Table 5: Processing Activity Under Transformed Thread Sequenced Prescheduling

Clock Cycle	PE_1	PE_2	Ready Threads
0	-	-	A
1	A,cm	-	-
2	A,cm	-	-
3	A,1	-	-
4	A,2	-	-
5	A,3	-	-
6	A,4	-	DCF, BE
7	DCF,ch,1	BE,cm-	-
8	DCF,2	BE,cm	-
9	DCF,3	BE,cm	-
10	DCF,4	BE,1	-
11	DCF,5	BE,2	-
12	DCF,6	BE,3	-
13	DCF,7	BE,4	-
14	DCF,8	BE,5	-
15	DCF,9	BE,6	-
16	DCF,10	BE,7	-
17	DCF,11	BE,8	-
18	DCF,12	BE,9	-
19	DCF,13	BE,10	-
20	DCF,14	BE,11	-
21	DCF,15	-	-
22	G,ch,1	-	G
23	G,2	-	-
24	H,cm	-	H
24	H,cm	-	-
25	H,1	-	-
26	H,2	-	-

chy for their application. These results are also very important in the further refinement and definition of the parallel architecture briefly described in this paper. The data dependency graph analysis phase during compile time will provide important configuration information for the architecture, and may be exploited for use in reconfigurable multiprocessor systems.

References

[1] Chang, M.-C. and Lai, F., Efficient Exploitation of Instruction-Level Parallelism for Superscalar Processors by the Conjugate Register File Scheme, *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 278-93.

[2] Andrews, D. L., Application Specific Analysis of Parallel Computing Systems, *Ph.D. Dissertation*, Syracuse University, 1992.

[3] Sarkar, V., Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors, *Tech. Rep. CSL-TR-87-328*, Stanford University, 1987.

[4] Feo, J. T., An Analysis of the Computational and Parallel Complexity of the Livermore Loops, **Parallel Computing**, Elsevier Science Publishers 0167-8191, July 1988, pp. 163-185.

[5] **IF1 An Intermediate Form for Applicative Languages**, Reference Manual Version 1.0, M-170, University of California-Davis, July 31, 1985.

[6] **SISAL: Streams and Iteration in a Single Assignment Language**, Language Reference Manual Version 1.2, M-146, University of California-Davis, March 1, 1985.

[7] Evripidou, P. and Gaudiot, J.L., A Decoupled Graph/Computation Data-Driven Architecture with Variable-Resolution Actors, *1990 International Conference on Parallel Processing.*

[8] Kuck, D., et. al., Dependence Graphs and Compiler Optimizations, *Proceedings of the* 8th *ACM Symposium on Principles of Programming Languages*, January 1981, pp. 207-218.

[9] Sarkar, V. and Hennessy, J., Compile-Time Partitioning and Scheduling of Parallel Programs, *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, July 1986, pp. 17-26.

[10] Graham, R. L., Bounds on Multiprocessing Timing Anomalies, *SIAM Journal on Applied Mathematics*, 17(2), March 1969.

[11] Gilbert, E. J., An Investigation of the Partitioning of Algorithms Across an MIMD Computing System, *Tech. Rep. Note No. 176,* Computer Systems Laboratory, Stanford University, May 1980.

[12] Hornig, D. A., Automatic Partitioning and Scheduling on a Network of Personal Computers, *Ph.D. Dissertation*, Carnegie-Mellon University, 1984.

[13] Simons, B., Sarkar, V., Breternitz Jr., M. and Lai, M., An Optimal Asynchronous Scheduling Algorithm for Software Cache Consistency, *Proceedings of the Hawaii International Conference on Systems Sciences*, pp. 502-511, 1994.

Table 3: Processing Activity Under Thread Prescheduling

Clock Cycle	PE_1	PE_2	PE_3	Ready Threads
0	-	-	-	A
1	A,cm	-	-	-
2	A,cm	-	-	-
3	A,1	-	-	-
4	A,2	-	-	-
5	A,3	-	-	-
6	D,ch,1	C,cm	B,cm	B,C,D
7	D,2	C,cm	B,cm	-
8	D,3	C,1	B,1	-
9	D,4	-	B,2	-
10	D,5	-	B,3	-
11	D,6	-	B,4	-
12	D,7	-	B,5	-
13	D,8	-	B,6	-
14	D,9	-	B,7	-
15	D,10	-	B,8	-
16	F,cm	-	E,ch,1	E,F
17	F,cm	-	E,2	-
18	F,1	-	E,3	-
19	F,2	-	-	-
20	F,3	-	-	-
21	F,4	-	-	-
22	F,5	-	-	-
23	G,ch,1	-	-	G
24	G,1	-	-	-
25	G,2	-	-	-
26	H,ch,1	-	-	H
27	H,1	-	-	-
28	H,2	-	-	-

in that only one additional thread sequence is present after the critical path has been found. Following the transformation, the thread sequences are prescheduled as depicted in Table 4.

