The resource manager interacts closely with the transformation process. It directs the actual mapping of code and data to resources (e.g., processors, memory banks), as well as communication allocation. As decisions are made, the resource manager builds a resources-time graph which tracks the utilization of resources in the machine as a function of time. This models the dynamic load of the hardware and enables its efficient utilization. For example, if a set of nested loops provides more parallelism than that available in the machine, they need only be parallelized enough as to take advantage of the machine. Alternatively, if not enough parallelism is available in one particular loop or nested loops, to take full advantage of the hardware, the compiler will attempt to overlap the execution of that code with some other part of the program.
• **Code Generation.** The last phase of the compilation process is the code generation. Once the code and data have been mapped to processors/memory, the actual object code may be generated. This may eventually be done by each processor in parallel. The low-level code for synchronization is also generated at this level.

### 3.3.3. Global transformations/optimizations

In the subsections that follow, some of the techniques which will be used in the Microflow Fortran compiler are discussed. Because of the scale of optimization, new techniques are required to efficiently and effectively exploit the parallelism possible in Microflow.

#### 3.3.3.1. Loop quantization

Loop unwinding has been long known as a means to increase the effectiveness of pipelined machines. More recently loop unwinding has emerged as a primary technique for the exploitation of fine-grained parallelism within loops. The basic technique is very simple. The body of the loop together with the control code (i.e. counter and exit-test) is replicated a number of times, with counters and test properly changed to reflect the new loop body. For example, figure 8b below shows the effect of unwinding the loop in figure 8a three times. This form of unwinding is usually used for simple for loops, but other forms of iteration can be dealt with in similar fashion.

```plaintext
For i = x, y, 3 Do
    A[i] := expr(i);
    If i > y Then Goto exit;
    A[i + 1] := expr(i + 1);
    If i + 1 > y Then Goto exit;
    A[i + 2] := expr(i + 2);
    If i + 2 > y Then Goto exit;
End;
exit;
```

Figure 8 - Simple loop unwinding: (a) Original loop, (b) Loop unwound 3 times

If full unwinding of all nested loops were possible then we could achieve optimal speedups (subject to data-dependencies and given enough processors) since no artificial constraints
would be introduced by the iteration order of the loops. This is not usually possible due to resource limitations. Still, we would like to get the same effect, albeit on a smaller scale, as if all loops were unwound. Unwinding only one loop (the current state of the art) is not satisfactory, since it exposes parallelism only in that loop.

Loop quantization is the unwinding of several iterations of a series of nested loops. The unwinding is constrained by three factors: We want to maximize the amount of parallelism exposed — i.e., minimize dependency chains. This means that we may want to unwind some loops more than the others. Space constraints, and, most important, correctness preservation (i.e., not altering the relative order of dependent computations). Loop Quantization is illustrated below. Notice that the "quantum box" (the unwound loop body) moves across the iteration space in "quantum jumps" — hence the name.

(a) Original loop  (b) Quantized loop  (c) Quantum box (k1 by k2)

Figure 9 — Sample 2 dimensional loop quantization.
For further details see attached technical report number 85-709.

3.3.3.2. Symbolic unwinding/dependency-graphs

Most of the execution time of any program is spent in loops. To effectively exploit the massive parallelism of Microflow, our compiler must be able to distribute loops over thousands of processors. In some cases these loops will contain few or no dependencies across iterations, and will have very simple control flow. Even in these cases parallelization techniques such as those in [ChKu75,Kuck76] may not be the most appropriate, as they do not take into account instruction lengths, communication delays, collisions, and network topology. Furthermore when the control flow is complex, or the dependencies are dynamic the previous techniques would fail to take advantage of low-level parallelism (see previous section). A general technique to take advantage of low-level parallelism, without requiring full unwinding of the loops (full unwinding is not even feasible at compile time, if the bounds of the loop are unknown). For example, consider the simple loop below (also used in section 2.5.1):

\[
\begin{align*}
S &= 0 \\
\text{DO } 100 I &= 1, N \\
100 & \quad S = S + A[I]
\end{align*}
\]

and assume that we have an 8 processor cube. While this is obviously a trivial example, is just a simple instance of the general problem. The "obvious" solution in this case has to partition the loop into eight equal sub-loops each computing a partial sum, distribute this code (and the associated array fragments) at each of the processor nodes, and add the results together in a sum tree. Given the number of processors, this solution would be optimal.

We are investigating a system that can symbolically unwind loops and their associated dependency graphs, and extract the dependency relations among the individual statements.
The term \textit{symbolically} denotes unwinding (or other transformation) which is performed without regard to the number of processors or to the values of loop bounds. Symbolic unwinding uses a graph structure to represent the program. We intend to use an algebraic manipulation package such as MACSYMA for simplifying the expression [MACSYMA]. Since MACSYMA requires a textual description, we will need to instantiate the graph. For the example we will show the instantiation is trivial, although the worse case is an exponential blowup in the number of terms. We are studying representations that are simple enough for manipulation and which on realistic code will be compact.

