LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-421

FX-87 PERFORMANCE MEASUREMENTS: DATAFLOW IMPLEMENTATION

R. Todd Hammel
David K. Gifford

November 1988

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TR-421

FX-87 Performance Measurements: Dataflow Implementation

R. Todd Hammel
David K. Gifford

September 1988

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-83-K-0125.

Abstract

We analyze how much the FX-87 static effect system can improve the execution times of five benchmark programs on a parallel graph interpreter. Three of our benchmark programs do not use side-effects (factorial, fibonacci, and polynomial division) and thus did not have any effect induced constraints. Their FX-87 performance was comparable to their performance in a purely functional langauge. Two of our benchmark programs use side-effects (DNA sequence matching and Scheme interpretation) and our compiler was able to use effect information to reduce their execution times by factors of 1.7 to 5.4 when compared with sequential execution times. These results support our thesis that a static effect system is a powerful tool for compilation to multiprocessor computers. However, the graph interpreter we used was based on unrealistic assumptions, and thus our results may not accurately reflect the performance of a practical FX-87 implementation. The results also suggest that conventional loop analysis would complement the FX-87 effect system.

Categories and Subject Descriptions: D.1.3 [**Programming Techniques**] Concurrent Programming; D.3.4 [**Programming Languages**]–Processors: *Compilers*; D.3.m [**Programming Languages**]–Miscellaneous

General Terms: Benchmarks, Testing, Parallelism

Additional Key Words and Phrases: functional programs, imperative programs, side-effects, effect systems, DNA sequence matching, Scheme interpreter, dataflow graphs

1 Introduction

This report documents a series of experiments that we performed to explore our thesis that the FX-87 effect system permits a compiler to schedule imperative programs (*i.e.*, programs that may contain side-effects) for execution on a parallel computer. We will assume that the reader is familiar with the details of the FX-87 language [Gif87, Luc87] and with dataflow [Arv87a].

Our major results are as follows:

- On programs that do not contain side-effects, such as fibonacci and factorial, FX-87 performs just as well as functional languages. Although FX-87 programs can include expressions with side-effects, the compiler takes full advantage of the parallelism in a program written without them.

- The FX-87 effect system improved the performance of an imperative DNA sequence matcher by a factor of 3.7 to 5.4 (depending upon the method of compilation) over sequential execution. DNA sequence matching is a scientific application that entails stepping over the elements of a two-dimensional matrix and mutating them. Loop analysis or parallel vector operators would greatly improve the performance of FX-87 on this application.

- The FX-87 effect system improved the performance of the Scheme interpreter, a program strewn with side-effects, by a factor of 1.7 over sequential execution. The Scheme interpreter is a large, complex, heterogeneous program, which presents a very difficult challenge to the effect system.

These results support our thesis that an effect system can be used to schedule imperative programs for execution on a parallel computer. However, our experiments were run on a simulated graph interpreter using a set of assumptions that does not accurately model the real constraints on a practical implementation. This affects our analysis in two ways. First, the improvement factors quoted here must be regarded as optimistic. Second, problems associated with real implementations of functional languages are hidden by the ideal environment, and therefore we are unable to investigate the efficiency benefits of using masked side-effects in functional programs.

Future FX implementations will permit us to more accurately measure what performance improvements might be expected in practice.

The rest of the paper describes our results in greater detail. We first describe the dataflow compiler that we built (Section 2), then we examine the experimental results (Section 3). For a detailed description of the compiler and complete listings of the source code for the benchmarks, see [Ham88].

2 The FX-87 Dataflow Compiler

The FX-87 dataflow compiler translates FX-87 programs into dataflow graphs suitable for interpretation by the GITA dataflow simulator [Arv87a]. The eight stages in the compiler are as follows:

- *Parsing.* A simple recursive-descent, predictive parser performs the following tasks:
  - Checks that the program is syntactically correct, and associates an empty information node with each expression. The information node will be used by some of the following modules to store information about the associated expression.
  - Alpha-renames all the bound variables.
  - Transforms syntactic sugar to its desugared equivalent.
- *Type and Effect Checking.* This is the same type checker used in the FX-87 interpreter [Jou88]. It steps recursively through the parse trees and annotates each expression with its statically computed *type* and *effect.*
- *Type Removal.* FX-87 has a typeless semantics. In other words, after the type checking module, we can remove all type-related expressions from the parse tree. For example, this module replaces `plambda`, `proj`, and `the` expressions by the expression they contain. The information nodes for the remaining expressions are left unchanged.
- *Lambda Lifting.* This module moves moves `lambda` expressions to top level, and uses closures to model the environment associated with a procedure. The output of this module is a group of parse trees for

top-level definitions. Each parse tree is a `lambda` expression whose body contains no `lambda` expressions. For more information on lambda lifting, see [Joh86].

- *Adding* `effect-synchronizer` *instructions.* To handle effect constraints in the dataflow graph, special sorts of synchronization are required at the ends of procedures. This module adds `effect-synchronizer` instructions to the parse tree, placing one around the body of each procedure. Here is an example:

```
(define goo
  (lambda (...)
    (effect-synchronizer
        ...
        ...))).
```

  Later, when the parse tree is transformed to a dataflow graph, we will see exactly what the `effect-synchronizer` instruction does.

- *Computing Effect Constraints.* Using the effect information in the parse tree, this module annotates each expression with the expressions for which it must wait to ensure proper sequencing of side-effects. `Effect-synchronizer` expressions are annotated to wait for any expression that has side-effects and is contained within its body. We will refer to the constraints computed by this module as *conflict edges.*

- *Generating a Dataflow Graph.* This module transforms the parse tree for each top level definition into a program graph, an intermediate form used by the ID compiler [Tra86a]. Program graphs, a type of dataflow graph, are easily transformed into a machine level graph that runs on GITA (Graph Interpreter for a Tagged Token dataflow Architecture) [Arv87a]. To transform the parse tree into a program graph, the module must handle several different types of expressions:

  - Sequencing instructions, such as `begin` and `let`, have no corresponding graph constructs. This is because, in a dataflow graph, sequencing is based only on the flow of data.

- Applications become apply nodes in the graph. If an application must be delayed because of side-effects, then an extra arc is drawn to the application to prevent it from executing prematurely.
- `Effect-synchronizer` instructions simply return the value of a procedure invocation, but they delay it until all the side-effects in the procedure have taken place. Therefore, if application B must wait for application A because their side-effects interfere, then a return value from A can be used as an indicator that application B may proceed.

- *Transforming Program Graph to Machine Graph.* We use the same modules as the ID compiler to go from program graph to GITA machine graph. Several optimization modules make changes to the program graph, and then each program graph node is macro-expanded into machine graph nodes, each of which corresponds to a GITA machine-level instruction.

2.1 The FX-87 Library

The previous section suggests that the graphs produced by the FX compiler contain nothing but apply nodes and arcs (for data and effect constraints). This is not quite correct; these are the exceptions:

- There are nodes that handle the overhead of resource allocation, procedure invocation, and other low-level operations.

- Some simple applications are transformed directly into a node that performs the operation, as opposed to a procedure invocation. For example, `(+ 3 4)` is translated into a *plus* node, not an invocation of the procedure called `+`.

Despite these exceptions, a large part of the computation in a compiled FX-87 program is performed by calling the many library functions described in the FX-87 reference manual. Therefore, we will briefly describe the FX-87 library for GITA.

We have augmented the FX-87 compiler with the directive `#inline`. When the compiler sees `#inline` wrapped around an application, it compiles the application into a machine-level instruction, and not a procedure

invocation. Using this directive, we write the FX library procedures in FX, and compile them using the compiler we have described. For example,

```
(define + (lambda ((x int) (y int)) (#inline (+ x y))))
```

defines the library procedure +. As an example of storage allocation procedures, here is the definition for **new**:

```
(define new
  (plambda ((r region))
    (plambda ((t type))
      (lambda ((y t))
         (let ((empty-ref
                  (#inline (primitive-make-vector 1))))
            (#inline (primitive-vector-set! empty-ref 0 y))
            empty-ref)))))
```

This procedure uses a machine-level instruction called `primitive-make-vector` to allocate a vector of length one. It initializes the vector using the machine-level instruction `primitive-vector-set!`, and then returns it. In FX-87 it is important that the storage allocation procedures atomically allocate and initialize storage. In the above example, we do not want the **new** procedure to return the new structure uninitialized. We are assured that this will not happen because of the way we compiled `effect-synchronizer` instructions. Remember that a procedure does not return its value until its side-effects have completed. Therefore, the new structure will not be returned until it has been initialized.

The code for **new**, as it appears here, cannot be type checked because the types of `primitive-make-vector` and `primitive-vector-set!` are undefined. For a detailed explanation of the modifications that must be made to the type checker to allow it to type check the above code, see [Ham88].

2.2 Changes to GITA

GITA is a dataflow machine designed to interpret graphs compiled from the language ID [Nik86]. ID is a high-level functional language enhanced with

I-structures [Arv87b]. In order to run FX-87 programs on GITA we had to make the following changes to the machine:

- GITA uses a memory architecture called I-structure memory. I-structure memory maintains deferred read lists for every location, and it prohibits the programmer from writing any location more than once. For the purposes of FX, we added a more standard memory module. Each location in this new memory module can be written any number of times, and it is an error to read an uninitialized location.
- We added an I/O module for FX's I/O instructions.
- ID is an untyped language, therefore each alu operation must check that its arguments are of the correct type. FX is typed, therefore we were able to simplify the alu.

3 Analysis of Parallelism in FX

Using the compiler described in Section 2, this section analyzes the ability of FX-87's static effect system to discover parallelism in a number of different types of programs. Our primary goal is to determine the power of a static effect system under ideal conditions; therefore we will ignore some of the realistic influences on parallelism, like communication latency and synchronization costs, by running GITA in *idealized mode*. Specifically, we assume the following:

- All alu operations take one time unit.
- Any number of operations can be performed in one time unit.
- Communication is instantaneous.
- Each operation executes as early as possible, *i.e.* as soon as all its input data are available.