Table 4: Prescheduling for Transformed Thread Sequences

PE	Thread Sequence
PE_1	$A \rightarrow DCF \rightarrow G \rightarrow H$
PE_2	BE

As in the above examples, Table 5 illustrates the execution of the transformed graph.

With the graph transformation and prescheduling method, a further reduction in overall runtime is achieved, resulting in $t_{trans} = 26$. The total runtime incurred a memory latency penalty of 4 clock cycles caused by L1 cache misses by PE_1 at clock cycles 1,2,22 and 23 in Table 5. Two of the cycles (numbers

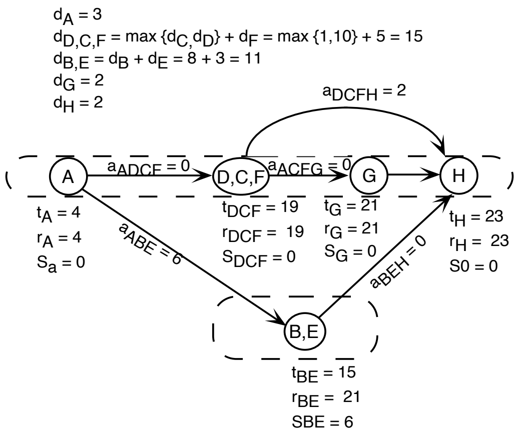

Figure 8: Transformed Example Program Graph

1 and 2) were due to cache cold start misses and are unavoidable. Note, also that the required computation resources were reduced while the overall runtime was improved (i.e. the available parallelism was reduced from 3 to 2). Improved load balancing can also result from these transformations as illustrated by this example. The theoretical speedup is further reduced and is computed in Equation 12.

$$Sp = \frac{t_{trans}}{t_{theor}} = \frac{26}{22} = 1.18 \tag{12}$$

The corresponding performance increase with respect to the single PE case, p_{trans}, is computed and given in Equation 13.

$$p_{trans} = \frac{t_{scalar} - t_{trans}}{t_{trans}} \times 100 = \frac{34 - 26}{26} \times 100 = 30.8\% \tag{13}$$

7 Conclusion

This paper has presented a method for analyzing a data dependency graph in order to compute the theoretical best runtime and to estimate the required maximum number of PEs for a given parallel computer architecture. These results were then used to develop static pre-runtime scheduling and transformation methods for reducing the overall runtime by minimizing memory latency penalties. These methods are efficient and thus can provide a practical optimization phase in the compiler for the defined architecture.

Further areas of improvement include the development of additional transformation rules and a hierar-

Table 1: Processing Activity Under FIFO Scheduling

Clk Cyc	PE_1	PE_2	PE_3	Ready Threads	FIFO Cont.
0	-	-	-	A	1,2,3
1	A,cm	-	-	A	2,3,1
2	A,cm	-	-	-	2,3,1
3	A,1	-	-	-	2,3,1
4	A,2	-	-	-	2,3,1
5	A,3	-	-	-	2,3,1
6	D,ch,1	C,cm	B,cm	B,C,D	2,3,1
7	D,2	C,cm	B,cm	-	2,3,1
8	D,3	C,1	B,1	-	2,3,1
9	D,4	-	B,2	-	2,3,1
10	D,5	-	B,3	-	2,3,1
11	D,6	-	B,4	-	2,3,1
12	D,7	-	B,5	-	2,3,1
13	D,8	-	B,6	-	2,3,1
14	D,9	-	B,7	-	2,3,1
15	D,10	-	B,8	-	2,3,1
16	-	F,cm	E,ch,1	E,F	1,2,3
17	-	F,cm	E,2	-	1,2,3
18	-	F,1	E,3	-	1,2,3
19	-	F,2	-	-	1,2,3
20	-	F,3	-	-	1,2,3
21	-	F,4	-	-	1,2,3
22	-	F,5	-	-	1,2,3
23	G,cm	-	-	G	2,3,1
24	G,cm	-	-	-	2,3,1
25	G,1	-	-	-	2,3,1
26	G,2	-	-	-	2,3,1
27	-	H,cm	-	H	3,1,2
28	-	H,cm	-	-	3,1,2
29	-	H,1	-	-	3,1,2
30	-	H,2	-	-	3,1,2

