A study of memory references in a data flow environment

Sharilyn Ann Thoreson
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation
https://lib.dr.iastate.edu/rtd/7354

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in “sectioning” the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.

University Microfilms International
300 N. ZEEB ROAD, ANN ARBOR, MI 48106
18 BEDFORD ROW, LONDON WC1R 4EJ, ENGLAND
THORESON, SHARILYN ANN

A STUDY OF MEMORY REFERENCES IN A DATA FLOW ENVIRONMENT

Iowa State University

Ph.D. 1980

University Microfilms International
300 N. Zeeb Road, Ann Arbor, MI 48106
18 Bedford Row, London WC1R 4EJ, England
A study of memory references in a data flow environment

by

Sharilyn Ann Thoreson

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

Signature was redacted for privacy.
In Charge of Major Work

Signature was redacted for privacy.
For the Major Department

Signature was redacted for privacy.
For the Graduate College

Iowa State University
Ames, Iowa

1980
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER I. INTRODUCTION</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motivation</td>
<td>1</td>
</tr>
<tr>
<td>Fundamentals of Data Flow Computers</td>
<td>2</td>
</tr>
<tr>
<td>The Data Flow Model</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER II. LOCALITY</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spatial Locality</td>
<td>12</td>
</tr>
<tr>
<td>Temporal Locality</td>
<td>13</td>
</tr>
<tr>
<td>Conclusions</td>
<td>21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER III. RESTRUCTURING</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Existing Restructuring Techniques</td>
<td>26</td>
</tr>
<tr>
<td>Special Problems</td>
<td>31</td>
</tr>
<tr>
<td>Restructuring Rationale</td>
<td>36</td>
</tr>
<tr>
<td>The Critical Path Method of Restructuring</td>
<td>39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER IV. EXPERIMENTS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theory of Parameters</td>
<td>56</td>
</tr>
<tr>
<td>Circuit times</td>
<td>57</td>
</tr>
<tr>
<td>Page size</td>
<td>61</td>
</tr>
<tr>
<td>Window size</td>
<td>63</td>
</tr>
<tr>
<td>Restructuring Experiments</td>
<td>65</td>
</tr>
<tr>
<td>Performance Analysis</td>
<td>72</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER V. CONCLUSIONS</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BIBLIOGRAPHY</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>81</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>APPENDIX A. THE VMM SIMULATOR</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Units</td>
<td>85</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Memories</td>
<td>87</td>
</tr>
<tr>
<td>Networks</td>
<td>92</td>
</tr>
<tr>
<td>Control</td>
<td>92</td>
</tr>
<tr>
<td>Measurements</td>
<td>95</td>
</tr>
<tr>
<td>APPENDIX B. EXPANSION OF FORM-PAGES</td>
<td>96</td>
</tr>
<tr>
<td>APPENDIX C. PROGRAM LISTING FOR EXPERIMENTS</td>
<td>98</td>
</tr>
</tbody>
</table>
CHAPTER I. INTRODUCTION

Motivation

Parallel computers have become increasingly more important in the quest to provide more speed and more computational power. Vector machines (24, 36, 38) array processors (17, 18, 29), and multiple-processor systems (39, 40, 46) are parallel architectural designs which have been proposed or constructed. Although each design is particularly well-suited for a special class of applications, the general parallel capabilities of each design appears limited due to a reliance on the von Neumann principles of architectural design (20). Each design still relies on a sequential scan of instructions to determine the scheduling of instructions.

An alternate design principle which may allow the architecture to more fully exploit the inherent parallelism of a computation is the principle of data flow. Its main premise is that the sequential flow of control imposed on programs by the conventional von Neumann machine impedes the progress of the computation. Rather than executing sequentially, instructions in a data flow machine execute as soon as their operands are available, provided there are sufficient resources.

A potentially costly factor of a data flow compiler is the need for a mechanism in memory to recognize when an instruction is ready to execute. Whether implemented in hardware or software, this mechanism will have a better cost/performance ratio if only a subset of the instructions need to be checked at any one time. This suggests the use of a cache memory or a virtual memory organization.
A cache or virtual memory in any system is practical only if a high hit ratio occurs, that is, a high proportion of references is satisfied locally without accessing the secondary memory. Programs executed in a typical von Neumann architecture often exhibit high degrees of locality, thereby providing the desired hit ratio. Little information is currently available on the run-time behavior of data flow programs. In particular, it is unknown to what extent data flow programs exhibit locality.

Numerous studies (15, 21, 22, 23) have shown that restructuring a program will improve its paging performance in a sequential environment. No study has yet shown whether restructuring a data flow program will improve its paging performance.

It is the purpose of this dissertation to study the behavior of data flow programs, to determine if locality exists in such programs, and if so, to investigate restructuring techniques for these programs. The results of this study will hopefully provide insight into the desirability of a cache memory or a virtual memory in a specific type of data flow environment.

**Fundamentals of Data Flow Computers**

A data flow program can be viewed as a directed graph where the nodes represent instructions and the arcs represent the flow of data. An example of a data flow graph is shown in Figure 1.1. A token on an arc represents an available operand. Figures 1.2 and 1.3 show possible successive configurations starting with the token configuration in Figure 1.1. Each data flow instruction executes as soon as its operands are
Figure 1.1. Data flow graph of \((-b+\sqrt{b^2-4ac})/2a\)

Figure 1.2. Execution of Figure 1.1
available, provided there is no contention for resources. The result of the execution of an instruction is sent to the successors of that instruction. By relying on this flow of data through the program to activate execution, data flow computers avoid the conventional sequential program scan and thus exploit the parallelism inherent in the program graph.

Exploiting this parallelism results in two-dimensional instruction traces. One dimension represents time; the other dimension represents the degree of parallelism. Figure 1.4 shows a two-dimensional instruction trace, hereafter called an execution fringe, for the complete execution of the program graph in Figure 1.1. The execution fringe records when an instruction begins execution. Thus Figure 1.4 tells us that instructions 1, 2, 3, and 4 began execution at time 1, instruction 5 began
at time 2, etc.

<table>
<thead>
<tr>
<th>t</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>9</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.4. Execution fringe for Figure 1.1

Another type of instruction trace is possible in a data flow environment. A reference fringe records references to instruction memory. Figure 1.5 shows the reference fringe for the program graph in Figure 1.1. Here we see that an instruction may be referenced but not yet be ready to execute. For example, instruction 6 is referenced by the arrival of an operand at time 2 but does not begin execution until time 3. This delay is caused by the necessity for instruction 6 to wait for the result from instruction 5. Note that the execution fringe is a subset of the reference fringe. This is true for a system without contention since, with the exception of the initial instructions, the execution of an instruction is triggered by some token or signal being stored into the instruction.

<table>
<thead>
<tr>
<th>t</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.5. Reference fringe for Figure 1.1
The data flow concept was first proposed in 1966 by Karp and Miller (25). Since then several data flow architectures have been proposed (3, 9, 14, 32, 35, 44, 45). These architectures can be divided into two types: single token per arc architectures and multiple token per arc architectures. In addition to the normal requirements, single token per arc architectures require that the output arcs of an instruction be cleared of tokens before the instruction executes. Generally implemented by using feedback signals, this requirement prohibits the overwriting of tokens, thereby aiding in the maintenance of determinacy and the avoidance of improperly terminating programs. Multiple token per arc architectures do not impose this output arc requirement. Instead they provide both the space for multiple tokens on each arc and a naming mechanism for identifying and matching tokens which must be used together.

The Data Flow Model

The specific virtual memory model (VMM) used in this study is outlined in Figure 1.6 and is a variation of the model originated by Dennis and Misunas (14). An overview description of the program used to simulate VMM is given in Appendix A. A more detailed description of the basic elements of the simulation is given elsewhere (30, 42).

In VMM, the instruction memory is a linearly organized collection of instruction cells containing the encoded data flow graph. Each cell may contain one data flow instruction, including the opcode, operands, destinations, and all necessary control information. The instruction memory corresponds to the secondary memory in a two-level paged virtual memory.
The instruction cache is also a linearly organized collection of instruction cells and corresponds to the primary memory. Only the most currently active pages reside in the instruction cache. Note that the instruction cache must contain sophisticated logic capable of recognizing which instructions are enabled for execution.

The networks act as switches. The arbitration network uses an instruction's opcode to find a path from the cache to an appropriate functional unit. Similarly, the distribution network uses an instruction's destinations to find a path from the functional unit to the appropriate instructions in the cache.

The functional units may be special purpose processors or general purpose microprocessors. The functional units provide direct semantic
support for operations appearing in high-level languages. Among the operations supported are arithmetic, relational, and input/output operations.

A procedure call loads the entire procedure into the instruction memory and the first page of the procedure into the instruction cache. Enabled instructions, that is, instructions which are ready to execute, are recognized in the cache and sent through the arbitration network to the functional units. At this time feedback signals indicating that a token has been consumed are sent through the distribution network to the predecessors of the fetched instructions. The functional units execute the fetched instructions and send the results through the distribution network to the instructions specified by the destinations.

If a feedback signal or a result is sent to an instruction which is not in the cache, a page fault occurs. The page containing the requested instruction is then loaded into the cache and the value is stored into the instruction. The interface between the instruction cache and the instruction memory may be designed in a number of different ways to provide the fast, highly parallel transfers which are necessary.

Several differences between VMM and the Dennis-Misunas model require comment. Their model does not require a strict enforcement of feedback but rather requires feedback signals only where necessary (7). Logically the two models execute equivalently; however, VMM will have more feedback signals and thus possibly a higher page fault rate and a larger working set size than a model not requiring a strict enforcement of feedback. The second difference is that the Dennis-Misunas model partitions
the cache and the memory and makes the restriction that instructions from memory partition i must appear only in cache partition i. VMM makes no such restriction and thus may possibly make better use of the cache. Finally, the Dennis-Misunas model loads and replaces only single instructions. Since VMM loads and replaces a page of instructions, VMM can possibly make better use of program locality.

It is important to note that in VMM the code in the instruction memory and the cache is not pure. Operands and control information are stored directly into the instructions. Thus multiple invocations of a procedure require multiple copies of the code in the instruction memory. However, instructions are serially reusable, and procedures are reentrant to the extent that streams of data may be pipelined through a procedure body provided the single token per arc requirement is not violated.

Because elementary data values may be stored directly in the instructions, only references to data structures need to access any external memory. These data structure references will exhibit different behavior than instruction references and, although important, are not considered in this study. The absence of data structure references does not affect the instruction reference fringe or execution fringe. Because it is excluded from this study, the data structure memory is not shown in Figure 1.6.

For the purposes of this study, it is assumed there is no contention for functional units, instruction memory, or data paths. A large amount of contention could seriously impair the paging performance. Attempting to optimize performance when contention exists is a task
scheduling problem and is not addressed in this thesis. In practice, contention might be controlled by the long-term scheduling policy which determines the program load. Such a policy would have to depend on compile time information regarding the expected resource usage requirements of a program.

This study is concerned with program behavior. Therefore the discussions and experiments are limited to a uniprogrammed system, where a single user pages against himself. The performance measures of interest are execution time, maximum cache requirement, and time-space product.
CHAPTER II. LOCALITY

The principle of locality has been defined in a number of equivalent ways in the literature.

Denning (11) defined it in 1970 as follows:

Let the reference density for page i be \( a_i(k) \) = Probability (kth reference is i). Let a ranking of a program's pages be a permutation \( R(k) = (l', 2', \ldots, n') \) such that \( a_l'(k) > \ldots > a_n'(k) \). A ranking change occurs at reference k if \( R(k-1) \neq R(k) \). Let a ranking lifetime be the number of references between ranking changes. Then the principle of locality states that the rankings are strict and the expected ranking lifetimes long.

In 1972, Denning (10) summarized locality as the following properties:

1) A program distributes its references nonuniformly over its pages.

2) The density of references of a given page tends to change slowly in time.

3) Two reference string segments are highly correlated when the interval between them is small, and tend to become uncorrelated as the interval between them becomes large.

In 1975, Denning and Kahn (12) described a program's execution as a sequence of phases, each of which is a locality.

Each of these definitions formalizes the same property of program behavior which is exhibited to varying degrees by all practical programs run sequentially (11). This property is the tendency of a program to favor a subset of instructions during a given interval. These subsets, called localities, correspond to the high level constructs used in the program. Thus the degree of locality exhibited by an unstructured program is determined by the programmer's style of programming and
Locality may be divided into two categories: spatial locality and temporal locality (28). Both types of locality appear in a data flow environment.

Spatial Locality

Spatial locality refers to the case where the next reference comes from a virtual space adjacent to the last reference. That is, if \( m \) were referenced at time \( t \), then the reference at time \( t+1 \) would tend to be from \( (m-k,m+k) \). This type of locality is produced by straight-line code in a sequential environment.

Straight-line code may also produce spatial locality in a data flow environment. In fact, one section of straight-line code may produce several areas of spatial locality. Each area of spatial locality is represented by a path of activity as determined by the program's data dependencies. Figure 2.1 shows an example where several paths of activity are generated from one section of straight-line code in a data flow program. Multiple paths are possible because the references are based on the flow of data rather than any imposed sequential flow of control.