As a result of these idealizations, the graphs in this section are useful only in the way they compare to each other, and do not purport to reflect the intricacies of execution on any real machine.

3.1 Some Simple Functional Examples

In order to assess the power of the FX effect system, we need to compile the example programs both with and without effect information and then compare the results. To this end, the compiler was constructed with a switch to select one of the following:

> **Compilation Method 1– Forced Sequential Execution:** The compiler module described earlier, which uses effect information to annotate the parse tree with conflict edges, is replaced by a module that simply draws conflict edges from one application to the next in the order dictated by the sequential semantics of the language. This does not completely remove parallelism from the resulting compiled code; GITA is still permitted to exploit any parallelism in lower level operations, such as procedure invocation and resource allocation. Furthermore, parallelism remains in library routines. The goal of this method of compilation is to simulate the execution of the program in the absence of any effect information, but with all other factors the same.
>
> **Compilation Method 2– Parallel with Effect-Synchronizers:** Programs are compiled with all the modules described in section 2. The **effect-synchronizers** around procedure bodies delay the return of a value until all the effects in the procedure body have occurred. Later, we will reduce the barrier introduced by these **effect-synchronizers**.

3.1.1 Factorial

Here is the function factorial written recursively in FX:

```
(define factorial (lambda ((x int))
                    (the pure int (if (= x 1)
                                      1
                                      (* x (factorial (- x 1)))))))
```

Compiling this function and running it on GITA produces many statistics that allow us to examine the characteristics of the compiled code. Three of the graphs produced, the *alu operations profile*, the *dynamic operation mix*

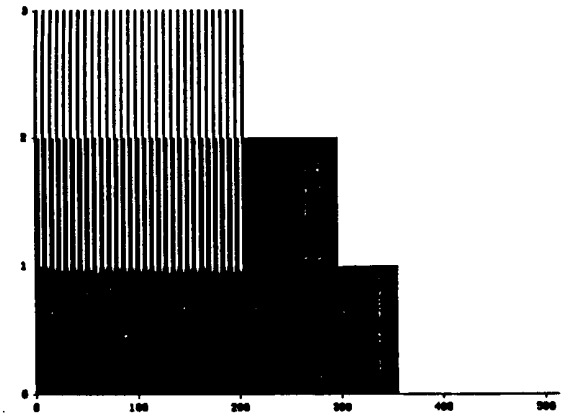

Figure 1: (Factorial 30): Alu Operations Profile– Sequential Execution

table, and the *invocation tree profile*, are of particular interest and will be examined frequently throughout this section.

The first of these graphs, the *alu operations profile*, is a plot of alu operations versus time. The alu operations are, for the most part, simple binary or unary operations that can be executed in a single cycle. The unit of time in these graphs is one GITA cycle. The *alu operations profile* shown in figure 1 was produced by calling the `factorial` function, compiled using method 1, on a value of 30. Figure 2 shows the same function called on the same value, but compiled using method 2.

Comparing these graphs, we make two observations. First, the sequential execution profile shows as many as three operations occurring at a single time step. This concurrency results from parallelism in GITA's procedure invocation mechanism. This sort of low-level parallelism is allowed to remain, because our goal in forcing sequential execution is to avoid only that parallelism gained by the compiler's static effect analysis. Second, the sequential and parallel executions took almost the same amount of time. This is because the dataflow constraints in the factorial function do not permit any parallel computation.

Next, we examine the *dynamic operations table*. This table shows the makeup of the set of instructions executed by the alu during the execution of the program. Tables 1 and 2 show the results produced by running the two different compilations.

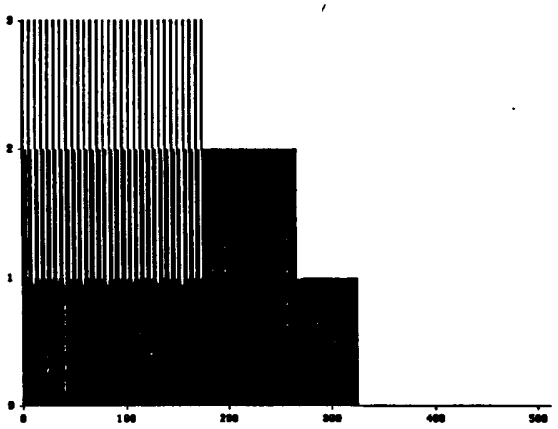

Figure 2: (Factorial 30): Alu Operations Profile– Parallel

Instruction Type	Count	Percentage
Identity	266	40.86
Tag	176	27.04
Arithmetic	88	13.52
Switch	60	9.22
Resource	59	9.06
Constant	2	0.31
Total	651	100
Critical Path		355

Table 1: (Factorial 30): Dynamic Operations Mix– Sequential

Instruction Type	Count	Percentage
Identity	237	38.10
Tag	176	28.30
Arithmetic	88	14.15
Switch	60	9.65
Resource	59	9.49
Constant	2	0.32
Total	622	100
Critical Path		326

Table 2: (Factorial 30): Dynamic Operations Mix– Parallel

Most of the instruction type categories in the *dynamic operations table* are clear. The type called *tag* includes all those instructions that alter the tag of a token (*i.e.,* for procedure calls and manager requests). The type called *switch* is for the switch instruction used to send tokens to the selected arm of a conditional. *Resource* instructions include getting new contexts for procedure invocations and the allocation of new memory structures. The *critical path* is the same as total execution time on the *alu operations profile.* It represents the longest chain of data and conflict dependencies in the program.

The above tables immediately suggest two questions: why so many identity instructions? and why are there more identity instructions in the sequential run? The answer to both is that identity instructions are used to perform synchronizations. They do this in two distinct ways: first, they are placed in the machine graph to handle some low-level runtime chores, such as completion detection. Second, they are used by the FX-87 compiler to control the sequencing of instructions. In the case of parallel execution, they enforce side-effect constraints. In the case of sequential execution, they are used to force the applications to proceed in sequence. Since forced sequential execution requires more constraints than parallel execution, sequential runs will, in general, execute more identity instructions. Fortunately, the additional instructions needed to force sequential execution are a negligible percentage of the total number of instructions executed, and therefore do not introduce any significant unfair bias against sequential execution.

Finally, we look at the *invocation tree profile.* This graph plots the number

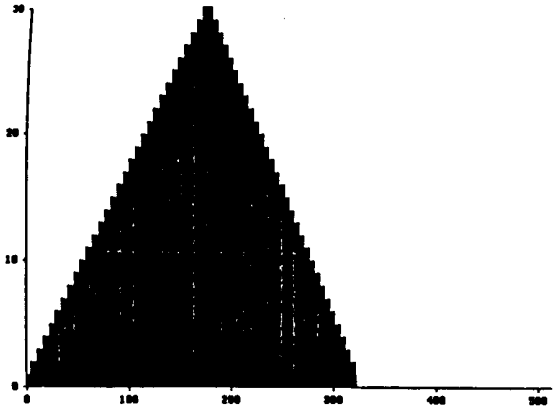

Figure 3: (Factorial 30): Invocation Tree Profile– Parallel

of procedures that have been invoked but not terminated versus time. Since the profiles for sequential and parallel execution look practically the same, we show only the latter (Figure 3). As we would expect, the factorial function applied to the number 30 results in a maximum recursion depth of 30, as shown in the profile.

3.1.2 Fibonacci

As we saw in the last section, some programs, such as `factorial`, may afford very little parallelism because of dataflow constraints. Now we will look at another simple `pure` program that is less constrained by the flow of data.

```
(define fibonacci (lambda ((x int))
                    (the pure int (if (or (= x 1) (= x 2))
                                      1
                                      (+ (fibonacci (- x 1))
                                         (fibonacci (- x 2)))))))
```

Compiling `fibonacci` using both methods 1 and 2, and then evaluating `(fibonacci 10)` for both cases, results in the two *alu operations profiles* shown in figures 4 and 5. Clearly, there is plenty of parallelism in this program: sequential execution takes approximately 1200 time steps, compared to approximately 110 for parallel. Both runs executed around 2600 instructions, with the sequential one needing 108 more identity instructions.

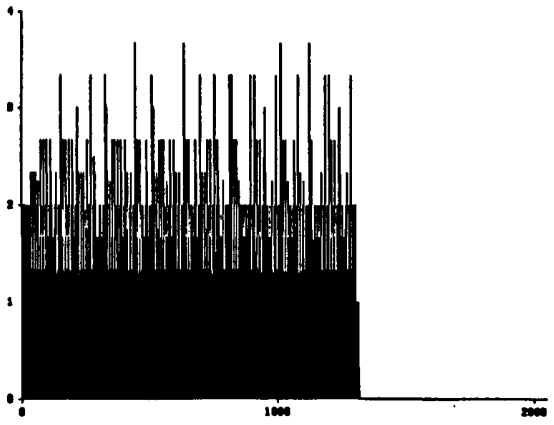

Figure 4: (Fibonacci 10): Alu Operations Profile– Sequential

The source of parallelism in `fibonacci` is in the two recursive calls. The type checker reports that both calls are `pure` and thus do not interfere, and there are no dataflow constraints between them. Hence, the parallel execution simultaneously pursues computation down both branches of the recursion. Incidentally, the profile in figure 5 is exactly like that produced by fibonacci code written in ID and compiled for GITA. In general, FX-87 performs just as well as ID on functional programs that do not use data structures.