essentially 'marked' with a PE identifier and can be executed as soon as all predecessor threads have terminated execution. The threads are marked in such a way as to minimize L1 cache miss penalties in an attempt to reduce overall runtime by minimizing memory access overhead.

Using this method, three thread sequences are identified in the example dag and assigned to particular PE as indicated by Table 2. Note that it is still possible to incur a cache miss penalty when a particular thread in one sequence depends upon data computed in a thread from another sequence.

The following table is of the same format as that in the FIFO scheduling example and illustrates execution events for the prescheduling approach.

The thread sequence prescheduling approach reduced the overall runtime to $t_{presch} = 28$. The runtime reduction results in further decreasing the theoretical speedup as calculated in Equation 10.

Table 2: Various PE's and Their Thread Sequence Assignments

PE	Thread Sequence
PE_1	$A \rightarrow D \rightarrow F \rightarrow G \rightarrow H$
PE_2	C
PE_3	$B \rightarrow E$

$$Sp = \frac{t_{presch}}{t_{theor}} = \frac{28}{22} = 1.27 \qquad (10)$$

The corresponding performance increase with respect to the single PE case, p_{presch}, is computed and given in Equation 11.

$$p_{presch} = \frac{t_{scalar} - t_{presch}}{t_{presch}} \times 100 = \frac{34 - 28}{28} \times 100 = 21.4\% \qquad (11)$$

6.3 Prescheduling after Transformation

In this technique, threads are prescheduled as above, but only after additional compile time processing is accomplished on the dag. By exploiting the allowable latency values, a_{ij}, various threads can be combined through graph transformation rules without affecting the theoretical runtime. These transformations are applied to the graph prior to prescheduling if they will result in decreasing memory access latencies.

So far, we have identified three simple transformation rules. However, appropriate graph transformation rules will vary depending on the multiprocessor architecture. The rules given below apply to the simple architecture described in Section 3.

1. If a predecessor thread has more than one consumer thread, transfer as many instructions in the producer thread forward to consumer threads as possible. This will increase parallelism by allowing the 'bottleneck' producer thread to contain as few instructions as possible.

2. If multiple predecessor threads have a single common consumer thread, transfer as many instructions back to the predecessor threads as possible.

3. Combine all tandem threads into a single thread.

Using these rules, the example dag can be transformed as shown in Figure 8. Note that after graph transformation, the exploitable parallelism is reduced

Like the r_{t_i} values, these quantities can be computed through the use of a backward traversal through the dag incurring a computational cost of $O(|V| + |E|)$. The governing equation is:

$$a_{ij} = (r_{t_j} - CPT_j) - r_{t_i} \tag{7}$$

6 Thread Scheduling

The parameters defined in the previous section gave overall and intermediate execution times without regard to clock cycles expended due to memory access overhead. In this section, we will analyze overall runtime considering memory latencies for three thread scheduling models:

1. FIFO PE scheduling
2. PE thread assignment prescheduling method
3. Thread assignment after graph transformation

Each of these scheduling methods will be defined and associated overall runtimes and parallelism measures will be computed in the following subsections. To simplify these results, the following assumptions will apply to each example.

1. Cache miss penalties are 2 clock cycles
2. A cache miss will occur if a PE executes a thread when that same PE did not execute the threads' immediate predecessor
3. A 3 PE architecture is utilized since maximum available parallelism is 3 for the example graph shown in Figure 4

Obviously, these are simplistic assumptions and depend upon other parameters such as the presence of the multi-level cache hierarchy and the size and line fill lengths of the L1 caches. Nevertheless, the overall trend is illustrated. Similar methods have been used in the past for partitioning and scheduling for MIMD machines [11] and distributed computers [12].