Figure 2.1 gives the high-level code, an abbreviated version of the compiled code, and the data flow graph. An encoded form of the graph, the compiled code is abbreviated in that only the information directly relevant to this discussion is given. The entries in an instruction are as follows: instruction number opcode operand(s); destination instruction(s). This abbreviated syntax will be used in the
remains of this thesis. For a detailed discussion of all the information needed for execution, see Appendix A.

Actually, each path of activity represents a potential area of spatial locality. In order to realize this potential spatial locality, the instructions which comprise a path of activity need to be grouped together in virtual memory.

Temporal Locality

Temporal locality refers to the case where the set of pages which were referenced during the last interval will be referenced again during the next interval. That is, if t is a point in time and the set P was referenced during interval (t-k,t), it is likely that P will be referenced again during interval (t,t+k). Recurrent use of the same instructions, such as one finds in loops, produces temporal locality.
In single token per arc processors, a data dependency between successive iterations of a loop restricts execution to only one iteration at a time. If we consider the execution of the loop's instructions as comprising a locality pattern, then the complete execution of the loop will appear as a number of repetitions of that pattern, none of which overlap. Figure 2.2 shows the locality patterns in an execution fringe for a loop run on a single token per arc processor.

A multiple token per arc processor allows a loop to unwind (3). Unwinding a loop means executing the loop as fast as the data dependencies permit without including synchronizing instructions to enforce single tokens per arc. Thus several iterations may be active concurrently, as shown in Figure 2.3. A loop whose only data dependency is the index generation will unwind quickly and produce locality patterns which overlap. The delay between successive locality patterns is due to the time required to generate an index. Figure 2.4 shows a loop whose data dependencies will not permit any unwinding. In this case a multiple token per arc processor cannot execute the loop any faster than a single token per arc processor.

A data independent loop contains no data dependency between iterations. Ideally, all iterations of such a loop would execute simultaneously. If duplicate references per time step are ignored, simultaneous execution produces the same execution fringe as if only one iteration were executed, as shown in Figure 2.5.

Unfortunately, it is not clear how to implement simultaneous execution of a loop. Approximations include recursing, streaming, and
while i<=n do
  y:=i*i+2+3*i+4;
  output y file=outf format=F(8,2);
  i:=i+1
end

High-level code

10 <= _,_;11,12,13,14,15,18,19
11 Merge _,_;10,14,15,19
12 Merge _,_;10,12
13 Merge _,_;18
14 ** _,;2;16
15 *3,_,;16
16 +=,_,;17
17 +=,4;18
18 write _,_,';F(8,2)';13
19 + _,1;11

Compiled code

<table>
<thead>
<tr>
<th>t</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>17</td>
<td>18</td>
<td>13</td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>17</td>
<td>18</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution fringe

Figure 2.2. Loop executed on a single token per arc processor. An explanation of individual data flow instructions is given in Appendix A
while i≤n do
  y:=i^2+3*i+4;
  output y file=outf format=F(8,2);
i :=i+1
end

High-level code

10  ≤_;11,12,16
11  **_;2;13
12  *3,_;13
13  +_,_;14
14  +_,4;15
15  write_,'outf','F(8,2)';_
16  +_,1;10

Compiled code

<table>
<thead>
<tr>
<th>t</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution fringe

Figure 2.3. Unwinding loop executed on a multiple token per arc processor
while i < n do
    sum := sum + i
    i := i + 1
end

High-level code

24 < , ; 25, 26, 27, 28
25 Id ; 29
26 Id ; ?
27 Id ; 29, 30
28 Id ; ?
29 + , ; 25, 26
30 + , 1 ; 24, 27, 28

Compiled code

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>24</td>
<td>25</td>
<td>29</td>
<td>24</td>
<td>25</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>30</td>
<td></td>
<td>27</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution fringe

Figure 2.4. Loop that won't unwind executed on a multiple token per arc processor
for all $i \in \{0, n\}$ do
    $y := i^2 + 3i + 4$;
    output $y$ file= outf format=F(8,2)
end

Figure 2.5. Simultaneous execution
physically replicating code. Each approximation produces a specific type of memory reference pattern and thereby affects the locality.

Rewriting a loop as a recursive procedure will allow a data independent loop to unwind in a single token per arc processor. Figure 2.6 shows how the locality patterns will overlap if the code is reentrant. The delay between successive locality patterns is due to the overhead of a procedure call. If the code is not reentrant, each procedure call will load a new copy of the procedure. Since successive iterations are then handled by separate procedure bodies, the locality patterns no longer exist and all temporal locality is lost, as shown in Figure 2.7.

Rewriting the loop as a stream computation (45) will allow stream-oriented execution. A stream computation pipelines values through the code in a first-in, first-out order. Figure 2.8 shows that in this case the locality patterns will again overlap. The delay between successive locality patterns is equal to the production rate of the computation once the pipe is filled. Note that a locality pattern includes all of the instructions in the stream computation. Thus the entire program would constitute the locality if the entire program could be streamed.

Physically replicating the code of a loop provides for more concurrent execution at the expense of increased instruction memory requirements. For loops where the number of iterations is known a priori, the code for the loop’s body may be replicated that many times, thus removing all looping. Figure 2.9 shows that all temporal locality is lost in this extreme case of code replication. Perhaps more familiar is the
proc P(in(i,n), out(y))
begin
  file outf;
  integer i,y,n;
  if i<n then begin
    y:=i**2+3*i+4;
    output y file=outf format=F(8,2);
    call P(in(i+1,n), out(y))
  end
end

High-level code

Compiled code

<table>
<thead>
<tr>
<th>t</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution fringe

Figure 2.6. Recursive procedure with reentrant code
Execution fringe

Figure 2.7. Recursive procedure of Figure 2.6 without reentrant code. Instruction 1 is a new copy of instruction 1.

case where the code is replicated only once, thus providing two copies. In this case, execution would proceed twice as fast as the original version with only a minimal increase in instruction memory requirements. Temporal locality is affected as shown in Figure 2.10. The locality pattern is approximately doubled, and only half as many occurrences of the pattern exist.

Conclusions

Data flow programs do exhibit locality.

Straight-line code produces potential areas of spatial locality. Unlike sequential environments, a data flow environment may require the reordering of a program's instructions in order to exploit these potential areas of spatial locality.

Recurrent use of instructions produces varying degrees of temporal locality. The degree to which a program exhibits temporal locality is directly related to the program's locality patterns. A program with overlapping locality patterns, as produced by streaming, unwinding, or
```
y := 1**2 + 3*144;
output y file=outf format=F(8,2);
```

High-level code

```
15 **_,2;17
16 *3,_,17
17 +_,_;18
18 +_,4;19
19 write _,outf,F(8,2);_
```

Compiled code

<table>
<thead>
<tr>
<th>t</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution fringe

Figure 2.8. Stream computation
y := i**2 + 3*i + 4;
output y file=outf format=F(8,2);

i := i+1;
y := i**2 + 3*i + 4;
output y file=outf format=F(8,2);

i := i+1;
y := i**2 + 3*i + 4;
output y file=outf format=F(8,2);

High-level code

<table>
<thead>
<tr>
<th>t</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>15</td>
<td>16</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>1</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
</tbody>
</table>

Execution fringe

Compiled code

Figure 2.9. Complete replication of code
while i<=n do 
    y:=i**2+3*i+4;
    output y file=outf format=F(8,2);
    i:=i+1;
    y:=i**2+3*i+4;
    output y file=outf format=F(8,2);
    i:=i+1
end

High-level code

10 <= _._;11,12,13,14,15,18,19
11 Merge _._;10,14,15,19
12 Merge _._;10,12
13 Merge _._;18
14 **_._;16
15 *3,_._;16
16 +_._;17
17 +_,4;18
18 write _._,'F(8,2)';24
19 +_,1;20,21,25
20 **_,2;22
21 *3,_,22
22 +_,_;23
23 +_,4;24
24 write _._,'F(8,2)';13
25 +_,1;11

Compiled code

<table>
<thead>
<tr>
<th>t</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>12</td>
<td>16</td>
<td>11</td>
<td>18</td>
<td>24</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>20</td>
<td>17</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>21</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Execution fringe

Figure 2.10. Partial replication of code
recursing with reentrant code, displays tightly-bound temporal locality. A program with nonoverlapping locality patterns displays loosely-bound temporal locality. Because a tightly-bound locality will be more active than a loosely-bound locality, it is more important to keep a tightly-bound locality resident in the cache during its execution. Interestingly, the higher activity rate of a tightly-bound locality will tend to keep the locality resident.
CHAPTER III. RESTRUCTURING

Program restructuring is the reordering of code and/or data segments within a virtual memory space. Restructuring is intended to improve a program's performance in a virtual memory system by making the program's reference patterns more local. In other words, restructuring attempts to increase the degree of locality exhibited by a program.

In particular, restructuring techniques attempt to increase spatial locality without decreasing temporal locality. They do this by grouping instructions or blocks of instructions together according to some measure of spatial locality while keeping areas of temporal locality intact.

Existing Restructuring Techniques

Most of the existing restructuring methods consist of four phases. First, a program is partitioned into blocks. Considered an atomic entity, each block consists of a relocatable set of contiguous instructions or data. To be beneficial a block should be smaller than a page. Second, a restructuring graph is constructed using the blocks as nodes and some form of interblock connection as edges. Each edge is given a weight. Third, blocks are clustered together in such a way as to minimize the sum of remaining interblock weights. Fourth, the resulting clusters are relocated in contiguous space in virtual memory.

The major differences in these techniques are in the choice of using static or dynamic information to determine the interblock connections and in the choice of the measure to use as the interblock weight.
The techniques which use static information rely on program structure and may be done at compile-time. These techniques include those of Ramamoorthy, Lowe, Ver Hoef, Baer and Caughey, and Jain.

Ramamoorthy (33) assumes the existence of the restructuring graph with edges representing possible branches between blocks. He partitions this graph into maximal strongly connected subgraphs and link subgraphs. A strongly connected subgraph is a subset of a graph where every node is reachable from every other node. Such a subgraph which is not a proper subset of any other strongly connected subgraph is a maximal strongly connected subgraph. These subgraphs correspond to outermost loops. The link subgraphs correspond to straight-line code between loops. Ramamoorthy suggests relocating the program such that each maximal strongly connected subgraph is in one segment and the link subgraphs are distributed among the segments to fill holes. This restructuring requires a minimum primary memory size equal to the largest segment. Realizing that this size requirement could be prohibitive, Ramamoorthy supplements his technique by breaking apart maximal strongly connected subgraphs into fixed size pages. He uses probabilistic information about the frequency of use of branches between strongly connected subgraphs within a maximal strongly connected subgraph in order to break the subgraph along branches which are infrequently used.

Lowe (26) partitions a program into instruction units which are blocks of instructions with only one entry and exit per block. The edges in the restructuring graph are control paths of length one between instruction units. Lowe modifies Ramamoorthy's algorithm by including a
size constraint during the phase where maximal strongly connected subgraphs are formed. Thus the resulting clusters will fit in pages without resorting to an additional step to split them according to frequency of use information. Lowe packs the smallest, fastest loops into clusters first to reduce interpage transfers. Lowe specifies only algorithms to cluster cycles, not a complete restructuring technique.

Ver Hoef (43) uses the same restructuring graph that Lowe uses. Loops are detected, and the instruction units comprising a loop are merged together under the constraint that a merged group fit into a page. After the loops are detected and merged, the resulting structure is traversed and nodes are merged again with the constraint that the merged node fit in a page. A clean-up pass attempts to minimize the total required storage by combining nodes which will fit together in a page.

Designed for Fortran programs, the restructuring technique of Baer and Caughey (5) is similar to those of Lowe and Ver Hoef. The program is partitioned into instruction units, loops are detected, and the level of embeddedness is computed for each loop. Innermost loops are packed into pages first with the constraint that loops are not split across page boundaries unless the loop is larger than a page.

Jain (22) uses a restructuring graph which has strongly connected subgraphs as nodes and control paths of length one as edges. Recognizing that a program's memory allocation is generally more than one page, Jain defines the resident set at time t to be those pages which are in memory at time t. Since a reference between two blocks in the resident set is not a page fault, Jain chooses to use the time-space product as
the criterion for clustering rather than interblock connections repre-
senting page faults. To minimize the time-space product, each strongly
connected subgraph should be contained in the minimum number of pages and
holes should be filled, even if it means splitting nodes.

The techniques which use dynamic information to determine the inter-
block connections rely on a "typical" reference trace from an execution
of the program. These techniques work well only for programs which are
relatively insensitive to input data. Hatfield and Gerald (21) have
shown that many systems programs, such as compilers, assemblers, or edi-
tors, are remarkably insensitive to input data. The techniques of
Hatfield and Gerald, Ferrari, and Johnson are dynamic.