3.1.3 Polynomial Division

To close this section on functional examples, we'll look at a somewhat more complicated program that computes the quotient and remainder of polynomial division. (See [Ham88] for source code.) For example, `div-polys` called on a dividend of $x^3 + 3x^2 + 4x + 5$ and a divisor $x + 1$ returns the quotient $x^2 + 2x + 2$ and remainder 3.

Compiling for sequential execution and calling `div-polys` on the above example produced the statistics in figures 6 and 7 and table 3. Similarly, compiling for parallel execution and using the same example resulted in figures 8 and 9 and table 4. The most interesting difference between the two sets of statistics is in the *invocation tree profile.* In the case of sequential execution, the procedure applications within the body of a procedure occur one after another, each one terminating before the next one begins. Therefore, the

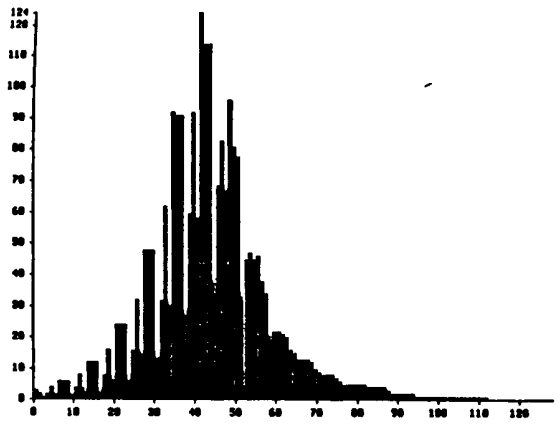

Figure 5: (Fibonacci 10): Alu Operations Profile– Parallel

invocation tree profile represents the depth of invocations versus time, *i.e.*, the number of stack frames on the control stack of a traditional sequential machine versus time. In the case of parallel execution, however, any application within the body of a procedure is allowed to proceed if it is sure to be executed and does not need to be constrained because of its effect. Perusing the code for `div-polys` reveals that there is no effect interference, but there is a conditional which contains a recursive call to `div-polys`. Hence, the *invocation tree profile* for the parallel execution is explained as follows: upon entering `div-polys` for the first time the branch containing the recursive call is selected and many of the invocations entailed therein are quickly executed. The wild spread of invocations is stopped only by conditionals whose predicates are unknown (*i.e.*, no speculative parallelism). The recursive call to `div-polys` encounters exactly that, undetermined conditional execution. Some time later when the predicate finally arrives it sets off another wave of invocations. The *invocation tree profile* shows four such waves, corresponding to the order of the dividend.

The *dynamic operations table* contains a few new categories that deserve explanation. *Fetch* and *Store* are counters for accesses to the standard store that we added to GITA; these accesses are like those in a standard memory architecture in that they do not rely on deferred reads or write-once enforcement. Accesses that *do* rely on this more sophisticated I-structure memory are counted in *I-Fetch* and *I-Store*. This raises the obvious question: why

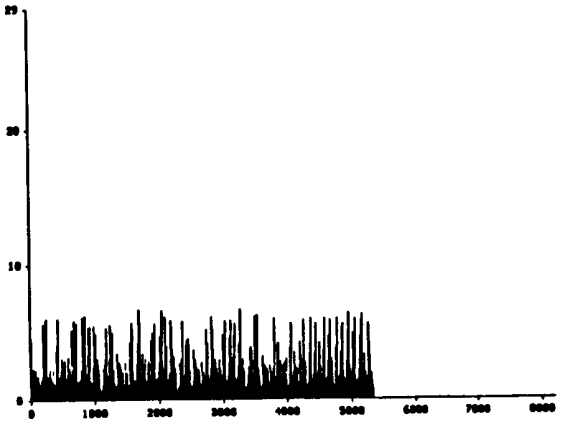

Figure 6: Polynomial Division: Alu Operations Profile– Sequential

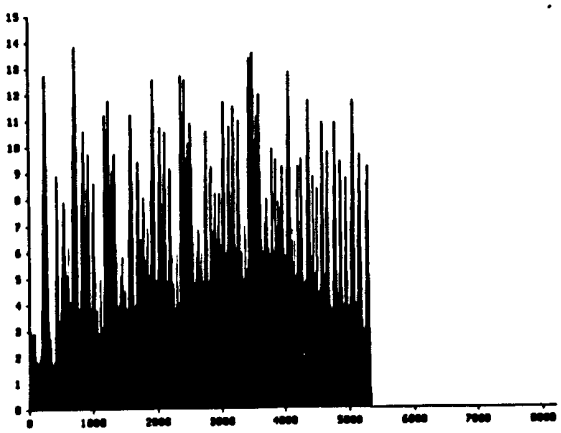

Figure 7: Polynomial Division: Invocation Tree Profile– Sequential

Instruction Type	Count	Percentage
Identity	7,883	49.13
Tag	4,709	27.35
Resource	1,690	10.53
Switch	623	3.88
Fetch	253	1.58
Arithmetic	244	1.52
Store	236	1.47
Address	236	1.47
Closure	105	0.65
I-Store	42	0.26
I-Fetch	21	0.13
Constant	2	0.01
Total	16,044	100
Critical Path		5,342

Table 3: Polynomial Division: Dynamic Operations Mix– Sequential

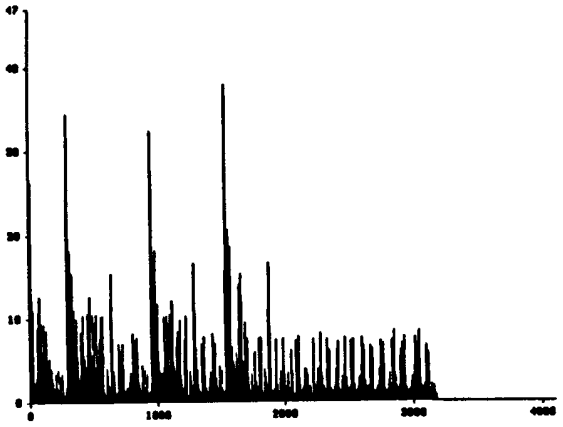

Figure 8: Polynomial Division: Alu Operations Profile– Parallel

are I-structures being used in a program compiled from FX? The answer is that closures are built using I-structures, just as they are in compiled ID programs. There is no reason why regular memory could not be used for this task, except that this would require more changes to the compiler back-end and would sacrifice a small amount of parallelism gained through the non-strictness inherent in argument-passing with I-structures. The category *closures* is just a count of all the instructions related to the construction and manipulation of closures.

Unfortunately, the speedup with parallel execution, about 1.7, is less than phenomenal. Like `factorial`, the computation in this program is severely constrained by the flow of data from one application to the next. However, if we consider a case where some client uses `div-polys` to perform many unrelated polynomial divisions, then, since the effect system assures the compiler that multiple calls to the polynomial division function cannot possibly interfere, the client's requests will be scheduled in parallel. For example, in figures 10 and 11 and table 5, we have run a small program that calls `div-polys` on 4 separate sets of dividends and divisors and puts the results in a list. As we expected, the polynomial divisions completely overlap, and the resulting execution time is essentially the same as that for a single division, even though approximately 4 times as many instructions have been executed.

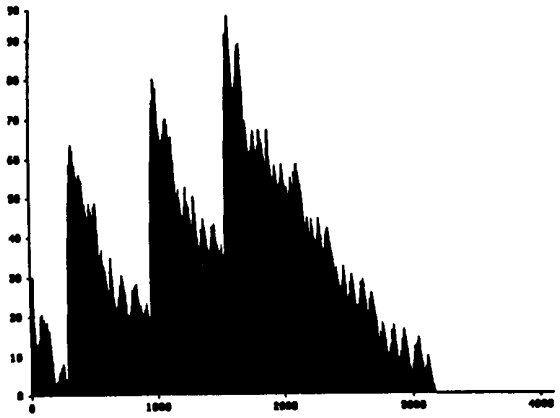

Figure 9: Polynomial Division: Invocation Tree Profile– Parallel

Instruction Type	Count	Percentage
Identity	7,772	48.78
Tag	4,709	29.56
Resource	1,690	10.61
Switch	623	3.91
Fetch	253	1.59
Arithmetic	244	1.53
Store	236	1.48
Address	236	1.48
Closure	105	0.66
I-Store	42	0.26
I-Fetch	21	0.13
Constant	2	0.01
Total	15,993	100
Critical Path		3,183

Table 4: Polynomial Division: Dynamic Operations Mix– Parallel

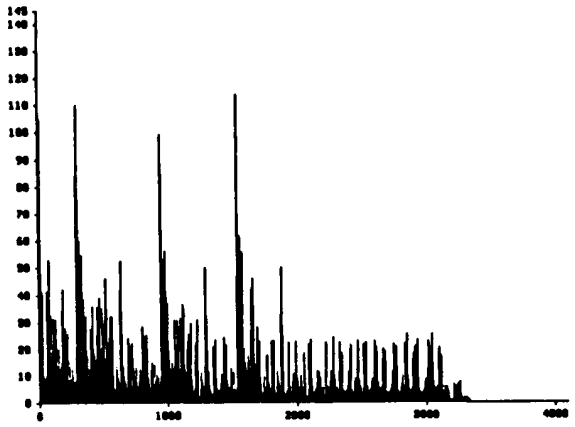

Figure 10: Four Polynomial Divisions: Alu Operations Profile– Parallel

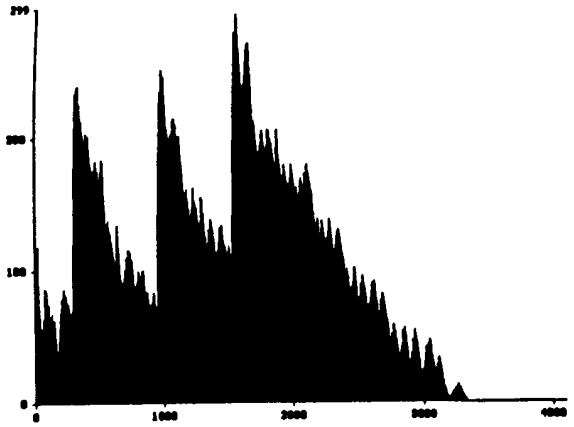

Figure 11: Four Polynomial Divisions: Invocation Tree Profile– Parallel

Instruction Type	Count	Percentage
Identity	26,162	48.82
Tag	15,824	29.53
Resource	5,685	10.61
Switch	2,086	3.89
Fetch	843	1.57
Arithmetic	819	1.53
Store	804	1.50
Address	804	1.50
Closure	350	0.65
I-Store	140	0.26
I-Fetch	70	0.13
Constant	2	0.00
Total	53,589	100
Critical Path		3,341

Table 5: Four Polynomial Divisions: Dynamic Operations Mix– Parallel

3.2 Sequence Matching

DNA sequence matching determines how similar two DNA sequences are using a metric that assigns penalties for base pair insertions, deletions, and substitutions. Since the sequences are from hundreds to thousands of elements long and DNA databases are very large, it is necessary to rely on a computer program to match sequences. In this section, we look at an FX-87 program that finds the best way to align two sequences.