6.1 FIFO Scheduling

This method is intuitively simplistic and allows exploitation of available parallelism to occur on a purely data-driven basis. Whenever a thread is ready for execution, a PE is allocated for its execution by accessing a FIFO structure containing pointers to idle processors. This concept has been proposed in the past for various architectures similar to the one described here [7]. In this scheduling scheme, the instruction template and required data may not necessarily be present in the allocated PE's L1 cache causing a possible cache miss penalty in overall execution time.

Table 1 depicts the activity present in the architecture for each clock cycle. The first column indicates the current clock cycle and the next three columns contain thread identifiers corresponding to those in Figure 3 which indicate current execution on a given PE. Some of the thread identifiers have "cm" or "ch" next to them. This indicates whether the data was present (ch = cache hit), or, not present (cm = cache miss) in the respective PE's L1 cache. Each thread identifier also has a number next to it which represents the offset of the PE's program counter register relative to the beginning of the threads' code template. The "Ready Threads" column contains a list of threads that are ready for execution based upon all data dependencies being satisfied. The last column entitled "FIFO Contents" contains the queue contents in order from left to right that indicate which PE will be scheduled for the next available thread.

The execution data presented in Table 1 represents the parallel execution of the example program where the various threads are scheduled using a FIFO and are executed in a purely data-driven manner. Using this approach, it is seen that the theoretical speedup has decreased from that assuming a single PE (given in Equation 3) and is computed by dividing the total runtime using FIFO scheduling, t_{fifo}, by the ideal case as shown in Equation 8.

$$Sp = \frac{t_{fifo}}{t_{theor}} = \frac{30}{22} = 1.36 \tag{8}$$

This decrease in theoretical speedup can alternatively be viewed as a performance increase, p_{fifo}, when compared to the single PE as computed in Equation 9.

$$p_{fifo} = \frac{t_{scalar} - t_{fifo}}{t_{fifo}} \times 100 = \frac{34 - 30}{30} \times 100 = 13.3\% \tag{9}$$

6.2 PE Thread Prescheduling

In this technique, the parameters defined in the previous section are computed at compile time and used to determine thread sequences in the dag. Each of the thread sequences are assigned to a particular PE in the architecture resulting in a PE prescheduling scheme. Each thread, or vertex, in the dag is that are not contained in any previously formed thread. As an example, the critical path in Figure 7 will be marked as *thread sequence [0]* the first execution of the while loop. The remaining nodes C, B, E will then be traversed the second execution of the while loop to find the critical path of the nodes remaining. Nodes B, E will be contained in *thread sequence [1]*, and the remaining node C will form *thread sequence [2]*. After *thread sequence [2]* has been formed, no nodes remain, and the program terminates. Note *thread sequence [1]* and *thread sequence [2]* cannot initiate execution until node A has completed as given in the original dependence graph. Note this analysis also yields the maximum available parallelism by determining the total number of thread sequences which is present in the variable, `i` in the pseudo-code shown above.

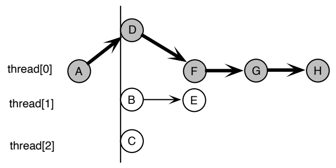

Figure 7: Example Program Graph with Thread Sequences Identified

5 Compile-time Analysis

Once a dag representing a program has been formulated by the compiler, pre-runtime analysis may be undertaken to determine the critical path as defined in the previous section. This section contains the development of a set of parameters and methods for their computation to determine the critical path and other relevant measures. The quantities to be computed are:

- CPT_i = clocks per thread for the i^{th} thread. This represents the total execution time required for one instance of a single thread (or equivalently dag node)
- t_i = cumulative execution time required when a particular thread has completed execution. This value disregards latencies introduced by memory accesses and considers only the CPT_i and data dependence parameters in the dag.
- r_{t_i} = required cumulative execution time. This is the minimum necessary value of the execution time disregarding memory access latencies such that the theoretical runtime can be achieved.
- a_{ij} = allowable latency. The allowable latency is assigned to each dag edge and represents the total amount of latency allowable between execution completion of thread i and the initiation of execution of thread j such that the theoretical runtime is not affected.