Hatfield and Gerald (21) partition a program into blocks, which they
call sectors. They then obtain a sector reference trace by executing
the program. The interblock weights are computed from the sector refer-
ence trace. For a given edge between block i and block j, the weight is
equal to the number of times sector i was referenced immediately before
or immediately after sector j was referenced. This weight is a measure
of nearness between blocks i and j. Hatfield and Gerald then cluster the
sectors attempting to minimize a function, such as the square, of the
interconnecting weights. They report improvements in paging performance
between two-to-one and ten-to-one using this technique.

After partitioning a program into blocks, Ferrari (15) obtains a
block reference trace using a working set memory management algorithm.
He defines a critical reference to be a reference to a block which is not
in the working set. The working set at the time of a critical reference
is called a critical working set. Ferrari's goal is to minimize the number of critical working sets, thereby minimizing the number of page faults. He attempts to attain this goal by using as the interblock weight from block i to block j the number of critical working sets having i as their critical reference and containing j. The sum of the weights between blocks i and j then represent the number of critical working sets which will not be critical if i and j are packed together in a page. Ferrari improves paging performance by clustering blocks so as to minimize the remaining interblock weights.

Johnson (23) experimented with several models of computing interblock weights and with several clustering algorithms. Among the ways of computing interblock weights are Hatfield and Gerald's nearness method, Ferrari's critical working set method, a method where the weight is the number of times both i and j were in the working set when either i or j was referenced, and a method comparable to the critical working set method but using a sector LRU stack instead of a working set. Each of the clustering algorithms attempts to improve locality by merging together the nodes which maximize some function of the interblock weights. Johnson reports reductions in number of page faults of between twenty-to-one and forty-to-one when comparing a program organized according to one of his restructuring techniques to the program organized in a bad way, such as ordering the blocks by size.

Another existing restructuring technique does not fit into either of the above categories. The technique of Baer and Sager (6) works well only at the nonexecutable levels of a multilevel memory hierarchy. Here
restrictions is truly dynamic in that it occurs as the program executes. Let the memory at level \( i \) have page frames of size \( s \) and the memory at level \( i+1 \), that is, one level slower, have page frames of size \( k \times s \). When a page fault occurs at level \( i \), the entire page at level \( i+1 \) which contains the referenced instruction is loaded into memory \( i \). This is in effect preloading \( k-1 \) small pages. The referenced page of size \( s \) is placed at the top of the LRU stack for memory \( i \). The \( k-1 \) preloaded pages are placed in the bottom \( k-1 \) slots of the LRU stack. If pages need to be removed to make room for incoming pages, the \( k \) pages at the bottom of the LRU stack are clustered together and offloaded to a page in memory \( i+1 \). Thus the clustering of small pages is dynamic and depends on program behavior. This technique performs best for programs which tend to reference only small portions of their pages. For these programs, the technique clusters unused portions together and removes them from the faster memory.

Special Problems

Certain problems make the above techniques unsuitable for use on data flow programs. These problems stem from the basic differences between sequential and data flow environments.

All of the static and dynamic restructuring techniques partition the program into single-entry, single-exit blocks. Consisting of straight-line code, each block is an area of spatial locality in a sequential environment. No restructuring is needed within a block, since execution of the block starts at the first instruction and sequences through the
remaining instructions in the block. Therefore, each block is used as an atomic entity in the rest of the restructuring technique.

Single entry, single exit blocks from the high level program do not represent areas of spatial locality in a data flow environment. Several areas of spatial locality may be found in one block of straight-line code. Restructuring within each block may be necessary to exploit the potential spatial locality. Thus a more microscopic approach than current techniques provide may be needed.

The dynamic techniques assume the existence of a typical reference string. Problems arise if the reference string used is not typical. These same problems occur if the reference fringe is not typical.

Additionally, problems arise regarding the uniqueness of a reference fringe. Page faults are not represented in a reference string, so reference strings are unique irrespective of the memory management algorithm, page size, or page fault wait time. However, it is impossible to disregard page faults in a reference fringe due to the parallel nature of the fringe. Since independent activity may continue during the page fault wait time, a page fault skews the reference fringe and the locality pattern. For example, Figure 3.2 shows a reference fringe for the program in Figure 3.1. This fringe was generated under the assumption that the entire graph was in the instruction cache. On the other hand, the reference fringe in Figure 3.3 was generated for the program in Figure 3.1 assuming a page fault wait time of one, a working set memory management algorithm with a window size of three, a page size of five, and the instruction organization shown in Figure 3.3. This illustrates the
Figure 3.1. Sample program written in a typical high-level language and the corresponding data flow code
Figure 3.2. Execution fringe for Figure 3.1 assuming program resident

dependence of a reference fringe and a locality pattern on paging parameters. An infinite number of reference fringes are possible for one program. Thus not only would a dynamic restructuring technique for data flow programs rely on typical input data, but it would also have to rely on a certain set of paging parameters and a certain instruction organization.
Instruction organization

| $t$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 0   | 1 | 3 | 7 | 2 | 8 | 32| 3 | 12| 13| 15| 16| 20| 5 | 22| 25| 26| 30 |    |    |
| 4   | 5 | 9 | 10| 14| 18| 17| 21| 28| 29|    |    |    |    |    |    |    |    |    |    |
| 6   | 11| 19| 24| 27|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 20  | 21| 22| 23| 24| 25| 26| 27| 28| 29| 30 | 31 | 32 | 33 | 34 |    |    |    |    |    |
| 31  | 7 | 2 | 8 | 32| 3 | 12| 13| 15| 16| 20 | 5  | 22| 25| 26|    |    |    |    |    |    |
| 9   | 10| 14| 18| 17| 21| 28| 29|    |    |    |    |    |    |    |    |    |    |    |    |
| 11  | 19| 24| 27|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 35  | 36| 37| 38| 39| 40| 41| 42| 43| 44| 45 | 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 |    |
| 30  | 31| 7 | 2 | 8 | 32| 3 | 12| 13| 15| 16 | 20 | 5  | 22| 25| 26| 30 |    |    |    |    |
| 9   | 10| 14| 18| 17| 21| 28| 29|    |    |    |    |    |    |    |    |    |    |    |    |
| 11  | 19| 24| 27|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 54  | 55| 56| 57|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 31  | 7 | 2 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

Execution fringe

Figure 3.3. Instruction organization and execution fringe for Figure 3.1 assuming paging
But the purpose of restructuring is to change the instruction organization, which will change the reference fringe, which in turn will affect the expected success of the restructuring. The mutual dependence between a reference fringe and a dynamic restructuring technique could lead to a vicious cycle of finding a reference fringe, restructuring using that fringe, finding another reference fringe, using that fringe to restructure, etc. Therefore, the success of a dynamic restructuring technique for data flow programs is expected to be small.

The overhead involved in maintaining page tables prevents the success of Baer and Sager's restructuring technique at the executable level of virtual memory. This holds whether the environment is sequential or data flow. Since this study considers only a two-level virtual memory, Baer and Sager's technique is not applicable.

Restructuring Rationale

Restructuring is intended to increase spatial locality while maintaining the existing temporal locality. Maintaining temporal locality provides the benefits of lowering the expected execution time and time-space product by packing a section of code which is expected to execute many times into the fewest number of pages possible. The static restructuring techniques above maintain temporal locality by detecting loops in the program and by keeping the instructions of each loop together in the restructured organizations. This idea is heuristically sound and is one which is incorporated into the proposed restructuring technique of the next section.
Increasing spatial locality is an attempt to store together those instructions which will be used sequentially. In sequential environments spatial locality is based on flow of control; in data flow environments it is based on flow of data. But is increasing spatial locality important in a data flow environment? Consider the program of Figure 3.1. This program may be rewritten, as shown in Figure 3.4, in a language which allows the assignment of a value to an identifier only once during the course of execution of the program. These so-called single-assignment languages do not require any relative order between data dependent instructions since any use of a value depends on its only production regardless of where the production appears in the program text. Thus the extent to which an unrestructured single-assignment program exhibits spatial locality is largely due to programming style. Figure 3.5 shows the code generated from the program in Figure 3.4. This instruction organization has a lower degree of spatial locality than the organization in Figure 3.3. The execution fringe in Figure 3.5 was generated using the single-assignment organization and the same parameter values that were used to generate the reference fringe in Figure 3.3. Note the difference in the execution time. This small difference could loom large if this code were in the body of a loop which executed many times.

The effect of a single-assignment language on program behavior is important because single-assignment data flow languages (1, 4) are currently being developed. Considered compatible with the concept of data flow computation, single-assignment languages are expected to play an increasingly important role in the future of data flow.
proc main
begin
    file=inf,outf;
    real a,b,c,t1,t2,t3,t4,x,y;
    integer i;
    while i<=3 do;
        output x,y file=outf format=F(5,2),F(5,2);
        t2:=2*a;
        input a,b,c, file=inf format=F(5,2),F(5,2);
        t3:=-b+t1;
        x:=t3/t2;
        t4:=-b-t1;
        i:=i+1;
        t1:=sqrt(in(b**2-4*a*c));
        y:=t4/t2;
    end;
    i:=1
end

High-level code

Compiled code

Figure 3.4. Single-assignment program
The Critical Path Method of Restructuring

In any data flow program there are a finite number of static paths from the initial instruction to the end of execution. These paths are based on the data dependencies in the program. Each path has an execution length equal to the sum of its instructions' execution times. The path with the largest execution length is called the critical path.
The critical path may not be unique in that there may be several paths with the same execution length. In this case each path having the largest execution length is a critical path.

Since activity continues during the processing of a page fault, a page fault in itself is not necessarily undesirable. Instead, only page faults which lengthen the program's execution time are undesirable. Because the critical paths determine the program's execution time, distributing instructions over pages in such a way as to minimize the number of page faults along the critical paths will minimize execution time. Caution must be taken that such a distribution does not cause a noncritical path to become critical due to the insertion of page faults.

A general description of the critical path restructuring technique is as follows. First, the compiled program is partitioned into blocks. Each block contains the instructions in a single loop or the instructions in straight-line code outside of any loop. Second, the critical path of each block is determined. Third, the instructions along the critical path are clustered so as to minimize the expected number of page faults along this path. Instructions off the critical path may be clustered with critical path instructions in order to prepage critical path instructions. The remaining paths are each clustered in descending order of execution length using this same procedure. Fourth, the remaining instructions are placed into pages.

A simple example will clarify the method. There are four blocks in the program in Figure 3.6: one for the code before the implied-do input loop, one for the input loop, one for the code between the input loop
proc ex
begin
  real a,b,c,a1,be,becub,alcub,dif;
  integer nx,j,i;
  file inf;
  real array x(l:10),s(l:10);
  input nx,a1 file=inf format=I(2),F(5,3);
  input (x(j) do j=1 to nx) file=inf format=F(5,3);
  c:=(x(l)+x(3))-(2.0*x(2));
  a:=x(l)-b-0.5*c;
  be:=1.0-a1;
  becub:=be**3;
  alcub:=a1**3;
  i:=1;
  repeat
    s(i):=a+b+0.5*c;
    dif:=s(i)-x(i);
    a:=x(i)+becub+dif;
    b:=((b+c)-(1.5*dif))*((a1*a1)*(2.0-a1));
    c:=c-alcub*dif;
    i:=i+1
  until i>nx
end

Figure 3.6. High-level program

and the repeat loop, and one for the repeat loop.

Figure 3.7 shows the compiled code which would be contained in the first block. Notice that the code for the calculation of be, becub, and alcub is included in the first block because this code does not depend on the input loop. Instruction 0 is designated as a root instruction because no other instruction in the block has 0 as a destination. Instructions 4, 6, 8, 13, 40, 41, 42, and 53 are designated as leaf instructions because their destinations are all outside this block.

The critical paths of a block are determined by traversing the encoded data flow graph from the root of a block to its leaves. The completion time of each instruction is computed from the completion times of
its immediate predecessors and the expected time for it to execute. In particular, an instruction's completion time is equal to the sum of its execution time and the maximum completion time of its predecessors. During the calculation of the completion times, backward pointers are formed from an instruction to the predecessor(s) which had the maximum completion time. These predecessors are called critical predecessors. Figure 3.8 gives the final completion times and the backward pointers for the instructions in the first block assuming unit execution time for all instructions. These pointers specify reversed critical paths from the root. The critical path(s) of the block is the critical path(s) of the leaf with the largest completion time. In the first block, the reversed critical path is 40, 39, 7, 5, 3, 2, 1, 0.

Next the instructions are clustered so as to minimize page faults along the critical path. To do this, the entire critical path from the leaf to the root is pushed onto a stack. Using a stack allows us to
access the path in the correct order, from the root to the leaf. As the instructions are pushed onto the stack, each is marked as "placed" to avoid putting an instruction in two places in the virtual address space. Using a predetermined page size, instructions are popped off the stack and logically placed in pages.