[Ham88] shows an encoding of a DNA sequence matching algorithm in FX. The program takes two vectors of symbols representing two DNA sequences, and, by inserting blanks in either one, produces two new vectors of symbols representing the match with the lowest cost. Cost is assigned for inserting gaps (the longer the gap the greater the cost) and for matching elements that are not the same. Without delving into too much detail, the program has three main parts:

1. Assuming sequence 1 has length x and sequence 2 length y, allocate two new matrices of size $(x+1) \times (y+1)$ and initialize them. We will call these the *match* and *path* matrices.

2. Conceptually, the two sequences and the match matrix are aligned as shown in figure 12. Each element $m_{i,j}$ of the match matrix represents the lowest possible cost of matching the first i elements of sequence 1 with the first j elements of sequence 2. Each element of the *path* matrix $p_{i,j}$ records three things:

   (a) the coordinates of the *match matrix* element that were used to compute $m_{i,j}$ (either $(i-1, j-1)$, $(i, j-1)$, or $(i-1, j)$)

   (b) the length of the current gap.

   (c) the length of the match.

   The elements of the *match* and *path matrices* are filled in by stepping i from 1 to x and j from 1 to y, and filling in $m_{i,j}$ and $p_{i,j}$ as follows:

   (a) Using the value of $m_{i-1,j}$ and the length of the current gap from $p_{i-1,j}$, and assuming we are matching the i^{th} element of sequence 1 against a blank for sequence 2, compute a new cost. Do the

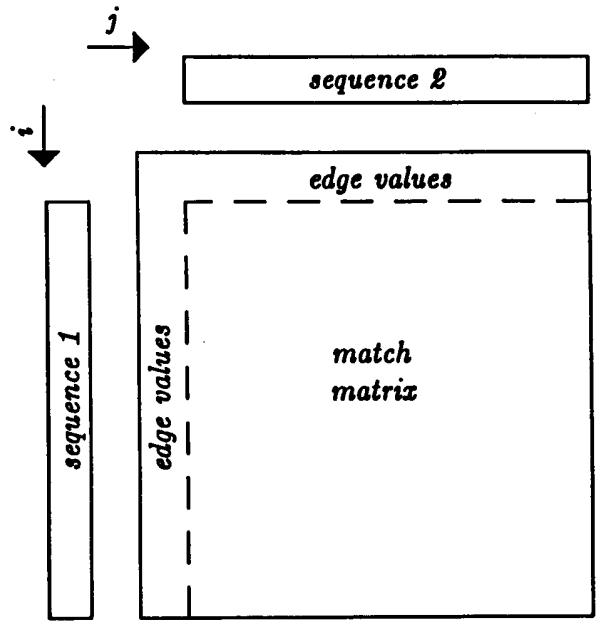

Figure 12: Alignment of sequences with match matrix

same for the indices $(i-1, j-1)$, matching the i^{th} element of sequence 1 against the j^{th} element of sequence 2, and for the indices $(i, j-1)$, matching a blank for sequence 1 against the j^{th} element for sequence 2. The minimum of these three costs is the new value for $m_{i,j}$.

(b) Fill in $p_{i,j}$ to record which of the three new costs computed were used as the value for $m_{i,j}$. Also, record how this choice effects the length of the current gap and the length of the alignment produced thus far.

3. Once the match and path matrices are filled in, use the path matrix to trace back through the decisions that were made to achieve the lowest cost alignment, and construct it.

The important thing to notice is that each element (i, j) in the match and path matrices depends only on the values in the match and path matrices at indices $(i-1, j-1)$, $(i-1, j)$, and $(i, j-1)$, and on element i of sequence 1 and element j of sequence 2. Therefore, it is possible to compute all the elements of each diagonal in parallel.

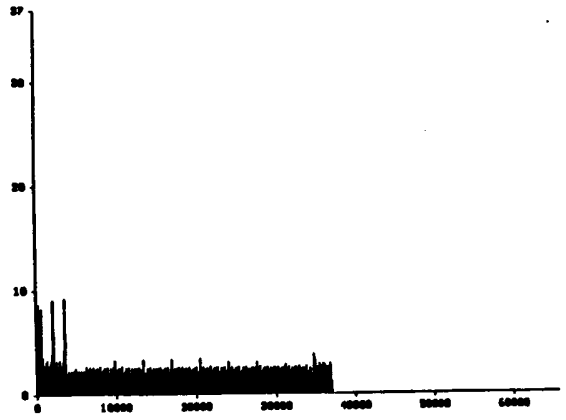

Figure 13: DNA Sequence Matching: Alu Operations Profile– Sequential

3.2.1 Sequential Execution, Parallel Execution, and ID

We begin the analysis of the DNA sequence matching program with the statistics from sequential execution (fig 13, fig 14, table 6). These numbers were produced by comparing C A T A C G C C against A T U U A G C C to produce a best match, C A T A C - G C C against - A T U U G C C

Looking first at the *alu operations profile*, we see that it looks very sequential, except for a few small bursts of activity near the beginning of the computation. These are caused by the many calls to library function `make-vector` entailed in the allocation of the *match* and *path matrices*. Even in forced sequential execution, library routines are still allowed to exploit any parallelism within their bodies. `Make-vector` is a library routine that takes two arguments, an integer and some initial element, allocates a vector of the length given by the first argument, and fills it with the second argument. All the memory writes to fill the new vector can, and do, run in parallel.

The *invocation tree profile* shown in figure 14 clearly reflects the three phases of the computation: in the first 4500 time steps, there are two large spikes for all the procedures that allocate and initialize the large matrices. Then around time step 5000, there are two smaller spikes for setting the edges of the matrices to properly reflect the edge conditions. From about 6000 to 35000, the computation settles into a steady ascent topped by eight smaller ascents. This pattern reflects the nested *do* loops that step the two indices through each element of 8×8 submatrices of the *match* and *path matrices*.

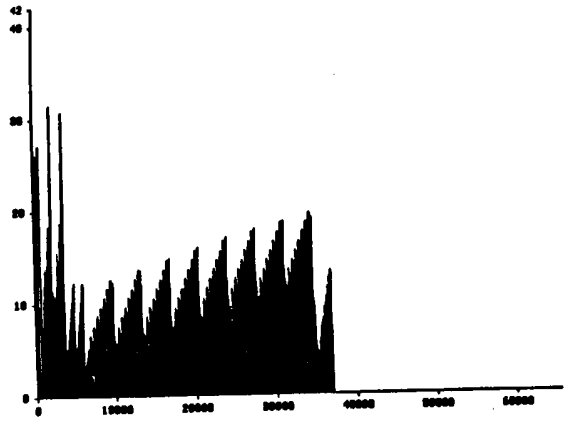

Figure 14: DNA Sequence Matching: Invocation Tree Profile– Sequential

Instruction Type	Count	Percentage
Identity	41,652	44.77
Tag	26,232	28.20
Resource	8,625	9.27
Switch	5,282	5.68
Arithmetic	4,429	4.76
Closure	2,245	2.41
Fetch	1,604	1.72
Store	1,138	1.22
Address	1,138	1.22
I-Store	424	0.46
I-Fetch	255	0.27
Constant	2	0.00
Total	93,026	100
Critical Path		37,074

Table 6: DNA Sequence Matching: Dynamic Operations Mix– Sequential

As stated earlier, *do* is desugared into a tail-recursive function, where each iteration is handled by making a recursive call. So in the profile, the overall ascent is the result of the recursive calls made by the outer *do* loop, and the eight smaller ascents the result of recursive calls by the inner one. Finally, the last spike is for the procedures that trace back through the *path matrix* and construct the result.

Before trying the parallel run, we will briefly examine the way DNA sequence matching can be approached in the functional language ID. At first, the above algorithm seems inherently side-effecting, and in a language like FX-87, where the allocation and initialization of data structures are an atomic operation, it is indeed side-effecting. In ID, however, the programmer explicitly allocates an uninitialized I-structure and then explicitly writes values to its elements; hardware restricts him to writing each element only once. As an example, here is code, written in both ID and FX-87, that allocates a two element vector and puts the value 1 in the first element and the value 2 in the second.

```
{id-vector = array (0,1)        % new vector, uninitialized
 id-vector [0] = 1
 id-vector [1] = 2}


(let ((fx-vector ((proj make-vector @R1) 2 0)))    ; new vector with
                                                   ;  each element
                                                   ;  initialized to
                                                   ;  0.
  (vector-set! fx-vector 0 1)
  (vector-set! fx-vector 1 2))
```

Thus, in ID, the first step in the sequence matching algorithm allocates uninitialized matrices. The second step iterates over all their elements, computing new values based on values already written, and writing each element once and only once. The third and final step is just as before: trace back over the *path matrix*, constructing the result.