Each node or vertex in the dag has a required execution time, CPT_i, associated with the i^{th} thread length. Data dependencies among the various threads are represented by the dag edges, thus a thread j that depends on or consumes data computed in thread i exists in a path from the i^{th} node to the j^{th} node. The cumulative execution time, t_j, becomes the sum of all CPT values for predecessor nodes. This simple sum must be modified slightly in the instance when a consumer node has dependence on two or more immediate predecessors. In this case, the maximum of all predecessor t_i values must be taken to ensure all required data is available. Clearly, the t_n value for the terminal n^{th} node in the dag will be the theoretical runtime, or, execution time of the critical path. Mathematically, the cumulative execution time can be stated as:

$$t_i = CPT_i + max\{t_j\} \forall (v_j, v_i) \in E \qquad (5)$$

Where E is the set of all edges in the dag and v_i is a particular vertex in the dag. The t_i values can be computed for each vertex through a single forward traversal algorithm with computational complexity, $O(|V| + |E|)$, where $|V|$ and $|E|$ represent the total number of vertices and edges present in the dag respectively.

The required cumulative execution time, r_{t_i}, can likewise be computed via a backward traversal of the dag also incurring a computational complexity of $O(|V| + |E|)$. The relationship used to define this quantity is given as:

$$r_{t_i} = min\{r_{t_j} - CPT_j\} \forall j | (v_i, v_j) \in E \qquad (6)$$

Finally, the allowable latencies, a_{ij}, can be computed for each dag edge in the set E. This computation depends on the required cumulative execution times, r_{t_i}, and thus must necessarily occur after they have been computed. The a_{ij} values represent the total amount of latency that may occur between the execution termination of thread i and the execution initiation of thread j, hence these values provide important information for pre-runtime optimizations.

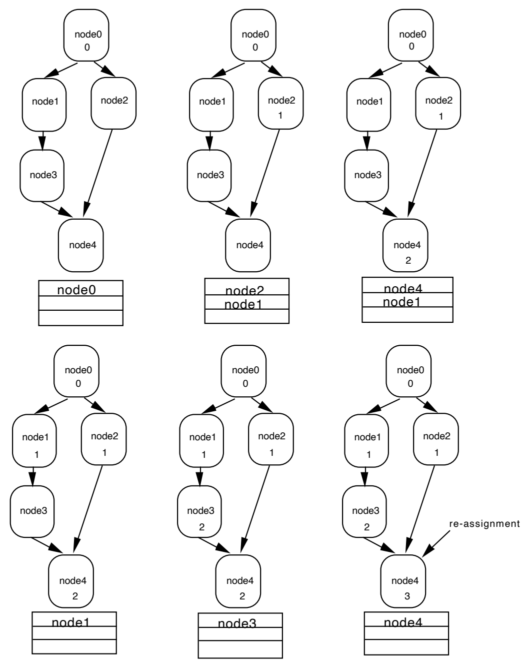

Figure 6: Diagram of Stack and Level Assignments During a Traversal

Lemma 2 *The computational complexity of the graph traversal algorithm is dependent on the number of predecessors for each node. For a directed acyclic graph, the maximum number of predecessors is* $\frac{n(n-1)}{2}$*, for a graph with n edges. The proof follows:*

Proof:

1. Consider a graph $G = (N, E)$ with N nodes. Each node can have at most $N-1$ predecessors (edges) giving at most $N(N-1)$ edges for graph G.

2. Consider two arbitrary nodes n_i and n_j in graph G. If node n_i has $N-1$ predecessors, then node n_j must be an immediate predecessor of node n_i. If we constrain graph G to be acyclic (i.e., no cycles), then node n_j cannot have n_i as a predecessor or a cycle would exist between the two nodes. Therefore an antisymmetric relation exists between the two nodes.

3. Suppose node n_i is from an antisymmetric graph $G = (N, E)$ and has $N-1$ edges. Then a second arbitrary node n_j can have at most $N-2$ edges, a third node n_k at most $N-3$ edges, etc. The maximum number of edges is given by the series:

$$\|E\| = \sum_{i=0}^{N} i = \frac{N(N-1)}{2} \tag{4}$$

□

The graph traversal algorithm considers all forward paths in the graph. The number of forward paths cannot be greater than the number of edges in the graph. Therefore, the worst case computational complexity is $O(E) \cong O(\frac{n(n-1)}{2})$.