When a page is full, a test is made to determine if a page fault can be expected when this page is referenced by its critical predecessor. This test attempts to determine if a noncritical reference will be made to this page early enough to effect prepaging and thereby avoid a page fault along the critical path. The process of testing will also find any "unplaced" instruction which could cause this prepaging. An instruction will cause the desired prepaging if the instruction is referenced early enough that the page has entered the cache before any critical

Figure 3.8. Completion times and back links for the first block
reference has been made to it but late enough that the page has not fallen out of the window. If the test determines that a page fault will occur along the critical path and if the test finds an appropriate instruction to cause prepaging, that instruction will replace the last instruction on the page and the replaced instruction will become the first instruction on the next page. This forces the desired prepaging, and the critical path is continued on the next page.

When the stack is empty, the next most critical path is chosen and the process of clustering begins again. After all of the paths from the leaves have been traversed and their instructions clustered, there may yet be some "unplaced" instructions. These are placed on a new page or in holes on other pages for this block.

A clean-up phase to coalesce pages might be helpful at this point. One has not yet been incorporated into the Critical Path Method. Holes in pages are filled with no-op instructions to maintain the page placements. Although no-op instructions will never be referenced, they do increase the time-space product.

A mapping of old instruction numbers to new instruction numbers is computed by linearizing the pages. This mapping is used to renumber the instructions and the destinations. The final page contents and mapping for the first block assuming a page size of 5 is shown in Figure 3.9.

Consider the block for the input loop. The code generated for this block is given in Figure 3.10 and uses the same code generation templates as a while loop. The locality pattern for a while loop is totally synchronized by the conditional, because the body of the loop cannot begin
Figure 3.9. Page contents and mapping for the first block

```
9 <= _,10,11,12,14,15,16,17,18,23
10 Merge _,9,16
11 Merge _,9,17
12 Merge _,18
14 Merge _,15,23
15 Id _,21
16 Id _,21,22
17 Id _,11
18 Read _,'F(5,3)' ;19,20
19 Select _,1;12
20 Select _,2;21
21 Append _,_,14
22 + _,1;10
23 Id _,24,25,27,30,31,35,48
```

Figure 3.10. Compiled code for the input loop
execution until the conditional executes. Thus the locality pattern for a while loop is similar to the locality pattern of any while loop in that the pattern begins with the conditional and ends with the instructions which feed data to the next iteration. In the case of a while loop run on VMM, the instructions which feed data to the next iteration are the merge instructions.

Since the desired critical paths are those of the locality pattern, the destinations of the merge instructions are broken and the conditional is designated as a root instruction. The merge instructions and any instruction whose destinations are all outside this block are designated as leaf instructions. The restructuring method outlined above may now be applied to this block. Figure 3.11 shows the completion times, back links, page contents, and mapping for the input loop.

Finding the critical paths in a repeat loop requires more work because the locality pattern is not as easily determined as in a while loop. The body of a repeat loop begins execution before the conditional executes. There is no knowledge of which instructions will execute together during the first iteration, because this activity was triggered from outside the block. Additionally, there is no one instruction which is guaranteed to execute alone. Therefore, three or four iterations of the phase computing completion times is needed to establish the critical paths of the locality pattern. The iterations allow the references within the body of the repeat loop to overwhelm the influence of the references from outside the loop which initially triggered execution of the loop. The references within the loop should be used to determine the critical
path, because the loop behavior which is desired is that of the established locality pattern. The need to iterate to establish the correct critical path is completely analogous to the behavior of the locality pattern for a repeat loop. The pattern does not establish itself on the first iteration due to the way a repeat loop executes (34).
The merge instructions and any instruction whose destinations are outside the block are designated as leaves. The conditional and any instruction which is a destination of a merge instruction are designated as roots. The algorithm which computes completion times is applied to the instructions with the merge destinations intact. The merge destinations will cause an infinite loop in the completion times computation. Therefore a counter is attached to each leaf. After enough iterations to establish a pattern, the destinations of leaves are ignored, thereby terminating the computation. Figure 3.12 shows the code generated for the repeat loop of the program in Figure 3.6. Also shown are the final completion times and backward pointers after three iterations of the algorithm. Note that the destinations of the leaves were used only twice.

Details of the critical path restructuring technique are given by the following algorithms.

CRITICAL-PATH:

For each procedure do;
   Input the procedure.
   Call PARTITION.
   For each block do;
      Call ORDER.
      Call LINK.
      Call FORM-PAGES.
   End.
   Call RENUMBER.
End.

End CRITICAL-PATH.

PARTITION:

In this study, programs were partitioned into blocks by hand to allow more flexibility in the experiments. This process may be
Figure 3.12. Compiled code, completion times, and back links for repeat loop

automated in a number of ways. The high-level program could be partitioned into strongly connected subgraphs and link subgraphs just as Ramamoorthy does, or the high-level program could be partitioned into loops and straight-line code by recognizing looping constructs and using them as block boundaries. A similar method could be used
on the compiled code if there are fixed, recognizable templates for
the code generation of looping constructs.

End PARTITION.

ORDER:

In this step instructions are placed in data dependent order.
That is, no instruction X will follow an instruction which depends
on X. Care must be taken when ordering the instructions in a loop
since loops have cyclic dependencies. These cyclic dependencies must
be broken before the instructions are ordered.

This step was unnecessary in this study because the compiler
used produces code in nearly complete data dependent order (2). The
cases where the order is not completely data dependent order in the
code produced by this compiler are predictable and are handled as
special cases. However, not every data flow compiler may generate
code in data dependent order. In particular, it is expected that a
compiler for a single-assignment language may not order the code
according to data dependencies.

It is interesting to note that this step may not be necessary
at all for correct execution. However, having the code ordered by
data dependencies substantially reduces the complexity of LINK.

End ORDER.

LINK:

Set each instruction's completion time equal to the execution
time for that instruction type.

Determine the root instructions and the leaf instructions by
using information provided by the compiler.
Set the iteration counter of each instruction to the desired number of iterations; one for while loops and blocks of straight-line code and at least three for repeat loops.

Decrement the iteration counter of each leaf instruction to set up the mechanism to break the loop on the last iteration.

Mark each root instruction "visit".

For each instruction X with iteration counter > 0 and marked "visit" do:
  For each destination instruction of X do;
    Calculate the completion time of the destination instruction assuming its execution was triggered by X.
    If the calculated completion time is equal to the stored completion time then add X to the list of back links.
    If the calculated completion time is greater than the stored completion time then do;
      Set the stored completion time equal to the calculated completion time.
      Empty the list of back links.
      Put X on the list of back links.
    End.
    Mark the destination instruction "visit".
  End.
Remove the "visit" mark from X.
Decrement X's iteration counter.

End LINK.

FORM-PAGES:

Determine the leaf instructions.

While there is an untraversed leaf do;
  Find the leaf with the largest completion time.
  Using back links, push the "unplaced" instructions on the path from this leaf to a root onto a stack, marking each instruction "placed".
  If the current page is not empty and the entire path will not fit in the remaining space then update current page.
  For each instruction on the stack do;
    Pop the instruction off the stack.
    Call PLACE (Instruction).
  End.
End.

For each instruction not yet "placed" do;
Mark the instruction "placed".
Call PLACE-LAST (Instruction).
End.
Update current page.
End FORM-PAGES.

Several stacks may actually be used in FORM-PAGES. When an instruction has two or more critical predecessors, one of them is chosen to be pushed onto the stack containing the path. The other critical predecessors are pushed onto a holding stack. After the path has been entirely traversed, an instruction is removed from the holding stack. The path specified by this instruction is then traversed, pushing the instructions onto another path stack. After all of the instructions have been removed from the holding stack and these paths traversed, several path stacks may exist. The original path stack contains a critical path; the other path stacks contain branches of critical paths. Stacking the branches at this time prevents the instructions on these branches from being erroneously chosen to prepage a portion of a critical path.

The stacks are unstacked in the order that they were formed. That is, the instructions in the original stack are clustered first, then the instructions in the first branch path stack, then the instructions in the second branch path stack, etc. A branch will be clustered with previously placed portions of a critical path if the entire branch will fit on a page with the previously placed portions. The highest priority when clustering is to keep the instructions of a critical path in order and together.

The algorithmic description of FORM-PAGES given above did not include the handling of multiple stacks. The inclusion of multiple stacks
in the algorithmic description leads to a level of detail which tends to obscure rather than elucidate. However, for the sake of completeness, an algorithmic description of FORM-PAGES which includes the multiple stacks is given in Appendix B.

PLACE (Instruction):

Place the instruction in the current page.
If the page is full then Call TEST (Page).
End PLACE.

In TEST, no-ref is a boolean which is set to false when a prepaging instruction is found.

TEST (Page):

Attempt to find a page P which contains a critical predecessor of the first instruction in Page by using the back links of the first instruction.

If P exists then do;
Set no-ref to true.
For each instruction in P while no-ref do;
If this instruction's completion time is early enough to prepage Page then do;
If any destinations of this instruction reside in Page then set no-ref to false.
Else do;
Attempt to find an "unplaced" destination instruction of this instruction.
If such a destination instruction is found then do;
Mark the destination instruction "placed".
Update the current page.
Remove the last instruction in Page and place it in the current page.
Place the destination instruction into the vacated position in Page.
Set no-ref to false.
End.
End.
End.
End.
End TEST.
In PLACE-LAST, flag is a boolean which is set to false when the
instruction is placed into a page.

PLACE-LAST (Instruction):

Set flag to true.

For each back link of the Instruction while flag do;
   If there is a hole on the page containing the instruction
      pointed to by the back link then do;
      Place Instruction in that page.
      Set flag to false.
   End.
End.

For each destination of Instruction while flag do;
   If there is a hole on the page containing the destination
      instruction then do;
      Place Instruction in that page.
      Set flag to false.
   End.
End.

If flag then do;
   Place Instruction in the current page.
   If the page is full then update current page.
End.

End PLACE-LAST.

RENUMBER:

Form a mapping of old instruction numbers to new instruction
numbers by linearizing the page positions.

Use this mapping to renumber the instructions and destinations.

End RENUMBER.

Figure 3.13 shows the page contents for the entire program of
Figure 3.6.
Figure 3.13. Page contents for entire program of Figure 3.6
CHAPTER IV. EXPERIMENTS

This chapter describes the various experiments that were run in order to determine the advantages and disadvantages of program restructuring in a data flow environment. The experiments were run on the VMM simulator. The experiments did not involve streams or other constructs which produce overlapping locality patterns due to the limitations imposed by the existing compiler and simulator.

The decision was made to choose only one algorithm for memory management and to not experiment with other algorithms in this study. A comparative study of memory management algorithms in data flow environments and their effect on restructuring is left for future work. A working set algorithm was chosen for this study, because it allows for a study of program behavior without introducing contention for the instruction cache. The window size \( w \) is defined to be in time units rather than reference units. Thus the working set at time \( t \) is the set of pages referenced during \( (t-w, t) \) rather than being the set of pages referenced in the last \( w \) references. This modification simplifies the complications raised by the parallel nature of the reference fringes. Without this modification, a tie-breaker would be needed to resolve conflicts since more than one page may be referenced concurrently. A poor choice of a tie-breaker could easily result in thrashing.

Since this study is primarily concerned with program behavior, the performance measures used are execution time, maximum working set size, and time-space product. No attempt is made to capture measurements
concerning system performance, such as system throughput or utilization.

The complexity of the experiments lay in the choice of values for the parameters: primary circuit time, secondary circuit time, page size, and window size. Because a reference fringe is parallel and a page fault does not prohibit independent activities, each parameter value affects a reference fringe much more than each parameter value affects a sequential reference string. Thus the choice of parameter values is quite important.

Preliminary studies on parameter value choices showed certain trends as one parameter value varied while all others remained fixed. These trends are discussed in the following section.

Theory of Parameters

The program, PECR, used in the experiments in this section is listed in Appendix C. It has a complicated structure which includes a number of nested conditionals and repeat loops. The longest executed loop without any embedded loops has a critical path length of 7. There are 332 data flow instructions in the compiled code. A total of 444 instructions execute and 2289 references are made to the instruction cache.

Circuit times

The primary circuit time is the average time required to move one instruction around the circuit from the cache, through a functional unit, and back to the cache. The primary circuit time includes the time to recognize that the instruction is enabled, the time to fetch the instruction, the time to move it through the arbitration network, the time to
execute it, the time to move the result packets through the distribution network, and the time to store the result packets in the cache. Table 5.1 shows the effect of changing the primary circuit time on the execution time, the number of page faults, the maximum working set size, and the time-space product.