Looking at the statistics in figures 15, 16, 17, and table 7, we can see that the ID version of DNA sequence matching discovered the parallelism available in the algorithm. The *alu operations profile* shows a large start-up

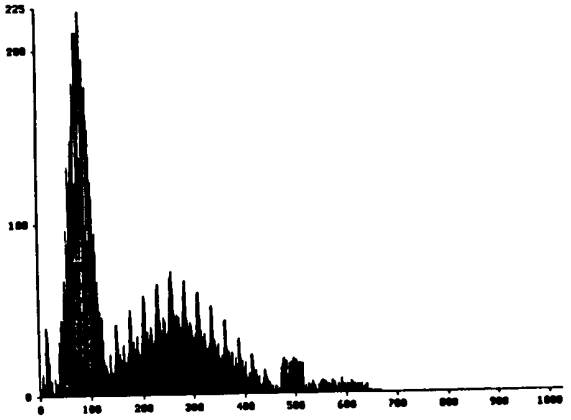

Figure 15: DNA Sequence Matching: Alu Operations Profile– ID

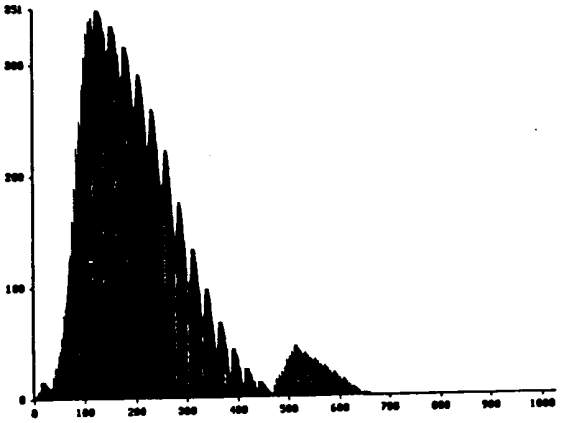

Figure 16: DNA Sequence Matching: Invocation Tree Profile– ID

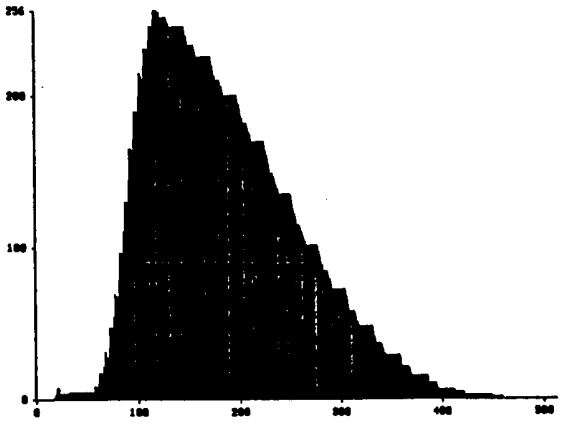

Figure 17: DNA Sequence Matching: Deferred I-fetches Profile– ID

Instruction Type	Count	Percentage
Identity	8,950	39.41
Tag	5,545	24.42
Resource	2,121	9.34
Arithmetic	1,816	8.00
I-Fetch	1,474	6.49
Switch	1,168	5.14
I-Store	728	3.21
D	407	1.79
Loop	258	1.14
Address	163	0.72
Constant	75	0.33
Bounds	4	0.02
Total	22,709	100
Critical Path		667

Table 7: DNA Sequence Matching: Dynamic Operations Mix– ID

transient in the first 120 time steps, where the loops unroll and all the I-structure reads take place. Then we see a series of spikes where the diagonals of the matrices are filled in. Finally, there is a small burst of activity to search the *path matrix* and construct the answer. Note that even though all the I-structure reads take place early in the computation, along with the unfolding of the loop indices, most of them cannot be immediately satisfied and must be put in deferred read queues in the memory. This is shown in the *total deferred i-fetches profile* in fig 17. The *dynamic operations table* shows considerably fewer instructions have been executed than in the FX-87 version. This is mostly accounted for by the fact that ID makes use of special GITA mechanisms for efficient tail-recursion, and the FX-87 compiler does not.

Returning now to FX, the statistics for parallel execution in figures 18 and 19 and table 8 show a 3.7 speedup over the earlier sequential run, but the shapes of the *alu operations profile* and the *invocation tree profile* are essentially unchanged. FX was able to discover parallelism in two places. First, notice that the two large spikes near the beginning of the *invocation tree profile* for sequential execution are combined into one very large spike in parallel execution. This is because the allocations of the *match* and *path matrices* do not interfere and have no data dependencies, and are therefore allowed to proceed in parallel. Second, FX discovered some parallelism in the body of the main loop, thereby shortening the time spent on computing each element of the matrices.

Unfortunately, the potentially largest gain, computing the elements of each diagonal in parallel, was not exploited. The problem here is fundamental to the idea of static interference calculation, and is best explained by examining two constraints put on the programmer by FX. The first constraint was alluded to earlier and concerns the allocation of data structures. All the built-in functions in FX-87 that dynamically allocate new storage also initialize it to some value provided by the caller. For the programmer, allocation and initialization are an atomic operation. So in the case of the DNA algorithm, wherein the matrices must be filled in based on other elements already computed, a procedure must be called that allocates a new matrix (filling it with some initial value) and then iterates over its elements, reading and writing them as necessary. The operative word here is *writing*; this implies that the matrices must be allocated in a mutable region.

The second constraint is a simple issue of static versus dynamic infor-

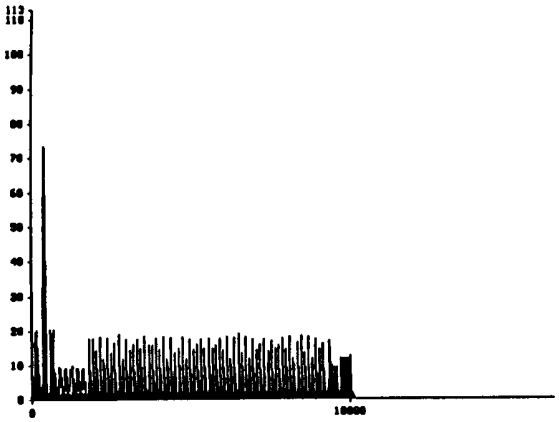

Figure 18: DNA Sequence Matching: alu operations profile– parallel

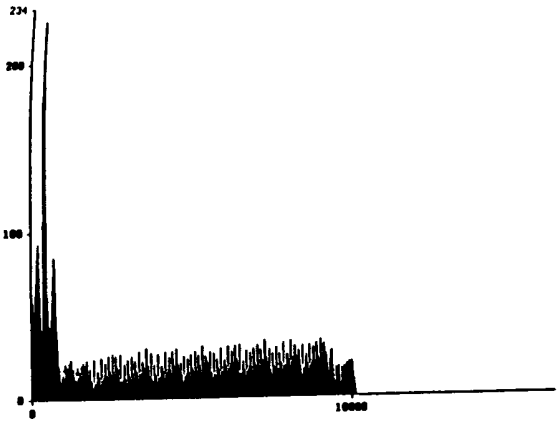

Figure 19: DNA Sequence Matching: invocation tree profile– parallel

Instruction Type	Count	Percentage
Identity	39,825	43.49
Tag	26,232	28.65
Resource	8,625	9.42
Switch	5,922	6.47
Arithmetic	4,157	4.54
Closure	2,245	2.45
Fetch	1,604	1.75
Store	1,138	1.24
Address	1,138	1.24
I-Store	424	0.46
I-Fetch	255	0.28
Constant	2	0.00
Total	91,567	100
Critical Path		10,145

Table 8: DNA Sequence Matching: Dynamic Operations Mix– parallel

mation. In FX a *region* is a static concept, whereas memory allocation is a dynamic one. In other words, if the programmer wants to dynamically allocate matrices, as in the DNA example, then by the way regions are expressed he is forced to repeatedly allocate in the same statically declared regions. For example, consider the function `make-vector`. When the programmer uses this function, he must project it over some region (*e.g.*, `((proj make-vector @R1) 10 0)`). Every element in a vector created with `make-vector` is in the same region, and a write to any element will, by the FX effect system, interfere with a write or read to any other element, even if the indices are different. Saying that each element is in a different *region* forces interference calculations to leave the realm of static computation.

Considering these two constraints, it is now clear why FX failed to exploit some of the parallelism exposed by ID. By constraint 1 we were forced to put the *match* and *path matrices* in a mutable region, and by constraint 2, every write to any element of a matrix interfered with every other read or write to that matrix. Hence, each element had to be updated sequentially, making the resulting graphs look very similar to those for sequential execution. Later, in the section on parallel operators, we will try to address this problem, but for now we will look at some other optimizations.