4.2 Thread Sequence Formation

Once the critical path is found, the theoretical best execution time t_{theor}, is computed, and the critical path is marked as the graphs' controlling *thread sequence[0]*. The remaining portion of the graph can now be re-evaluated for partitioning onto the remaining $N-1$ processors shown in Figure 7. The algorithm can be stated in pseudo code as:

```
i=0
while(nodes){
  thread_sequence [i] := critical_path()
  i += 1; }
```

where the function `critical` $\leq$ `path()` finds and marks the critical path from the graph for all nodes

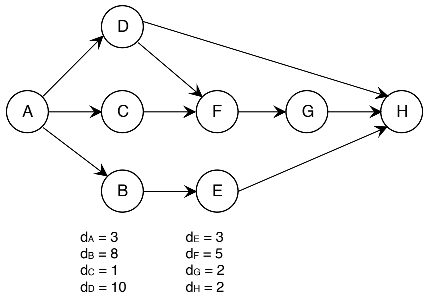

Figure 4: Arbitrary Program Graph

Where n is equal to the number of nodes contained within the graph. Assuming no communications costs, execution of this graph on a scalar machine would take $t_{scalar} = 34$ machine cycles.

Now consider executing the same graph on the machine model shown in Figure 3 with unlimited numbers of processors, and no penalty for communications. For these assumptions, the ideal execution time of this graph will be given by the critical path as shown in Figure 5.

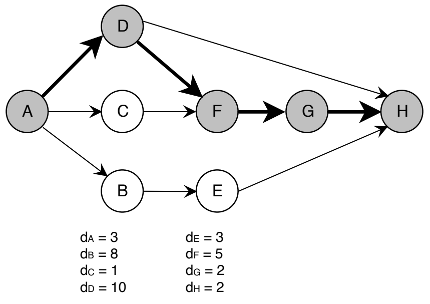

Figure 5: Arbitrary Program Graph with Critical Path Denoted

The execution time of the graph cannot be any faster than the critical path, even with the addition of more processors. This is a general result that can be verified by the following lemma.

Lemma 1 *The execution time of a weighted directed acyclic graph with no communications overhead cost is bounded by the critical path through the graph.*

Proof:
For any given path, p_i, present in the graph from the initial node, a, to the terminal node, n, the total execution time, T_{p_i}, along that path disregarding any communication costs is equivalent to the sum of the individual execution times of each node present in p_i, $T_{p_i} = d_a + \ldots + d_n$. The graph may be viewed as a collection of such paths, $\{p_1, p_2, \ldots, p_i, \ldots, p_m\}$. Hence, the overall program runtime, $T_{overall}$ must be at least equal to the maximum value of the total execution time of all paths forming the graph since it is necessary that all paths (or thread sequences) finish execution for the program to terminate. Since these execution times are computed without regard to communication overhead, the maximum value is a lower bound on the program's execution time given by $T_{overall} = max\{T_1, T_2, \ldots, T_i, \ldots, T_m\}$. $\square$

For the graph given above the critical path is given by nodes $A \rightarrow D \rightarrow F \rightarrow G \rightarrow H$ and is given as

$$t_{theor} = 3 + 10 + 5 + 2 + 2 = 22 \tag{2}$$

Therefore, the theoretical speedup achievable on this program is given as:

$$Sp = \frac{t_{scalar}}{t_{theor}} = \frac{34}{22} = 1.55 \tag{3}$$

This is the theoretical best case execution time and speedup for the arbitrary program represented by the graph above. The addition of more processors or resources cannot provide additional speedup.

4.1 Critical Path Analysis

The critical path can easily be computed using a stack based, depth first traversal of the data dependency graph. This approach is shown in Figure 6. Figure 6 shows a simple data dependence graph, and the corresponding stack contents as the graph is traversed. The first diagram in Figure 6 shows how the stack is initialized by assigning the top node of the graph a level of zero and then pushing the entry on to the stack. The second diagram shows the state of the stack after the top node has been visited by the traversal algorithm. In this case, nodes 2 and 1 were identified as the predecessor nodes of the stack top node, and will be considered for further traversal. The traversal continues by comparing the predecessors current level, given by the integer value shown in each node with d_i plus the level of the stack top node. A predecessor may contain a previously assigned higher level as shown in Figure 6 if it has already been referenced during an earlier traversal. If the existing level is greater than d_i plus the level of the successor node, then the previously assigned level remains valid and the node is not pushed onto the stack. Otherwise, the predecessor node is assigned one plus the successor node's level, and pushed onto the stack. Figure 6 shows the stack and level assignments during a traversal.