Table 5.1. Effect of primary circuit time

<table>
<thead>
<tr>
<th>Page size</th>
<th>Circuit ratio</th>
<th>Window size</th>
<th>Execution time</th>
<th>Page faults</th>
<th>Working set size</th>
<th>Time-space product</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1:1</td>
<td>3</td>
<td>94</td>
<td>168</td>
<td>185</td>
<td>7010</td>
</tr>
<tr>
<td>5</td>
<td>1:2</td>
<td>3</td>
<td>170</td>
<td>218</td>
<td>155</td>
<td>10550</td>
</tr>
</tbody>
</table>

Increasing the primary circuit time increases significantly the execution time regardless of the underlying instruction organization, page size, window size, or secondary circuit time, because the primary circuit time is effectively the instruction cycle time and is not affected by page faults. Increasing the primary circuit time also causes a relative decrease in the window size. Since an instruction takes twice as long to execute, the rate of references is halved. Thus there will be half the number of references in the window. This causes a decrease in the maximum working set size and an increase in the number of page faults. The time-space product increases as the primary circuit time increases, but it does not double with the primary circuit time due to the decreased working set sizes.
The secondary circuit time is the average time to fetch a page from the instruction memory. This time includes the time to request a page, the time to transfer a page from the instruction memory, and the time to store the page in the instruction cache. In other words, the secondary circuit time is the page fault wait time. The effect that the secondary circuit time has on execution time depends upon the ratio of secondary circuit time to primary circuit time. If this ratio is small, page faults have very little effect; if the ratio is large, page faults may overwhelm the execution. Table 5.2 shows the effect of changing the secondary circuit time.

Table 5.2. Effect of secondary circuit time

<table>
<thead>
<tr>
<th>Page size</th>
<th>Circuit ratio</th>
<th>Window size</th>
<th>Execution time</th>
<th>Page faults</th>
<th>Working set size</th>
<th>Time-space product</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1:2</td>
<td>3</td>
<td>170</td>
<td>218</td>
<td>155</td>
<td>10550</td>
</tr>
<tr>
<td>5</td>
<td>2:2</td>
<td>3</td>
<td>187</td>
<td>245</td>
<td>145</td>
<td>10255</td>
</tr>
<tr>
<td>5</td>
<td>4:2</td>
<td>3</td>
<td>233</td>
<td>264</td>
<td>130</td>
<td>10440</td>
</tr>
</tbody>
</table>

As the secondary circuit time increases, the delay caused by a page fault increases, thereby increasing the execution time. More pages may fall out of the window during the delay caused by a page fault when the secondary circuit time is large. This decreases the maximum working set size and increases the number of page faults. The time-space product is a function of the execution time and the working set sizes. For these runs, increases in execution time were offset by decreases in working set
sizes resulting in a rather stable time-space product.

Actual values for the circuit times will be fixed with the configuration of the system. These times depend upon the speeds of the networks, memories, and processors. It is believed that values on the order of one are appropriate values for the ratio of secondary circuit time to primary circuit time. This seemed unreasonable at first, because there is a tendency to compare the circuit ratio to the traditional ratio of secondary memory access time to primary memory access time. Although this comparison is intuitively appealing, it is not valid. While the circuit times do include memory access times, the bottleneck in the primary circuit time is not expected to be the time to access memory but rather the time to travel through the networks.

Traditionally, a cache memory consisted of very fast, expensive memory and was used to provide a faster memory access time. The cache memory in VMM aids in providing a faster primary circuit time by being small, not necessarily by being very fast. By being small, the cache allows the arbitration and distribution networks to be smaller, because fewer instruction frames need to be connected to functional units via the networks. This means that the networks may be narrower, that is fewer connections may be needed between the cache and the network. More importantly, the networks may be shorter, because there may be fewer switches needed to form a path between an instruction and a functional unit. A shorter network means a faster primary circuit time. Thus a cache memory affects the primary circuit time in more ways than merely providing a faster memory access time.
In fact, it is completely possible that the secondary memory could be as fast or faster than the cache memory. This would not jeopardize the need for a cache due to the effect of the cache on the networks. The cache is also needed due to the expense of the logic used to recognize enabled instructions.

**Page size**

Typically as page size increases, execution time decreases. This occurs because a larger page size usually means that more instructions are maintained in the cache and thus the probability of a hit is higher. With the resulting higher hit ratio comes a reduction in the number of page faults, which tends to reduce the execution time of the program.

Unfortunately as page size increases, the working set size when measured in number of instructions tends to increase due to the presence of additional inactive instructions. This results in the undesirable need for a larger cache memory.

Table 5.3 shows the effects of changing page size. Since the entire program has 332 instructions, it is resident in the cache when the page size is 332. Thus 78 is the theoretical minimum execution time for this program with this data. Notice that a page size of 10 provides a degradation in execution time of only 12% while improving the time-space product by almost a factor of 3.

As the page size decreases, fewer instructions reside in the cache, because fewer inactive instructions are paged in with active instructions. This reduces the maximum working set size and the time-space product. It also reduces the probability of a reference being a hit, thereby
Table 5.3. Effect of page size

<table>
<thead>
<tr>
<th>Page size</th>
<th>Circuit ratio</th>
<th>Window size</th>
<th>Execution time</th>
<th>Page faults</th>
<th>Working set size</th>
<th>Time-space product</th>
</tr>
</thead>
<tbody>
<tr>
<td>332</td>
<td>1:1</td>
<td>3</td>
<td>78</td>
<td>1</td>
<td>332</td>
<td>25564</td>
</tr>
<tr>
<td>10</td>
<td>1:1</td>
<td>3</td>
<td>87</td>
<td>97</td>
<td>230</td>
<td>8950</td>
</tr>
<tr>
<td>5</td>
<td>1:1</td>
<td>3</td>
<td>94</td>
<td>168</td>
<td>185</td>
<td>7010</td>
</tr>
<tr>
<td>1</td>
<td>1:1</td>
<td>3</td>
<td>111</td>
<td>693</td>
<td>114</td>
<td>4367</td>
</tr>
</tbody>
</table>

increasing the number of page faults and the execution time.

Possible values of page size for a given program range inclusively between one instruction and the total number of instructions in the program. Under the assumption that processing a page fault takes a fixed amount of time, regardless of the page size, the working set size is minimized and the execution time is maximized when the page size is one instruction. On the other hand, a page size equal to the total number of instructions maximizes the working set size and minimizes execution time.

A page size of one instruction sounds absurd compared to the typical page sizes found in systems today. However, a data flow instruction is considerably larger than a traditional machine instruction. Calculations based on the information needed in a data flow instruction indicate that one instruction will require between ten and fifteen words of memory. The actual instruction size depends on the decisions made when the system is configured. Such decisions would determine the size of the address space, whether instructions are fixed or variable size, how many destinations are allowed per instruction, etc. For example, suppose the information in the opcode segment requires a half word of storage, each
operand requires two and a half words of storage, and each destination requires a word of storage. If the system has fixed size instructions which allow three operands and five destinations, each instruction will require thirteen words of storage. It is interesting to note that a page size of five instructions where each instruction is thirteen words is comparable to a traditional page size of sixty-four words.

One way to decrease the size of instructions is to decrease the number of destinations allowed per instruction. This method will necessitate the inclusion of more instructions in the program, because instructions may logically have more destinations than the number allowed. Instructions must be added to fan out the result to all of the logical destinations. The memory requirements of these added instructions may quickly offset the memory gains caused by the decreased instruction size, if the number of destinations allowed is too small. The optimal number of destinations per instruction is an open question.

An instruction size of ten or fifteen words is not excessive when compared to the sequence of traditional machine instructions which are required to do the same operation as the data flow instruction. For example, an add operation in a sequential machine may require six words: a word for a load instruction, a word for an add instruction, a word for a store instruction, and three words for the operands and result.

**Window size**

Since the working set algorithm was chosen, the window size is a parameter which must be considered. Table 5.4 shows the effects of changing window size. Increasing the window size will tend to cause more
Table 5.4. Effect of window size

<table>
<thead>
<tr>
<th>Page size</th>
<th>Circuit ratio</th>
<th>Window size</th>
<th>Execution time</th>
<th>Page faults</th>
<th>Working set size</th>
<th>Time-space product</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1:1</td>
<td>24</td>
<td>79</td>
<td>81</td>
<td>295</td>
<td>15180</td>
</tr>
<tr>
<td>5</td>
<td>1:1</td>
<td>19</td>
<td>83</td>
<td>91</td>
<td>270</td>
<td>14195</td>
</tr>
<tr>
<td>5</td>
<td>1:1</td>
<td>15</td>
<td>83</td>
<td>93</td>
<td>255</td>
<td>12980</td>
</tr>
<tr>
<td>5</td>
<td>1:1</td>
<td>14</td>
<td>85</td>
<td>99</td>
<td>255</td>
<td>12735</td>
</tr>
<tr>
<td>5</td>
<td>1:1</td>
<td>12</td>
<td>85</td>
<td>100</td>
<td>255</td>
<td>11965</td>
</tr>
<tr>
<td>5</td>
<td>1:1</td>
<td>9</td>
<td>91</td>
<td>127</td>
<td>245</td>
<td>11010</td>
</tr>
<tr>
<td>5</td>
<td>1:1</td>
<td>7</td>
<td>91</td>
<td>135</td>
<td>220</td>
<td>9835</td>
</tr>
<tr>
<td>5</td>
<td>1:1</td>
<td>3</td>
<td>94</td>
<td>168</td>
<td>185</td>
<td>7010</td>
</tr>
</tbody>
</table>

instructions to be maintained in the cache, thereby tending to decrease the execution time and increase the working set size.

Possible values for window size range inclusively between zero and the total execution time. However, a tighter practical bound may be recognized by considering the role which the window plays in the working set algorithm. The purpose of the window is to maintain the entire locality in the cache during the consecutive time units in which the locality is active. In other words, the window should be large enough to keep the pages of a loop in the cache while the loop is executing. Thus the maximum window size needed for a particular program is a function of the longest executing loop. An upper bound for window size is the time to execute once the longest executing loop.

Because the execution time of a loop with an embedded loop is dependent on the number of iterations of the inner loop, it is often impossible to determine a window size large enough to maintain the outer loop for all valid data sets. Therefore, a more reliable value for window
size may be the execution time for the longest executing loop with no
embedded loops. This execution time is exactly the critical path length
of the loop. In the case of PECR, this length is 7. It is important to
note that an upper bound on window size based on critical path length is
loose, because it does not consider program interaction. A much smaller
window size could maintain the locality, because parallel activity may
reference instructions on the locality's critical path and thus maintain
the locality. Of course, it is possible to use a window size larger than
this bound, but the working set size would be increased without the bene-
fit of a decreased execution time unless the window size were enough
larger to begin to maintain a larger locality. For example, a window
size of 15 maintains a larger locality than a window size of 14.

Care must be taken that the window is not too small. If the window
size is less than the primary circuit time, the window will empty while
an instruction is executing. This would lead to thrashing, because the
pages which are paged out while an instruction executes are likely to be
the pages referenced by that instruction.

Restructuring Experiments

In order to examine the effect of instruction organization on per-
formance, four programs were run using three different organizations.

The first organization, COMP, is that resulting from the ISU data
flow compiler. This compiler is for a typical block-structured language
which has approximately the power of Fortran. The language is not a
single-assignment language and therefore requires that the high-level
statement which assigns a value to a variable precede any uses of the variable with that value. The compiler was not written to design a high-level data flow language but was written as a tool for data flow studies and as a study itself into the complexities of automatic translation of high-level programs into data flow graphs. Grouped by constructs, expressions, and data dependencies, the generated code appears in the same order as the high-level code, in much the same way as the code generated for a sequential machine would appear.

The second organization, RAND, is a random reorganization of the compiled, assembled code. A random number generator was used to assign a random number to each instruction. Then the instructions were sorted in ascending order using the random numbers as sort keys. The resulting order was used as a mapping to renumber instructions and destinations. This organization is in some sense a worst case in that the instructions are not organized at all.

The third organization, CP, is that produced by applying the critical path method discussed in Chapter III.

The programs and their input data are listed in Appendix C. A brief description appears below with the results of the restructuring experiments.

The first program, RUNG, basically consists of one large while loop. Of the 88 data flow instructions in the program, 75 instructions are in the loop. The length of the critical path is 23. The data which was used caused four iterations of the loop. During its execution, 321 instructions execute and 1042 references are made to the instruction
cache. The results of the various experiments on RUNG are given in Table 5.5. Given a page size of 88, the program is resident in the cache and provides the theoretical minimum execution time of 103. This minimum is attained by five other experiments and is approached by the other experiments. The maximum working set size is approximately equal in all cases.