3.2.2 Procedure Invocation Barriers

In Section 2, we said that an `effect-synchronizer` was compiled so that the return value of a procedure was delayed until the procedure completed all of its side-effects. Furthermore, we added that if application A precedes application B in sequential execution and A and B interfere, then a return value from A could be used as an indicator that B may proceed. This puts an unnecessary barrier, however, to parallelism between expressions in different procedures. Consider this example: procedure X and procedure Y have latent effects that interfere, and procedure Y contains a significant amount of `pure` computation. Procedure Z makes a call to X, then to Y. Under our old method of compilation, the body of Z would be compiled so that Y is not invoked until X has returned. But there is no reason why the `pure` computation in Y, or for that matter any computation in Y that does not interfere with the latent effect of X, cannot be started before procedure X finishes. In an attempt to reduce the barrier to parallelism between procedures, we introduce a new method of compilation:

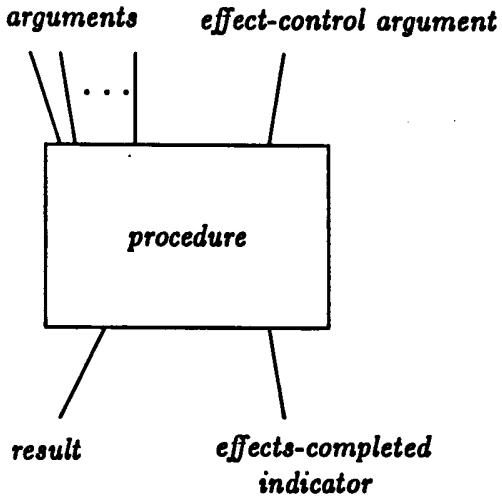

Figure 20: New procedure model

Compilation Method 3– Removing Procedural Barrier: The problem with compilation method 2 is that after invoking a procedure, we have no way of preventing the effects in the body of that procedure. Therefore, we must be careful that no procedure is invoked before its effects can occur. Compilation method 2 also has the dual restriction that there is no way for a procedure to indicate that its effects have completed other than by returning a value. We can partially correct these unnecessary restrictions by compiling each procedure as indicated in figure 20.

Now, in addition to its regular arguments, every procedure takes an *effect-control argument*, which is used by a caller to indicate that a called procedure can go ahead and execute side-effecting operations. Similarly, in addition to returning a value, every procedure will also return an *effects-completed indicator* to indicate that all of its effects have completed. Note that a procedure may return a value before or after indicating that its effects have completed.

The difference between compilation methods 2 and 3 is best viewed as an issue of atomicity. In method 2 procedures were atomic units. We could start them and receive their results, but there was no way to look inside them and control when their effects took place. In method 3 we have added an extra input and output that are used as effect control linkage between caller

and callee; we have separated the flow of data from the flow of effect control, and thereby avoided the cost of synchronizing data and effect information at procedure boundaries. As an obvious generalization of method 3, one might suggest that we pass to every procedure an extra argument for every one of the region constants appearing in the user's program, and likewise return an indicator for every region constant. In this way, the flow of effect information across procedure boundaries would be as unconstrained as within a procedure body. Unfortunately, this more aggressive approach runs into complications when we consider abstraction over regions and effects. For example:

```
(define increment-ref
   (plambda ((r region))
      (lambda ((a-ref (ref int r)))
         (set a-ref (+ 1 (get a-ref))))))
```

When we compile **increment-ref** we will notice that the **get** and **set** expressions have effects on region **r**. The **set** expression will be forced to follow the **get**, which should be delayed until the arrival of one of the effect-control arguments. We have an effect-control argument for every static region constant, but there is no way for the compiler to know which static regions constants are contained in region variable **r**. To overcome this lack of information, we have to move to runtime abstraction and projection over regions and effects. Although there is nothing conceptually difficult about doing this, it would require fundamental changes to the compiler and may involve considerable runtime overhead.

The issue of atomicity is of critical importance when we try to compile the FX library under method 3. The FX-87 effect system relies on the atomicity of its basic storage allocation procedures. Consider the library routine for function **new**:

```
(define new
  (plambda ((r region))
    (plambda ((t type))
      (lambda ((y t))
         (let ((empty-ref
                  (#inline (primitive-make-vector 1))))
            (#inline (primitive-vector-set! empty-ref 0 y))
            empty-ref)))))
```

Suppose we compile this library procedure under method 3, exactly as described above. Then **new** can allocate an empty structure, quickly return it, and then initialize it some time later. Consider the following caller of new:

```
(let ((new-ref ((proj new @R1) 1)))
  (get new-ref))
```

There is not a problem in this case. The **get** expression has effect (**read** **@R1**) and the **new** expression has effect (**alloc** **@R1**). In method 3 these two effects interfere, therefore the **get** will will wait for the effects-completed indicator from the **new**. But consider the following case:

```
(let ((new-ref ((proj new @=) 1)))
  (get new-ref))
```

The **new** expression in this case is **pure**. Therefore, the **get** will not interfere; it will not wait for the effects-completed indicator from the **new**, and may therefore try to read an uninitialized location. To prevent this problem, we compile the FX library by method 3, but add the additional constraint that no procedure return a value before its effects have completed.

Now, we can interpret the statistics from the DNA sequence matching code compiled under method 3 (fig. 21, 22, table 9). Since effect constraints no longer present a barrier to procedure invocation, we would expect a large

Figure 21: DNA Sequence Matching: Alu Operations Profile– Method 3

startup transient in which the loops unfold, and all the procedures that are sure to execute are invoked. The *alu operations profile* and the *invocation tree profile* show exactly what we would expect. After the huge burst of operations at the beginning, however, the computation reverts to the generally sequential task of handling one matrix element after another, and slowly terminating the hundreds of procedures that have been invoked. Unfortunately, the critical path is only slightly less than before. We knew that this new method of compilation would not overcome the problem discussed at the end of section 3.2.1; however, we had hoped that by quickly exposing the pure computation in the body of each iteration, we might shorten the time spent on each element of the matrix, and thereby shorten the overall execution time. The problem is that, in this program, the body of the main loop is dominated by sequentialized, side-effecting operations. Indeed, when we introduced large amounts of pure computation into the body of the loop, method 3 performed significantly better than method 2.

The *dynamic operations table* (table 9) shows approximately a 50 percent increase in the number of instructions executed over method 2. Most of these are accounted for by the overhead of handling an extra argument and an extra return value for each procedure. Specifically, there is a large increase in the number of *i-stores* and *i-fetches*, because we used *I-structures* as part of the new procedure linkage.

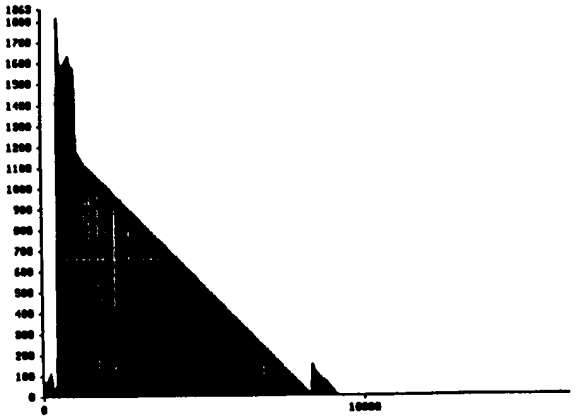

Figure 22: DNA Sequence Matching: Invocation Tree Profile– Method 3

Instruction Type	Count	Percentage
Identity	66,595	44.42
Tag	29,444	19.64
Resource	14,019	9.35
I-Store	11,212	7.48
I-Fetch	8,381	5.59
Switch	7,901	5.27
Arithmetic	4,157	2.77
Closure	3,858	2.57
Fetch	2,074	1.38
Store	1,138	0.76
Address	1,138	0.76
Constant	2	0.00
Total	149,919	100
Critical Path		9,201

Table 9: DNA Sequence Matching: Dynamic Operations Mix– Method 3

3.2.3 Futures in FX

In the last section, when we were discussing the FX library and atomicity, we mentioned returning structures before they were initialized and showed why this would not work for structures in the immutable region. The problem was that in the FX effect system, allocating and referencing an immutable structure were **pure** operations and therefore were only constrained by the flow of data. We could not allow a built-in procedure to return an immutable structure before initializing it, lest some other procedure might try to read one of the uninitialized locations. There was no problem with mutable structures; they were handled correctly by the FX effect system and the effects-completed indicator.

In this section, by using GITA's deferred read mechanism to handle immutable structures, we allow the built-in FX allocation procedures to return *futures* [Hal85], *i.e.* uninitialized locations that will eventually be filled in. The goal is to expose parallelism in the handling of pointers. Consider a situation where procedure A allocates and initializes a structure and then returns it to procedure B. Suppose that procedure B then performs many operations on the newly allocated structure without accessing it, such as storing it in some other structure or passing it to some other procedure. There is opportunity for parallelism in this example: if procedure A quickly returned a pointer to the new structure, then procedure B could store it and pass it while procedure A initializes it. Of course, procedure A must have some way of notifying procedure B that the initialization has completed, so that procedure B can begin accessing the new structure. By including *futures* in the FX implementation, we can exploit this sort of parallelism.

> **Compilation Method 4– Futures:** This is like compilation method 3, except that we compile the FX library without restrictions; in other words, we allow FX built-in functions to return values before their effects have completed, thereby exposing the structure allocating procedures as nonatomic operations. This means that early in the execution of any program, all the memory allocating subroutines that are sure to be executed will be invoked and will thereupon return pointers to the newly allocated, but uninitialized, structures. In the case of mutable structures, premature accesses to uninitialized locations will not occur, because all the accessing procedures will interfere with the allocation

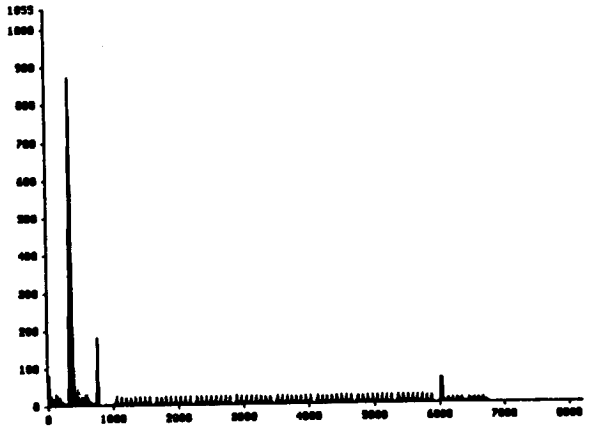

Figure 23: DNA Sequence Matching: alu operations profile– futures

procedure and will therefore wait for an *effects-completed* indicator. In the case of immutable structures, structure reads may occur before initialization is complete, but they will be handled by GITA's deferred read mechanism.