First, the system should be able to symbolically unwind the loop.

\[
S = S + A[1] \\
(S = S + A[i] \ i=3..N-2) \\
S = S + A[N-1] \\
S = S + A[N]
\]

While this is simple enough, we need the ability to automatically manipulate this structure.

Using, \textit{symbolic back substitution}, yields:

\[
S = (S = (S = (S = 0 + A[1]) + A[2]) + A[i] \ i=3..N-2) + A[N-1]) + A[N]
\]

Followed by dead-variable elimination.

\[
\]

Applying associativity\footnote{Associativity and distributivity are not, in general, numerically safe. When in doubt the system should warn the user of the danger and provide a means to change the decision.}:

\[
\]

Constant folding on zero, and extending the indices of the iteration:

\[
S = \sum_{i=1}^{N} A[i]
\]

By applying associativity again, the compiler could break down the computation to:
\[ S = \sum_{p=0}^{\mu} \left( \sum_{i=1+p}^{N} A[i] \right) / p \]

Which as shown earlier, can be distributed optimally.

3.3.4. Compiler technology

The structure and design of optimizing compilers is perhaps one of the best understood tasks in Computer Science. Optimization techniques have been very successful in improving the running time of programs; however, the compilers themselves are very time consuming.

Because it will be necessary to run extremely sophisticated optimizations on all large codes, we are investigating a means of speeding up the optimization process, especially during the period of debugging. Two possible approaches are to parallelize the compiler, or to perform optimizations incrementally, so that small changes to the source imply small increments to the runtime of the optimizing compiler.

3.3.5. Fortran extensions

We are studying extensions to Fortran for which the behavior is independent of network or processor speeds. Taking advantage of properties which are not determinable at compile time must be testable at runtime.

The extensions fall in the following categories:

- **Data Mapping.** For example, specifies whether a given array be high- or low-ordered interleaved, or whether the data is to be mapped onto a subcube.
- **Range information.** What are the possible values that the iteration or array indices can take on?
- **Size information.** What are the sizes of number of iterations or words that various entities consume?
- **Numerical properties.** Is it safe to apply associativity or distributivity.
- **Communications Schema.** Specifies the preferred communications schema.
- **Conditional Assertions.** Assertions which are true only if some condition holds.
3.4. **Interpreter**

We intend to build an interpreter which will accurately model the Microflow Architecture. Various architectural features will be evaluated, and the interplay between architecture, compiler and operating system will be studied.

The interpreter will enable us to evaluate several key issues for Microflow. These are the following issues discussed earlier in the proposal:

- Robustness of the network.
- Difficulty of tuning applications.
- Efficacy of compiler techniques.
- Tradeoffs for dynamic versus static scheduling.
- Performance estimates.
- Hardware/Software tradeoffs.

4. **Proposed research**

We have proposed research in compiler optimizations, compiler design and run-time behavior of the Microflow Architecture. To study these issues we intend to build an interpreter which will enable us to understand Microflow’s dynamic behavior, and to investigate compiler techniques, using Percolation Scheduling as a basis. Specific items to be addressed are:

- Compiler optimizations which are independent of the number of processors (such as symbolic unwinding).
- Fortran extensions which meet the deterministic, debugging and performance tuning requirements outlined in section 3.3.5.
- Parallel/Incremental compiler design to reduce compilation time when debugging.
- Runtime overheads and partitioning of programs into waves and threads.

5. **The Cornell research environment**

The Cornell University Department of Computer Science is a very highly rated department, particularly strong in theory and in programming languages. Architecture, although a relatively recent area for the department now contains three faculty members.
5.1. Computing Facilities

The proposed research will require extensive computing facilities, both for compilation and interpretation. We intend to use as the main machine for interpretation a recently acquired Gould 9080. This machine is a dual 5 MIPS processor which will be available for extensive runs. In addition, the Center for Theory and Simulation in Sciences and Engineering intends to obtain a large scale parallel processor. Such a machine would be ideally suited to our needs. If such a machine is acquired, we will request sufficient time to perform longer interpreter runs than would be possible using the Gould.

Additionally, the department has 5 Symbolics machines, which we intend to use for the compiler (PS is written in Lisp). The Symbolics machines also have MAXIMA, a symbolic manipulation package, which would be useful for our symbolic loop unrolling.

Since Cornell houses a National Center for Supercomputing, we also have access to scientific codes, which would be the grist for our system.
Bibliography

AIKe82.

AKPW83.

AIKe84.

Alliant85.

AKMS85.

Arla85.

Banjeree79.

MACSYMA.

Charles81.

ChKu75.

Dennis80.

DKSS85.

Ellis84.
Fisher83.


FERN84.


GPKK82.


Goodman83.


GLR83.

Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph, "Basic techniques for the efficient coordination of very large numbers of cooperating sequential processes," ACM Transactions of Programming Languages 5(2) pp. 164-189 (April 1983).

GGKMR84.


GuKiWa85.


HeLi82.


Kennedy80.


KKPLW81.


Kuck76.


Nicolau84.


Nicolau85.


PrVu81.