Table 5.5. Restructuring experiments on RUNG

<table>
<thead>
<tr>
<th>Method</th>
<th>Page size</th>
<th>Circuit ratio</th>
<th>Window size</th>
<th>Execution time</th>
<th>Page faults</th>
<th>Working set size</th>
<th>Time-space product</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP</td>
<td>5</td>
<td>1:1</td>
<td>--</td>
<td>103</td>
<td>1</td>
<td>88</td>
<td>8976</td>
</tr>
<tr>
<td>RAND</td>
<td>5</td>
<td>1:1</td>
<td>23</td>
<td>103</td>
<td>18</td>
<td>90</td>
<td>8190</td>
</tr>
<tr>
<td>CP</td>
<td>5</td>
<td>1:1</td>
<td>23</td>
<td>103</td>
<td>18</td>
<td>90</td>
<td>8985</td>
</tr>
<tr>
<td>COMP</td>
<td>5</td>
<td>1:1</td>
<td>18</td>
<td>103</td>
<td>18</td>
<td>90</td>
<td>8165</td>
</tr>
<tr>
<td>RAND</td>
<td>5</td>
<td>1:1</td>
<td>18</td>
<td>106</td>
<td>18</td>
<td>90</td>
<td>8985</td>
</tr>
<tr>
<td>CP</td>
<td>5</td>
<td>1:1</td>
<td>18</td>
<td>103</td>
<td>20</td>
<td>95</td>
<td>7915</td>
</tr>
<tr>
<td>COMP</td>
<td>5</td>
<td>1:1</td>
<td>13</td>
<td>107</td>
<td>46</td>
<td>90</td>
<td>7920</td>
</tr>
<tr>
<td>RAND</td>
<td>5</td>
<td>1:1</td>
<td>13</td>
<td>106</td>
<td>18</td>
<td>90</td>
<td>8985</td>
</tr>
<tr>
<td>CP</td>
<td>5</td>
<td>1:1</td>
<td>13</td>
<td>103</td>
<td>34</td>
<td>95</td>
<td>7600</td>
</tr>
<tr>
<td>COMP</td>
<td>5</td>
<td>1:1</td>
<td>8</td>
<td>107</td>
<td>52</td>
<td>90</td>
<td>7000</td>
</tr>
<tr>
<td>RAND</td>
<td>5</td>
<td>1:1</td>
<td>8</td>
<td>114</td>
<td>51</td>
<td>90</td>
<td>9335</td>
</tr>
<tr>
<td>CP</td>
<td>5</td>
<td>1:1</td>
<td>8</td>
<td>107</td>
<td>54</td>
<td>90</td>
<td>7030</td>
</tr>
</tbody>
</table>

The larger working set for the critical path organization is due to internal fragmentation. In all cases except the critical path organization with window size of 8, the maximum working set size is equal to the size of the entire program. Both the compiler and critical path organizations provide smaller time-space products than the random organization, because the random organization is less local than the compiler or
critical path organization.

Note that although the upper bound for window size is 23, a window size of 18 provides the theoretical minimum execution time with the compiler and critical path organizations. In fact, a window size of 13 provides the minimum execution time with the critical path organization. A window size smaller than the computed window size works well in part because the upper bound calculation did not include the page size. To maintain the locality, the window need only maintain the pages of the locality, not the instructions. A window size equal to the critical path length does not assume that any of the locality's instructions are together in pages. A window size equal to the critical path length minus the time to execute the instructions on one page will generally be large enough to maintain the locality for organizations which provide some degree of spatial locality.

The second program, INTE, is again basically one large loop, but the loop is a repeat loop and has an embedded conditional. The entire program has 85 instructions; the loop has 57 instructions of which 16 are in the conditional. The data used in the experiments causes the loop to iterate 5 times and the "then" and "else" bodies to alternate. The program executes 342 instructions and makes 1562 references to the cache. The length of the longest critical path is 21. Table 5.6 gives the results of the experiments run on INTE. The theoretical minimum execution time is 133. Note that the critical path organization with a page size of 5 and window size of 10 attains this minimum. Again the maximum working set is approximately equal to the size of the program. However, the compiler and
critical path organizations attain their maximum working sets only briefly; the random organization maintains the maximum working set throughout the execution and thus inflates the time-space product.

The third program, MATR, has a doubly nested while loop. Of the 82 instructions in this program, 29 instructions are in the inner loop and 34 additional instructions are in the outer loop. The data used caused 6 iterations of the inner loop and 4 iterations of the outer loop. The program executes 358 instructions and makes 1508 references to the cache. The length of the critical path of the inner loop is 11. The results of the restructuring experiments on MATR are given in Table 5.7.

The minimum execution time is not attained by any of the organizations with a page size of 5. This is due to the fact that there are
more locality changes in this program. Transitions between the inner loop and the outer loop tend to result in page faults, because the outer locality is not maintained.

The fourth program, PECR, is the program described in the previous section for use in the experiments on parameter choice. It has several nested conditionals and repeat loops. The longest executed loop without any embedded loops has a critical path length of 7.

The new organization, CP 2, is the result of applying the critical path algorithm to a different partition of the program PECR. The CP organization split the program into blocks only at the boundaries of loops. The CP 2 organization split the program into blocks at the boundaries of loops and at the boundaries of "then" bodies and "else" bodies.

Table 5.7. Restructuring experiments on MATR

<table>
<thead>
<tr>
<th>Method</th>
<th>Page size</th>
<th>Circuit ratio</th>
<th>Window size</th>
<th>Execution time</th>
<th>Page faults</th>
<th>Working set size</th>
<th>Time-space product</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP</td>
<td>5</td>
<td>1:1</td>
<td>16</td>
<td>117</td>
<td>29</td>
<td>70</td>
<td>6050</td>
</tr>
<tr>
<td>RAND</td>
<td>5</td>
<td>1:1</td>
<td>16</td>
<td>118</td>
<td>21</td>
<td>85</td>
<td>9205</td>
</tr>
<tr>
<td>CP</td>
<td>5</td>
<td>1:1</td>
<td>16</td>
<td>113</td>
<td>31</td>
<td>80</td>
<td>6280</td>
</tr>
<tr>
<td>COMP</td>
<td>5</td>
<td>1:1</td>
<td>11</td>
<td>120</td>
<td>39</td>
<td>65</td>
<td>5475</td>
</tr>
<tr>
<td>RAND</td>
<td>5</td>
<td>1:1</td>
<td>11</td>
<td>119</td>
<td>23</td>
<td>85</td>
<td>9150</td>
</tr>
<tr>
<td>CP</td>
<td>5</td>
<td>1:1</td>
<td>11</td>
<td>115</td>
<td>40</td>
<td>80</td>
<td>5775</td>
</tr>
<tr>
<td>COMP</td>
<td>5</td>
<td>1:1</td>
<td>6</td>
<td>121</td>
<td>47</td>
<td>55</td>
<td>4695</td>
</tr>
<tr>
<td>RAND</td>
<td>5</td>
<td>1:1</td>
<td>6</td>
<td>122</td>
<td>42</td>
<td>85</td>
<td>8975</td>
</tr>
<tr>
<td>CP</td>
<td>5</td>
<td>1:1</td>
<td>6</td>
<td>119</td>
<td>53</td>
<td>70</td>
<td>5095</td>
</tr>
<tr>
<td>COMP</td>
<td>5</td>
<td>1:1</td>
<td>3</td>
<td>132</td>
<td>80</td>
<td>50</td>
<td>4120</td>
</tr>
<tr>
<td>RAND</td>
<td>5</td>
<td>1:1</td>
<td>3</td>
<td>143</td>
<td>170</td>
<td>85</td>
<td>8525</td>
</tr>
<tr>
<td>CP</td>
<td>5</td>
<td>1:1</td>
<td>3</td>
<td>121</td>
<td>84</td>
<td>55</td>
<td>4250</td>
</tr>
</tbody>
</table>
This resulted in increased internal fragmentation. The CP 2 organization was an attempt to reduce the number of inactive instructions in the cache due to conditionals. This attempt failed due to the way that conditionals execute in this system. Both the "then" body and the "else" body will be paged into the cache because of the true and false gates. This offsets the attempt to keep inactive instructions out of the cache. Additionally, the CP 2 organization degrades execution time due to the increased number of transitions between smaller localities.

Table 5.8. Restructuring experiments on PECR

<table>
<thead>
<tr>
<th>Method</th>
<th>Page size</th>
<th>Circuit ratio</th>
<th>Window size</th>
<th>Execution time</th>
<th>Page faults</th>
<th>Working set size</th>
<th>Time-space product</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP</td>
<td>5</td>
<td>1:1</td>
<td>19</td>
<td>83</td>
<td>91</td>
<td>270</td>
<td>14195</td>
</tr>
<tr>
<td>RAND</td>
<td>5</td>
<td>1:1</td>
<td>19</td>
<td>85</td>
<td>91</td>
<td>335</td>
<td>23575</td>
</tr>
<tr>
<td>CP</td>
<td>5</td>
<td>1:1</td>
<td>19</td>
<td>82</td>
<td>90</td>
<td>285</td>
<td>14490</td>
</tr>
<tr>
<td>CP 2</td>
<td>5</td>
<td>1:1</td>
<td>19</td>
<td>86</td>
<td>104</td>
<td>295</td>
<td>15710</td>
</tr>
<tr>
<td>COMP</td>
<td>5</td>
<td>1:1</td>
<td>14</td>
<td>85</td>
<td>99</td>
<td>255</td>
<td>12735</td>
</tr>
<tr>
<td>RAND</td>
<td>5</td>
<td>1:1</td>
<td>14</td>
<td>88</td>
<td>116</td>
<td>335</td>
<td>22880</td>
</tr>
<tr>
<td>CP</td>
<td>5</td>
<td>1:1</td>
<td>14</td>
<td>85</td>
<td>101</td>
<td>260</td>
<td>13005</td>
</tr>
<tr>
<td>CP 2</td>
<td>5</td>
<td>1:1</td>
<td>14</td>
<td>89</td>
<td>113</td>
<td>270</td>
<td>13860</td>
</tr>
<tr>
<td>COMP</td>
<td>5</td>
<td>1:1</td>
<td>7</td>
<td>91</td>
<td>135</td>
<td>220</td>
<td>9835</td>
</tr>
<tr>
<td>RAND</td>
<td>5</td>
<td>1:1</td>
<td>7</td>
<td>97</td>
<td>201</td>
<td>335</td>
<td>19455</td>
</tr>
<tr>
<td>CP</td>
<td>5</td>
<td>1:1</td>
<td>7</td>
<td>88</td>
<td>128</td>
<td>220</td>
<td>9980</td>
</tr>
<tr>
<td>CP 2</td>
<td>5</td>
<td>1:1</td>
<td>7</td>
<td>93</td>
<td>153</td>
<td>240</td>
<td>10160</td>
</tr>
</tbody>
</table>
Performance Analysis

The performance of a program can be analyzed in terms of the performance of the program's individual blocks.

Actual performance is a function of the program's dynamic critical paths. A dynamic critical path differs from the critical paths discussed above, which are static, theoretical critical paths that may be determined at compile-time and do not include any paging delays. A dynamic critical path includes delays caused by page faults. Ideally, the dynamic critical path would be identical to the static critical path, but often, the dynamic critical path length exceeds the static critical path length and may even contain different instructions due to the inclusion of paging delays along different paths.

Consider a block of straight-line code. The execution time for such a block is equal to the length of the dynamic critical path.

The execution time of a loop is a function of the locality pattern's execution time, which is equal to the length of the dynamic critical path. After the locality pattern is established, the execution time of the loop is the product of the dynamic critical path length and the remaining number of iterations, assuming that the locality patterns do not overlap. The execution time for the loop before the locality pattern is established will typically differ slightly from the execution time for the established pattern, since the pages are being paged in for the initial iteration and are affected by references from outside the block. The execution time of the loop is the sum of the initial start-up time and the product of the dynamic critical path length and the number of
iterations.

Consider a loop which has overlapping locality patterns. For the purposes of calculating execution time, the portion of the loop which is overlapped with the next locality pattern appears to execute only the last time. Thus the execution time of such a loop equals the sum of the dynamic critical path length of the pattern's overlapped portion and the product of the dynamic critical path length of the pattern's nonoverlapped portion and the number of iterations. Again a small factor for initial start-up time should be added to the execution time.

Notice that the equation for the execution time of a loop with overlapping locality patterns will reduce to the equation for the execution time for a loop with nonoverlapping locality patterns when the length of the dynamic critical path of the overlapped portion is zero. Also notice that the equation will reduce to the sum of the length of the overlapped portion and initial start-up time when the locality patterns are completely overlapped. Figure 5.1 summarizes the execution time equations.

straight-line code:
\[ E = \text{length of dynamic critical path} \]

loop with nonoverlapping patterns:
\[ E = \text{number of iterations} \times \text{length of dynamic critical path} + \text{initial start-up time} \]

loop with overlapping patterns:
\[ E = \text{number of iterations} \times (\text{length of dynamic critical path of nonoverlapped portion of pattern}) \]
\[ + (\text{length of dynamic critical path of overlapped portion of pattern}) + \text{initial start-up time} \]

Figure 5.1. Execution time equations
The execution time of more complicated constructs are merely combinations of the execution times of the underlying constructs. For example, the execution time of an if-then-else construct equals the execution time of the "if" test plus the execution time of the "then" body or "else" body, whichever is chosen by the "if" test. The "if" test is merely straight-line code; the "then" and "else" bodies may be straight-line code, loops, or combinations. Another example of a complicated construct is a doubly nested loop. Here the execution time is the product of the number of iterations for the outer loop and the execution time of the outer loop, which equals the dynamic critical path length of the code preceding the inner loop plus the execution time of the inner loop plus the dynamic critical path length of the code following the inner loop.

By building up the program's constructs from blocks of straight-line code and loops, the execution time of the entire program may be analyzed. This would provide us with knowledge as to where restructuring could benefit the most.