The results for the DNA sequence matching example using method 4 are shown in figures 23, 24, and table 10. Comparing these statistics to those for method 3, we see that the *dynamic operations table* shows considerably fewer identity instructions in method 4. This is accounted for by the removal of restrictions on the library routines. The *alu operations profile* for method 4 shows a larger initial spike, corresponding to the extra instructions for returning and manipulating uninitialized structures; furthermore, we can see that using *futures* has shaved a couple thousand steps off the total execution time resulting in a 1.3 speedup over method 3 and 5.4 speedup over sequential execution.

3.2.4 Parallel Operators

In section 3.2.1 we discussed two constraints on the FX programmer: the static nature of *regions* and the atomicity of allocation and initialization. We showed how these two constraints prevented the FX effect system from exploiting the diagonal parallelism in the DNA sequence matching program. In this section we want to explore how we can overcome these constraints.

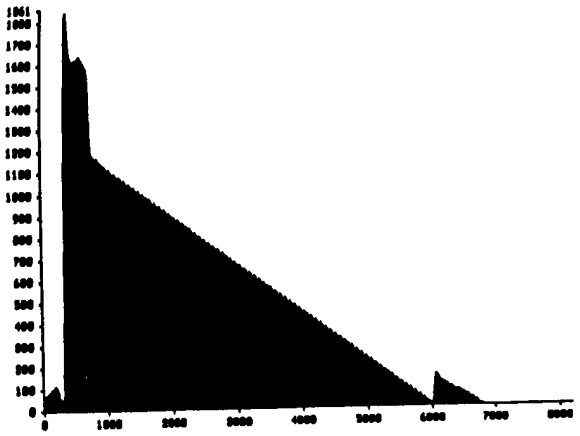

Figure 24: DNA Sequence Matching: invocation tree profile– futures

Instruction Type	Count	Percentage
Identity	61,785	42.75
Tag	29,444	20.37
Resource	14,019	9.70
I-Store	11,212	7.76
I-Fetch	8,273	5.72
Switch	7,901	5.47
Arithmetic	4,157	2.88
Closure	3,858	2.67
Fetch	1,604	1.11
Store	1,138	0.79
Address	1,138	0.79
Constant	2	0.00
Total	144,531	100
Critical Path		6,850

Table 10: DNA Sequence Matching: Dynamic Operations Mix– futures

At first we might consider making *regions* a dynamic concept. For example, we could overcome the problems in the sequence matching example by dynamically assigning a different region for every element of a vector. Unfortunately, following this line a reasoning eventually leads to the point where the FX effect system becomes nothing more than an expression of the runtime manipulation of locks. The static nature of the language's effect system is lost to complicated and costly runtime calculations. This seems like an unattractive approach.

Another alternative is to change the way data structures are allocated and initialized. Remember that in FX the programmer is forced to regard allocation and initialization as an atomic operation. In section 3.2.3 we exposed the nonatomicity of these operations, but only at the implementation level; the programmer is still not allowed to allocate a structure without giving some initialization value. Suppose we removed this restriction; suppose we allow the programmer to write something like `((proj make-vector @=) 10)`. This would return an uninitialized, immutable 10-element vector. Since the vector is immutable, we would be forced to have runtime overhead to insure that each element is written (initialized) only once. Also, reads to uninitialized locations would have to be deferred until their value arrives. It is clear that we have simply introduced I-structures into FX, along with all the runtime overhead they entail. Furthermore, I-structures do not have a sequential semantics [Arv87c], so we have also destroyed the sequential semantics of FX.

Another less drastic change is to introduce parallel operators. Parallel operators do not change the fact that allocation and initialization are an atomic operation, but they do give the programmer more control over how structures are initialized. Consider adding `make-vector-with-init` to the language:

```
make-vector-with-init : (poly ((r region))
                          (poly ((t type))
                            (subr (alloc r)
                                  (int (subr pure (int) t))
                                  (vectorof t r))))
```

`Make-vector-with-init` takes two arguments: an integer and a `pure` procedure. It then allocates a vector of length given by the first argument and

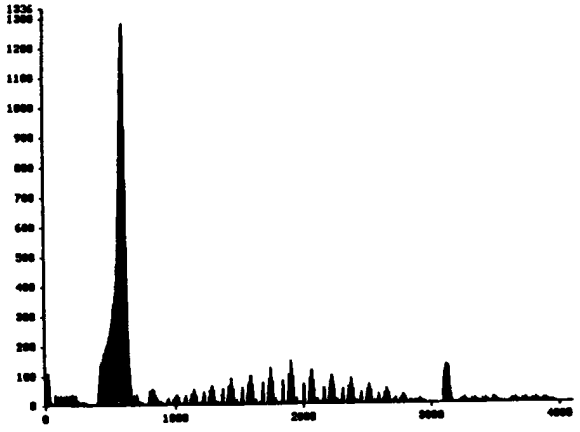

Figure 25: DNA Sequence Matching: Alu Operations Profile– Parallel Operator

fills each element with the result of calling the second argument on the element's index. `Make-vector-with-init` is true to the spirit of FX in that it atomically allocates and initializes a vector, and it has sequential semantics. Of course, since the second argument to `make-vector-with-init` is a pure function the invocations for all of the elements can proceed in parallel.

We have rewritten the DNA sequence matching code using this new parallel operator. Figures 25, 26, and table 11 show the results of running code compiled under method 3. The *alu operations profile* and the *invocation tree profile* have the same general shape as those from the ID run, albeit considerably stretched. Compared to the other methods of compiling FX, the *dynamic operations table* shows more instructions in every category. The is accounted for by the fact that much of the computation that goes on inside of `make-vector-with-init` to compute elements for the *match matrix* has to be repeated for each element of the path matrix. It is possible that a more clever encoding could have avoided this repetition, but the really important point is that this approach has exploited the parallelism along the diagonals of the *match* and *path matrices.*

Unfortunately, parallel operators, such as `make-vector-with-init`, violate the spirit of FX-87 in two ways. First, they introduce data parallelism into the language. Until now we have used the FX-87 effect system to make control decisions that afford parallelism (*i.e.,* decisions about procedure invocation). GITA, with its ability to pursue concurrently multiple branches

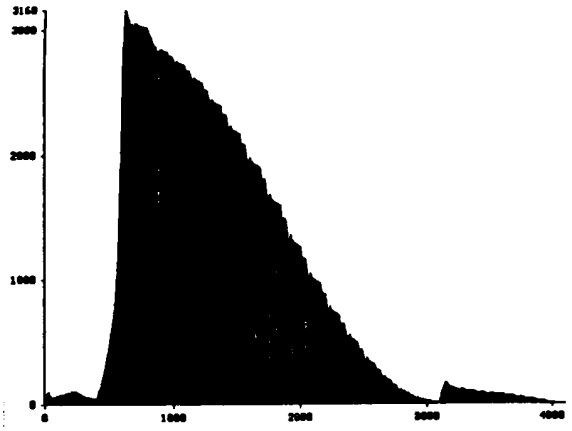

Figure 26: DNA Sequence Matching: Invocation Tree Profile– Parallel Operator

Instruction Type	Count	Percentage
Identity	93,692	42.64
Tag	35,576	16.19
Resource	19,648	8.94
I-Store	18,204	8.28
Switch	17,310	7.88
I-Fetch	14,466	6.58
Closure	9,569	4.35
Arithmetic	6,835	3.11
Fetch	2,380	1.08
Store	1,032	0.47
Address	1,032	0.47
Constant	2	0.00
Total	219,746	100
Critical Path		4,080

Table 11: DNA Sequence Matching: Dynamic Operations Mix– Parallel Operator

of procedure invocation, was well suited to our approach. Now, however, we have introduced an operator with clear-cut, homogeneous parallelism. We would like a SIMD-style machine, wherein a front processor issues instructions to a large collection of slave processors, each with its own memory. To see this more clearly consider the DNA example. After the large start-up spike, we see a series of smaller spikes that grow progressively larger and then smaller. Each of these corresponds to the calls to `make-vector-with-init` to create a match and path diagonal. The width of these spikes results from the execution of all the procedures to compute the value for each element of the new vector. The whitespace between spikes reflects time in which GITA is passing values from one iteration to the next. On a SIMD machine, we could assign one element per processor and issue a single stream of instructions to properly fill the vector. Iterations would be handled by the front-end processor. The overhead of this style of computation would be markedly less on a SIMD machine.

The second and more disturbing reason why parallel operators seem to violate the spirit of FX is that they require the programmer to think about parallelism. The programmer has written the main loop of the program to iterate over the diagonals of the matrices, using `make-vector-with-init` to build each diagonal. In other words, the programmer was aware of parallelism in the algorithm and wrote code with that in mind. A major goal of the effect system was to relieve the programmer of the burden of reasoning about parallelism, and now we have forced him to reshoulder that burden.

Before ending this section, we should point out that there may be a way for the programmer to use a parallel operator without considering the possibilities for parallelism in his program. Consider this: A parallel operator gives the programmer the power to initialize a vector as he pleases. Why, then, does the FX programmer have to think about parallelism and the ID programmer does not? The answer is that in ID we could define elements of a newly allocated vector to depend on other elements of that vector. But by passing a recursive function to `make-vector-with-init` we can do the same thing in FX:

```
(define init
   (lambda ((i int))
      (the pure int
         (if (= i 0)
             0
             (+ i (init (- i 1)))))))

(define new-vector ((proj make-vector-with-init @=) 4 init))
```

`New-vector` in the above example will be `#(0 1 3 6)`. That is, for each element v_i of the vector, $v_i = i + v_{i-1}$. For the DNA example we could have written a large recursive function defining each element of the *match* and *path matrices* in terms of its neighbors and then passed this to the parallel operator. The problem is that the initialization function may get called many times on the same argument. The solution to this problem is memoization, *i.e.*, remembering results and returning previously computed results when called on the same arguments. Note that it is always possible to memoize the initialization function since it must be `pure`. The way memoization is performed will greatly affect the results of this approach; we are currently investigating the various alternatives.