Note also that the program's execution time is a function of the dynamic critical paths. Minimizing the length of these dynamic critical paths will reduce the program's execution time.

The minimum length of a dynamic critical path is the length of the corresponding static critical path. There are three ways to cause these lengths to be equal.

First, reducing the secondary circuit time to zero will minimize the dynamic critical path length, because there will be no delays caused by page faults. This is probably not a realistic way to minimize the
dynamic critical path length.

Second, increasing the page size sufficiently so that the critical paths are maintained will minimize the dynamic critical path length. This has the unfortunate characteristic of inflating the time-space product and working set size.

Third, increasing the window size sufficiently so that the critical paths are maintained will minimize the dynamic critical path length. This also inflates the time-space product and working set size.

Note that the entire locality need not be maintained to minimize the locality's execution time. Thus page sizes which exceed the size of the critical path will tend to increase the time-space product and maximum working set size without reducing the execution time, assuming the organization is such that the critical path is in one page. This is similar to the argument presented earlier regarding the upper bound for window size.

The critical path organization lends itself to the easy maintenance of critical paths. With a critical path organization, a smaller page size or a smaller window size maintains the critical paths than the page size or window size needed with a compiler or random organization. Thus a critical path organization will result in a smaller time-space product and maximum working set size than the other organizations. The flaw in this argument is that the critical path organization has internal fragmentation, while neither the compiler or random organization has any internal fragmentation. This fragmentation will increase the time-space product and maximum working set size and thereby reduce the gains of the critical path organization.
CHAPTER V. CONCLUSIONS

The principle of locality does apply to programs run in a data flow environment. High-level constructs which are expected to display temporal locality do display temporal locality. The degree to which they display temporal locality varies depending on the details of the particular data flow environment involved and on the way the constructs are implemented. Programs with overlapping locality patterns were excluded from the restructuring experiments due to a lack of the necessary software tools.

The potential for spatial locality exists in programs run in a data flow environment. To realize this potential, restructuring will generally be needed to organize the code according to the flow of data, since most high-level languages are based on the flow of control. A study of the degree to which single-assignment languages are able to exploit spatial locality based on the flow of data is of interest and importance but is left for future work.

The importance of locality, in particular spatial locality, to the performance of a data flow program is highly dependent on the underlying system configuration. If the ratio of secondary circuit time to primary circuit time is very small, the instruction organization doesn't matter, because the penalty for a page fault is very small. The same is true in a sequential environment if the ratio of secondary memory access time to primary memory access time is very small. However, if the ratio is very small in a sequential environment, the reason for having a two-level memory disappears. In a data flow environment, a small ratio does not
reduce the need for a two-level memory due to the need for additional enabling logic in the executable level of memory and the need for arbitration and distribution networks to switch packets between the functional units and the executable level of memory.

Additionally, if the cache is large enough to maintain the entire bodies of loops and if the memory management algorithm is appropriately tuned to maintain the bodies of loops, the effect of the instruction organization on execution time is minimal. This is true because the organization within a loop or program which is resident in the cache during its execution is irrelevant. Also the effect of code outside of loops is minimal because it is executed only once.

Since the cache is expected to be small, the instruction organization becomes important. A random organization executes quickly only because it tends to pull in the entire program and keep it resident. The performance of a random organization is expected to quickly degenerate to a thrashing situation as the available cache size is reduced.

The compiler organization performed quite well compared to the random and critical path organizations. The execution times, maximum cache sizes, and time-space products for the compiler organization all tended to be small. The success of the compiler organization is attributed to the fact that it is organized according to expressions, constructs, and data dependencies. Organizing according to constructs provides a high degree of temporal locality, and organizing according to expressions and data dependencies provides a fairly high degree of spatial locality.

In general, the critical path organization slightly outperformed
the other organizations. This increased performance is due to the increased emphasis on spatial locality and on the attempt to guarantee pre-paging. However, it is questionable whether the work required to restructure a compiler organization based on constructs, expressions, and data dependencies is justifiable. It is an open question how the critical path organization would compare to an organization produced by a compiler for a single-assignment language. If such a compiler produced code which is badly distributed in the address space, similar to a random organization, the critical path organization would outperform it. However, a simpler restructuring technique, such as simply ordering the code according to data dependencies, could provide an organization which would perform just as well.

A critical path organization can provide the minimum execution time with a smaller window size than the compiler or random organizations. This results in a savings in the time-space product, although the maximum working set size needed tends to be comparable among the organizations. The savings in the time-space product are irrelevant if the program is allowed to maintain a working set size equal to the maximum working set size. However, if the cache is shared by several programs or tasks and if the size of the cache partition allowed a particular program is dynamic, the time-space product becomes very relevant. In this case, the importance of the critical path method of restructuring increases substantially.

One factor which was not addressed in the experiments in this study is contention for the data paths, that is, contention for the networks.
It is possible that the performance differences between the critical path organization and the compiler organization would widen if contention for the networks were introduced. The critical path organization is expected to allow better utilization of the arbitration network and possibly the distribution network, because the critical path organization tends to have enabled instructions distributed equally among the pages in the cache. Thus the demand for entry into the arbitration network would tend to be distributed evenly among the ports into the network, thereby reducing contention. A similar occurrence is expected at the interface between the distribution network and the cache. This hypothesis warrants further study.

Some possible modifications to the critical path technique are seen as areas for further research and are discussed below.

The removal of the TEST routine would remove the attempt to force prepaging. The resulting algorithm would restructure solely on spatial locality along critical paths. A performance comparison between programs organized by this algorithm and the same programs organized by the original critical path method could determine to what extent natural prepaging occurs.

The existing critical path method attempts to fill holes by using a best-fit algorithm. For each unplaced instruction in PLACE-LAST, only the pages containing destination instructions and critical predecessors of this instruction are checked for holes. Thus holes within blocks often exist unnecessarily after PLACE-LAST executes. This results in the insertion of more no-op instructions and the ultimate increase in
working set size. Replacing the best-fit algorithm with a first-fit algorithm would reduce the complexity of PLACE-LAST and reduce the number of no-op instructions. However, the resulting effects on execution time and therefore on time-space product are unknown.

All of the restructuring done in the critical path algorithm is within blocks. A phase could be added to additionally restructure these blocks in some way. For instance, a simple coalescing of blocks could further reduce internal fragmentation. Alternatively, a critical path analysis of the blocks could be used to combine blocks. This approach is expected to produce very minimal improvements as long as the size of pages is small relative to the size of blocks.

Other areas of research suggested by this study are the investigation of program behavior when no assumptions are made about the existence of contention, the investigation of data memory references, and the investigation of alternate memory management algorithms for data flow environments.

This study has perhaps raised more questions than it has answered. Several conclusions, however, seem clear. A cache memory is of importance for data flow programs with nonoverlapping locality patterns. Using a page size greater than one instruction reduces the execution time. Using a page size large enough to approach minimal execution time raises the possibility of the existence of an optimal instruction organization. Using the minimal window size to maintain the execution time shows a wide range of possible time-space products which are dependent on the instruction organization.
BIBLIOGRAPHY


APPENDIX A. THE VMM SIMULATOR

The original data flow simulator at Iowa State University was designed for a study of the feasibility of data flow as an architectural design principle. At that time a tool was needed to perform the basic operations of a data flow computer for measurement purposes and to serve as the target machine for the translation of high-level programs into data flow programs. The simulator is not intended to specify the actual implementation of data flow computers.

The VMM simulator is an extension of the original simulator, so some components are not relevant to this study but are merely carried over from the original design. In particular the data structure memory is not relevant and will not be described here.

The elements of the simulator dealing with virtual memory and its management are extensions to the original simulator.

Functional Units

The functional units are special purpose processors which are capable of performing only one type of operation each. To allow the most flexibility in experimenting with different system configurations, the number of each type of functional unit and the primary circuit time of each type of functional unit are parameters to the simulator. The primary circuit time is the time required for an instruction to move through the arbitration network, to be executed in the functional unit,
and to send its result tokens through the distribution network.

The types of functional units presently provided in the simulator are classified below. Most require no further explanation.

Arithmetic operations: +, -, *, /, **, Negate, Absolute
Boolean operations: And, Or, Not
Relational operations: <, >, <=, >=, =, #, Exists, Element, Eos
Structure operations: Append, Select
Input/Output operations: Read, Readedit, Write, Writedit
Procedure operations: Apply
Staging operations: Identity, Merge
Functional operations: Sin, Cos, Tan, Sinh, Cosh, Tanh, Arcsin, Arccos, Arctan, Log, Sqrt
Special operations: Constant

A few operations warrant further description. Exists A,i returns true if there is an ith component in the A data structure. Element A returns true if A is an elementary data value and false if A is a structure. Eos A returns true if A is an end-of-stream token. Append A,i,x makes x the ith component of the structure A. Select A,i returns the value of the ith component of the structure A. Readedit and Writedit perform the formatting operations, such as skipping characters or lines. Identity A returns the value A. The identity operation may be used to fan out a value to several destinations. Constant c,A returns the constant value c when a token A arrives. Merge A,B,G returns A if G is
true and returns B if G is false. The value which is not returned is held as an operand for the next activation of this instruction. Used in cases where a simple feedback signal is not enough to enforce the single token per arc rule, the merge instruction chooses an operand on the basis of a result from a conditional. Merge instructions are used at the bottom of if-then-else constructs to choose the "then" values or the "else" values. They are also used at the top of loops to choose between new values coming in from outside the loop and old values circulating around the loop.

Memories

A prototype of the program resides in a disk file. When a procedure call occurs, a copy of the called procedure is loaded into the instruction memory. This allows the simulator to make more efficient use of available space without affecting the model.

To facilitate this process of keeping procedures in the instruction memory only when they are active, all addresses within a procedure are relative to the procedure's first instruction. This results in the need for base registers. The number of base registers determines the maximum number of procedures which may coexist in the instruction memory. This number is a system parameter.

Each base register includes a free field which specifies whether this base register currently controls a procedure, a procedure id field which specifies which procedure, if any, is using this base register,
an extent field which specifies the size of the area of instruction memory controlled by this base register, and an offset field which specifies the address in instruction memory where the controlled area begins.

Also included in each base register is an activity field. Since there is no one data flow instruction which is guaranteed to be the last instruction to execute, some mechanism is needed to know when a procedure is inactive and can be safely removed from the instruction memory. The activity field provides the necessary information by keeping a count of the active instructions. An instruction is active if it is potentially enabled, enabled, fetched, or executing. If there are no active instructions, the procedure is inactive and may be removed from the instruction memory.

Each base register also includes three measurement fields. A page fault field records the total number of page faults incurred by this procedure during this activation. A memory reference field records the total number of references to this area of instruction cache. A time-space field records the time-space product of this procedure for its cache memory requirements.

The instruction memory is implemented as an array of pointers to instruction cells. Each instruction cell has an opcode segment, from one to three operand segments, and a destination segment. The number of instruction cells possible in the instruction memory is a system parameter.

An opcode segment of an instruction cell consists of an opcode field, a base register field, a control status field, a control receipt
field, and pointers to the operand and destination segments. The base register field specifies which base register manages this instruction. The control status and control receipt fields are used to enforce the single token per arc rule. The control status field specifies the number of feedback signals needed by this instruction. The control receipt field counts the number of feedback signals received. When the number received equals the number needed, the opcode segment is said to be enabled.

Each operand segment in an instruction cell has a gate control field, a gate field, a data type field, a data field, an initial enable count field, an enable count field, and two acknowledge fields.

The gate control field of an operand segment specifies the type of gating needed by the operand. Options for the gate control value are No, Constant, True, and False. A gate control value of No is the normal value and specifies normal execution. A Constant gate control value specifies that the value in the data field is a constant and should not be destroyed when the instruction executes. A True or False gate control value requires that a token arrive in the gate field before the operand is enabled. If the gate control value is True and a "true" token arrives in the gate field, the operand will be enabled when the data token arrives. If the gate control value is True and a "false" token arrives, the operand will be false fired when the data token arrives. False firing an operand means sending feedback signals to the instructions which sent the gate and data tokens and then destroying the tokens. A False gate control value is handled analogously. True and false gating is used to
control execution based on the result of a condition. For example, the initial operands in the "then" body of an if-then construct are true gated. The results of the "if" condition is sent to the gated operands. If the condition is true, the "then" body will execute; if the condition is false, the gated operands will false fire, destroying their tokens and denying execution to the "then" body. Gating is necessary to control execution in such cases because the execution is data-driven. No control instructions, like "branch on zero", exist.

The initial enable count field of an operand segment specifies the number of tokens needed to enable the operand. If the gate control value is Constant, no tokens are needed. If the gate control value is No, one data token is needed. If the gate control value is True or False, one data token and one gate token are needed. The enable count field counts the number of tokens received. The operand is enabled when all of the needed tokens arrive, provided it does not false fire.

The two acknowledge fields are used to store the addresses of the instructions which send the gate token and data token to the operand. These addresses are used to send the feedback signals.

A destination segment of an instruction cell includes a data type field, a base register field, a destination number field, and zero or more destination fields. The data type field specifies the data type of the instruction's result. The base register field specifies which base register to use when storing results. The destination number field gives the number of instructions that need this instruction's result. Each destination field specifies an instruction which needs this result, which
operand within that instruction should receive the result, and whether the result token should be stored in the gate field or the data field.

An instruction is enabled, that is ready to execute, when its opcode segment is enabled, all of its operand segments are enabled, and it is in the instruction cache.

No actual storage separate from the instruction memory is used in the simulator for instruction cells. Instead the instruction cache is simulated by the appropriate use of page tables. Each page table entry has six fields. The enter field records the time at which this page logically entered the cache. This field is used to determine if a page is logically in the cache and to calculate the time-space product. The check field records the time at which the most recent reference to this page occurred. This field is used to determine if this page is in the working set. The base register field specifies which base register controls the area of instruction memory occupied by this page. The activity field in the page table entry is a count of the number of instructions in this page which are enabled but not yet fetched. The use of this count prevents an enabled instruction from being paged out before it is fetched. The page fault field keeps a count of the number of times this page caused a page fault. The memory reference field counts the number of instruction cache references to this page during its current stay in the cache.
Networks

The effect of the networks is simulated by appropriately delaying the fetching of instructions and the storing of result tokens. When an instruction is fetched, a packet containing the information necessary for execution is formed. The packets are linked into a list which is ordered by scheduled completion time. The scheduled completion time is computed when the instruction is fetched by adding the current simulator clock time to the primary circuit time for this instruction type. The scheduled completion time is the time at which this instruction's result tokens will be stored. Other information in the packet includes the opcode, the address of the base register from which this instruction is offset, the number of the instruction from which this packet was fetched, a pointer to the destination segment, and the data types and the data values from the operand segments.

Control

The flow of control through the simulator is pictured in Figure 4.1. The major routines are described below

MAIN:
Initialize the system by giving values to the system parameters and by putting the bootstrap APPLY in the list of packets.

Call DECODER.

While there is activity do;
    Call ENABLER.
    Call LIST-FORMER.
    Call PAGE-OUT.
    Increment the system clock.
Figure 4.1. Flow of control through the simulator

Call DECODER.
End.

Report final paging measurements.

End MAIN.

APPLY (procedure, argument structure, return address):

Find space in the instruction memory for the procedure, reclaiming space if necessary.

Load the procedure into the instruction memory.
Initialize the page table entries for this procedure such that the first page is in the cache.

Set up the return mechanism by storing the return address in the first instruction.

Activate the procedure by storing the argument structure in the second instruction, which is specifically designated for argument passage.

End APPLY.

ENABLER:

For each instruction in the cache do;
   If the instruction is enabled, put it on the list of enabled instructions.

End.

End ENABLER.

LIST-FORMER:

For each enabled instruction do;
   If an appropriate functional unit is available then do;
      Calculate the scheduled completion time.
      Form the instruction packet and link it into the ordered list of packets.
      Send the appropriate feedback signals.

End.

End LIST-FORMER.

PAGE-OUT:

For each page do;
   If there are no enabled, unfetched instructions, and if the page has fallen out of the window then do;
      Reinitialize the page table entry to reflect a missing page.

End.

End PAGE-OUT.

DECODER:

For each instruction scheduled to complete during the current clock cycle do;
Execute the instruction.
For each destination do;
  Store the result token in the instruction.
End.
Remove the packet from the list.
End.

End DECODER.

Whenever a token or feedback signal is stored, a test must be made to determine if the instruction into which the token is stored is in the cache. If the instruction is missing, the appropriate page table entry must be updated to reflect a page fault occurring to bring the page into the cache.

Measurements

The VMM simulator is instrumented to give the following measurements, which are important to this study:

parallel execution time,
number of page faults for each page,
total number of page faults for each procedure,
number of references to each page for each duration in the cache,
total number of references for each procedure,
page reference fringe,
instruction reference fringe,
working set size at each clock step, and
time-space product.
APPENDIX B. EXPANSION OF FORM-PAGES

FORM-PAGES:

Determine the leaf instructions.

Set I to 1.

While there is an untraversed leaf do;
    Find the leaf with the largest completion time.
    Call PUSH_PATH (Leaf, Stack (I), Branch Stack).
    While Branch Stack is not empty do;
        Increment I.
        Pop an instruction off Branch Stack.
        Call PUSH_PATH (Instruction, Stack (I), Branch Stack).
End.

Set J to 1.
While J ≤ I do;
    If the current page is not empty and the entire Stack (J)
    will not fit in the remaining space then update
    current page.
    While Stack (J) is not empty do;
        Pop an instruction off Stack (J).
        Call PLACE (Instruction).
    End.
    Increment J.
End.

For each instruction not yet "placed" do;
    Mark the instruction "placed".
    Call PLACE_LAST (Instruction).
End

Update current page.
End FORM-PAGES.

PUSH_PATH (Node, Stack, Branch Stack):

While Node is not "placed" do;
    Push Node onto Stack.
    Mark Node "placed".
    Set found to false.
    For each of Node's back links X do;
        If X is not "placed" then do;
If found then do;
    Push X onto Branch_Stack.
End.
Else do;
    Set found to true.
    Set Temp to X.
End.
End.
If found then set Node to Temp.
End.
End PUSH_PATH.
APPENDIX C. PROGRAM LISTING FOR EXPERIMENTS

1. PROCEDURE PECR
2. BEGIN
3. REAL A,FV,FX,FM,U,V,W,TOL,EPS;
4. INTEGER N,M,NH,NT,JE,I,IC,NOD,JST,IST,J,LN,OPT,SW,ERROR,FLAG,
5. FLAG1;
6. REAL ARRAY C(1:10),T(1:10);
7. FILE INF,OUTF;
8. INPUT N,TOL,A,OPT FILE=INF FORMAT=I(2),F(5,3),F(5,3),I(2);
9. INPUT (C(J) DO J=1 TO N) FILE=INF FORMAT=F(5,3);
10. SW := 1;
11. EPS := 0.0;
12. M := 0;
13. LN := N;
14. ERROR := 0;
15. IF LN > 0 THEN BEGIN
16. IF OPT # 1 THEN BEGIN
17. FV := 1.0;
18. NH := LN/2;
19. JST := 2;
20. NOD := LN-NH-NH
21. END
22. ELSE BEGIN
23. FV := 0.5;
24. NH := LN-1;
25. JST := 1;
26. NOD := 1
27. END:
28. FM := FV*ABS(IN(A));
29. FX := FM;
30. IF FX # 0 THEN BEGIN
31. FV := 0.5*FX;
32. NT := NH*NH;
33. JE := 0;
34. W := 2.0;
35. I := 1;
36. REPEAT
37. U := 1.0;
38. V := 1.0;
39. T(I) := 1.0;
40. IC := I;
41. JE := JE+NH;
42. I := I+1;
43. J := I;
44. REPEAT
45. IF I > 2 THEN W := T(IC-1);
46. V := V+W;
47. \[ T(J) := V; \]
48. \[ IC := IC+NH; \]
49. \[ U := U+V; \]
50. \[ T(IC) := U; \]
51. \[ J := J+1 \]
52. \[ UNTIL J > JE; \]
53. \[ I := I+NH \]
54. \[ UNTIL I > NT; \]
55. \[ I := 2; \]
56. \[ REPEAT \]
57. \[ C(I) := C(I)\times FX; \]
58. \[ FX := FX\times FV; \]
59. \[ I := I+1 \]
60. \[ UNTIL I > LN; \]
61. \[ IC := NT; \]
62. \[ FLAG := 1; \]
63. \[ REPEAT \]
64. \[ IST := 1; \]
65. \[ I := IC; \]
66. \[ IF NOD \# 1 THEN IST := NH; \]
67. \[ J := LN; \]
68. \[ IF J = 0 THEN FLAG := 0 \]
69. \[ ELSE BEGIN \]
70. \[ U := C(LN); \]
71. \[ IF SW = 1 THEN BEGIN \]
72. \[ W := EPS+ABS(IN(U)); \]
73. \[ IF W > ABS(IN(TOL)) THEN BEGIN \]
74. \[ M := LN; \]
75. \[ I := 2; \]
76. \[ REPEAT \]
77. \[ C(I) := C(I)/FM; \]
78. \[ FM := FV\times FM; \]
79. \[ I := I+1 \]
80. \[ UNTIL I > LN; \]
81. \[ FLAG := 0 \]
82. \[ END; \]
83. \[ IF FLAG = 1 THEN EPS := W \]
84. \[ END; \]
85. \[ IF FLAG = 1 THEN BEGIN \]
86. \[ FLAG1 := 1; \]
87. \[ REPEAT \]
88. \[ I := I-IST; \]
89. \[ J := J-JST; \]
90. \[ IF J > 1 THEN BEGIN \]
91. \[ C(J) := C(J)\times U\times T(I); \]
92. \[ U := -U \]
93. \[ END \]
94. \[ ELSE FLAG1 := 0 \]
95. \[ UNTIL FLAG1 = 0; \]
IF J = 1 THEN C(l) := C(l)+U;
IF OPT = 1 THEN NOD := NOD-1;
IF NOD = 1 THEN IC := IC-NH-1;
LN := LN=1
END
END
UNTIL FLAG = 0
END

ELSE ERROR := 1;
OUTPUT M,EPS FILE=OUTF FORMAT=I92),F(8,5)

END

Input data: 4 .0012.0 14.0 0.0 3.0 1.0

PROCEDURE RUNGE
BEGIN
REAL H,Y,Z,X,RK1,RK2,RK3,RK4,RL1,RL2,RL3,RL4,N,I;
FILE INF,OUTF;
INPUT N FILE=INF FORMAT=F(2,0);
H := 1.0/N;
Y := 0.0;
Z := 1.0;
I := 0.0;
WHILE I <= N-1 DO
X := I/N;
RL := H*2*SQRT(IN(2.71828**(2*X)-Y**2));
X := X+H/2;
RK := H*(Z+RL/2);
RL := H*2*SQRT(IN(2.71828**(2*X)-(Y+RK/2)**2));
RK := H*(Z+RL/2);
RL := H*2*SQRT(IN(2.71828**(2*X)-(Y+RK2/2)**2));
X := X+H/2;
RK := H*(Z+RL3);
RL := H*2*SQRT(IN(2.71828**(2*X)-(Y+RK3)**2));
Y := Y+(RK1+2*RK2+2*RK3+RK4)/6;
Z := X+(RL1+2*RL2+2*RL3+RL4)/6;
I := I+1
END;
OUTPUT X,Y FILE=OUTF FORMAT=F(10,3),F(10,3)
END

Input data: 4
1. PROCEDURE INTEGRAL
2. BEGIN
3. REAL X,Y,K,EP,E1,U,C,A,V,D,B,K1,G,H,T1,T2,T3;
4. INTEGER N,M;
5. FILE INF,OUTF;
6. INPUT X,Y,K,EP FILE=INF FORMAT=F(10,8),F(2,0),I(2),F(3,1);
7. E1 := EP ** 2;
8. U := 1.0;
9. C := 1.0;
10. A := 1.0;
11. V := 0.0;
12. D := 0.0;
13. B := 0.0;
14. N := 1;
15. K1 := K - 1;
16. REPEAT
17. G := U;
18. H := V;
19. N := N + 1;
20. M := N / 2;
21. IF 2*M = N THEN BEGIN
22. T1 := X + (M-K1) * C;
23. T2 := Y + (M+K1) * D
24. END
25. ELSE BEGIN
26. T1 := X + M * C;
27. T2 := Y + M * D
28. END;
29. T3 := T1 ** 2 + T2 ** 2;
30. C := (X * T1 + Y * T2) / T3;
31. D := (Y * T1 - X * T2) / T3;
32. T1 := C - 1;
33. T2 := A;
34. A := A * T1 - D * B;
35. B := D * T2 + T1 * B;
36. U := G + A;
37. V := H + B
38. UNTIL (A**2+B**2)/(U**2+V**2)<=E1;
39. OUTPUT N,U,V, FILE=OUTF FORMAT=I(3),F(10,5),F(10,5)
40. END

Input data: .00000001 1 1 .1
1. PROCEDURE MATRICES
2. BEGIN
3. INTEGER N,I,J;
4. REAL ARRAY A(1:10,1:10);
5. FILE INF,OUTF;
6. REAL T,C,D,F;
7. INPUT N FILE=INF FORMAT=I(2);
8. C := N * (N + 1) * (N + N - 5) / 6.0;
9. D := 1 / C;
10. A(N,N) := -D;
11. I := 1;
12. WHILE I <= N-1 DO
13.   F := I;
15.   A(N,I) := D * F;
16.   A(I,I) := D * (C - F * F);
17.   J := 1;
18.   WHILE J <= I-1 DO
19.     T := J;
20.     A(I,J) := -D * F * T;
21.     A(J,I) := -D * F * T;
22.     J := J + 1
23.   END;
24.   I := I + 1
25. END
26. END

Input data: 5