3.3 Scheme Interpreter

Our final benchmark is a scheme interpreter written in FX-87. Scheme [Ree86] is an imperative, sequential dialect of lisp. The FX-87 code appearing in [Ham88] is based on the interpreter implemented and discussed in [Abe85]. This benchmark presents a particularly difficult challenge for systems designed to exploit parallelism. The computation is heterogeneous and strewn with side-effecting operations. It would be impossible to write this code in ID, since the program uses interactive I/O and other side-effects. Furthermore, simple parallel operators, effective in homogeneous, scientific computation, are of little help here.

The computation begins by initializing an environment of built-in functions. Then a prompt is printed and a scheme expression is read from the user. The program parses the user's expression and then steps recursively

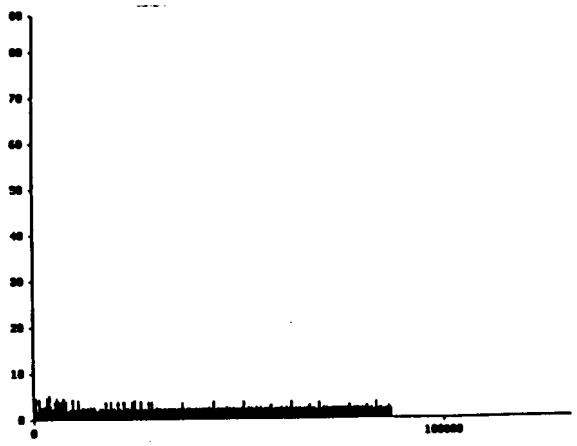

Figure 27: Scheme Interpreter: Alu Operations Profile– Sequential

down the parse tree to perform the evaluation. Finally, the answer is printed, the user is prompted for another expression, and the whole process begins again. For those familiar with lisp interpreters, this is just a read-eval-print loop.

In the examples that follow, the scheme interpreter was started and given the definition for `fibonacci`. It was then asked to evaluate `(fibonacci 5)`. Since scheme is a sequential language, we cannot expect the FX implementation to overlap the evaluation steps of the scheme interpreter. In other words, interpreting a scheme program means evaluating one expression after another, where any expression can have an effect on the store. Since FX has no information about effects in the scheme programs it is interpreting, it is forced to evaluate scheme expressions sequentially. The only opportunity for parallelism is within each evaluation.

The statistics for sequential execution (figures 27, 28, and table 12) are uninteresting except for the *invocation tree profile*. In all the other examples the number of pending procedures at the end of the computation was zero; in this example it is not. This is accounted for by the fact that the top level read-eval-print loop is a tail recursive function that loops forever, continually prompting the user for an expression and then evaluating it.

Pulling out all the stops, we compiled the interpreter by method 4 (futures) and ran it to produce the statistics in figures 29, 30, and table 13. This run executed 1.7 times as many instructions and produced a speedup of

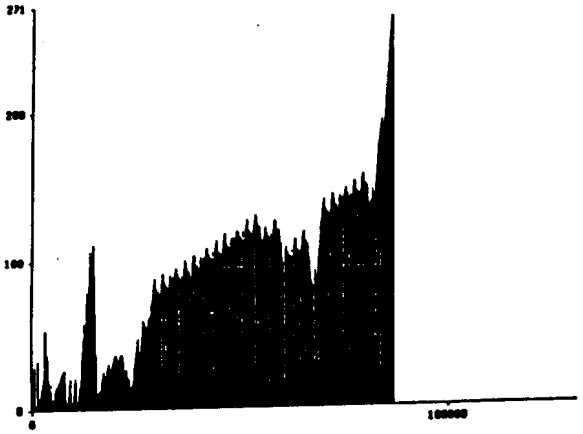

Figure 28: Scheme Interpreter: Invocation Tree Profile– Sequential

Instruction Type	Count	Percentage
Identity	103,813	46.00
Tag	64,515	28.59
Resource	22,698	10.06
Switch	15,813	7.01
Closure	4,793	2.12
Arithmetic	3,983	1.76
Fetch	3,648	1.62
Store	1,817	0.81
Address	1,817	0.81
I-Store	1,402	0.62
I-Fetch	1,082	0.48
I/O	268	0.12
Other	19	0.01
Total	225,668	100
Critical Path		88,000

Table 12: Scheme Interpreter: Dynamic Operations Mix– Sequential

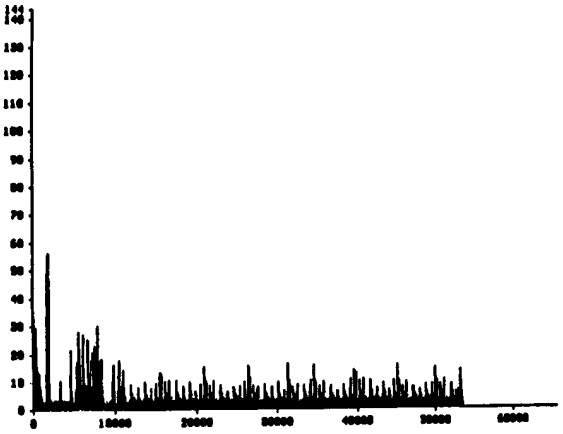

Figure 29: Scheme Interpreter: Alu Operations Profile– Futures

about 1.7. The *invocation tree profile* is comparitively choppy, reflecting the waves of invocations that follow the arrival of a predicate value at a conditional. Also, we notice that, unlike the sequential run, the number of pending procedures at the end of computation is zero. This is because we modified the top level read-eval-print loop to loop only three times. We were forced to do this because running an infinitely tail recursive function compiled under methods 3 or 4 swamps the machine with procedure invocations. Remember that with the new procedure linkage introduced in method 3 nothing prevents a procedure that is sure to be executed from being invoked. Therefore, an infinite tail recursive loop procedure sets off a never-ending wave of invocations.

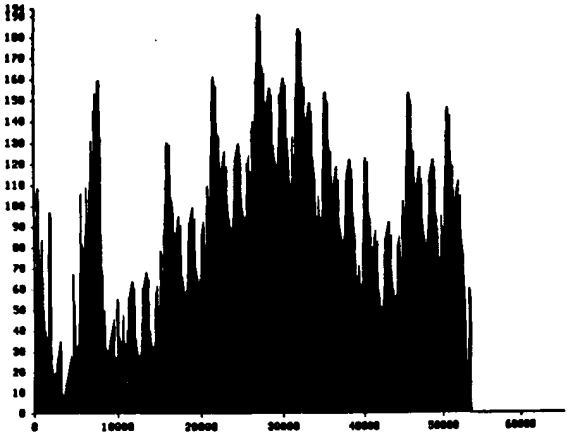

Figure 30: Scheme Interpreter: Invocation Tree Profile– Futures

Instruction Type	Count	Percentage
Identity	165,820	43.51
Tag	68,784	18.05
Resource	40,533	10.64
I-Store	36,413	9.55
I-Fetch	26,807	7.03
Switch	23,591	6.19
Closure	7,598	1.99
Arithmetic	3,994	1.05
Fetch	3,650	0.96
Store	1,819	0.48
Address	1,819	0.48
I/O	270	0.07
Other	19	0.00
Total	381,118	100
Critical Path		53,500

Table 13: Scheme Interpreter: Dynamic Operations Mix– Futures

References

[Abe85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge MA, 1985.

[Arv87a] Arvind and R. Nikhil, *Executing a Program on the Tagged-Token Dataflow Architecture*, MIT/LCS/CSG Memo 271, March 1987.

[Arv87b] Arvind, Rishiyur S. Nikhil and Keshav K. Pingali, *I-structures: Data Structures for Parallel Computing*, MIT/LCS/CSG memo 269, February 1987.

[Arv87c] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali, *Id Nouveau Reference Manual, Part II: Operational Semantics*, MIT/LCS/CSG, April 1987.

[Gif87] David K. Gifford, Pierre Jouvelot, John M. Lucassen and Mark A. Sheldon, *FX-87 Reference Manual*, MIT/LCS/TR-407, September 1987.

[Hal85] Robert H. Halstead, *Multilisp: A Language for Concurrent Symbolic Computation*, ACM TOPLAS, pp. 501-538, October 1985.

[Ham88] R. Todd Hammel, *An FX-87 Compiler for a Dataflow Machine*, to be completed September 1988.

[Joh86] Thomas Johnsson, *Lambda Lifting: Transforming Programs to Recursive Equations*, Programming Methodology Group, Department of Computer Science, Chalmers University of Technology, S-412 96 Goteborg, Sweden, 1986.

[Jou88] Pierre Jouvelot and David K. Gifford, *The FX-87 Interpreter*, MIT/LCS/PSRG, to appear in the 1988 IEEE International Conference on Computer Languages, February 1988.

[Luc87] John M. Lucassen, *Types and Effects– Towards the Integration of Functional and Imperative Programming*, MIT/LCS/TR-408, August 1987.

[Nik86] Rishiyur S. Nikhil, *Id Nouveau*, MIT/LCS/CSG memo 265, July 1986.

[Ree86] Jonathan Rees and William Clinger (editors), *Revised Report on the Algorithmic Language Scheme*, MIT/AI Memo 848a, September 1986.

[Tra86a] Kenneth R. Traub, *A Compiler for the MIT Tagged-Token Dataflow Architecture*, MIT/LCS/TR-370, August 1986.

[Tra86b] Kenneth R. Traub, *A Dataflow Compiler Substrate*, MIT/LCS/CSG Memo 261, March 1986.