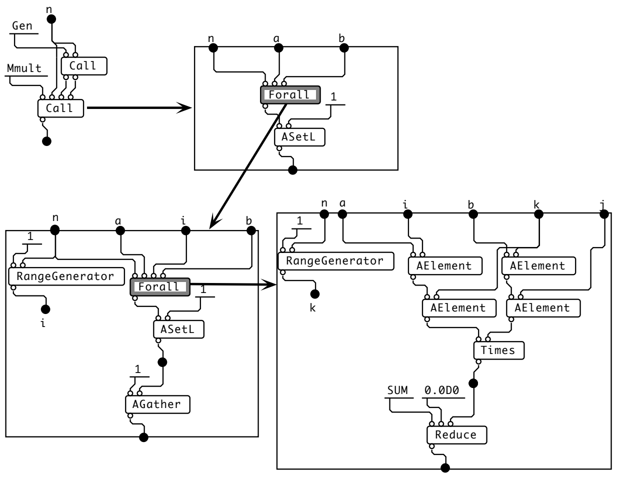

Figure 1: Data Dependence Graph of Program Example

```
type TwoDim = array [ array [ double_real ] ];
function Gen( n : integer returns
                  TwoDim, Twodim )
for i in 1, n cross j in 1, n
returns array of double_real(i)/double_real(j)
        array of double_real(i)*double_real(j)
end for
end function % Gen

function Mmult( n : integer;
                A, B : TwoDim returns TwoDim )
for i in 1, n cross j in 1, n
        c :=    for k in 1, n
                        t := A[i,k] * B[k,j]
                returns value of sum t
                end for
returns array of c
end for
end function % Mmult

function main( n : integer returns TwoDim )
let     A, B := Gen( n )
in      Mmult( n, A, B )

end let
end function % main
```

Figure 2: SISAL Code Sequence Example to Illustrate Parallelism

from both L1 and L2 caches, as well as any interconnect network contention.

4 Ideal Execution Time Prediction

The directed graph shown in Figure 4 below shows the data dependencies between operations for an arbitrary program. The graph can be transformed into a weighted acyclic graph by appending execution times for each node. The edges of the graph represent data movements. An ideal execution time can be computed for the graph by assuming no communication costs between operations. Although unrealistic, this ideal assumption is critical as it provides a hard lower bound on execution time for the graph that can be used for comparisons when actual communications times are introduced. Similar approaches have been used in the past for the computation of runtime bounds [10].

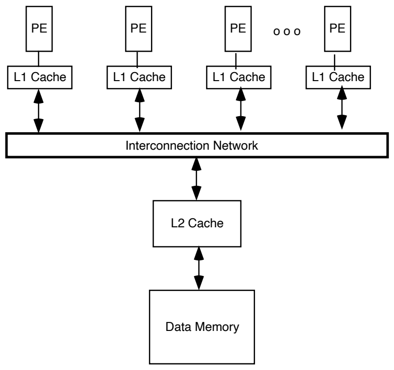

Figure 3: Architecture Block Diagram

As an example, consider the graph in Figure 4 where the execution time for node x is given as d_x. The total number of cycles required to compute this graph on a single CPU is given by the sum of the individual node execution times as:

$$t_{scalar} = \sum_{i=1}^{n} d_i \qquad (1)$$

algorithms to the graph in order to produce more efficient run times, and the ability to graphically display the program based on data dependencies instead of an artificial ordering of operations determined by the placement of an instruction in a textual format. The data dependence graph is a λ calculus based description of a program's dependencies and attributes. Some compilers automatically generate these graphs in intermediate optimization stages. One intermediate graphical form that is used by several different compilers is IF1 [5]. A program can be represented in IF1 as:

Definition 1 *A program is the set of function definitions, $PROG = \{p_1, p_2, \ldots p_n\}$, where function p_i defines a complete graph representing function i's computations, execution times, data dependencies, and operands.*

Definition 2 *A graph is a 5 - tuple (N, P, C, N_c, E_c)*

Definition 3 $N = \{n | n \in \{N_s \cup N_c \cup B\}\}$ *is a set of nodes where:* $N_s = \{n_p | n_p \in$ machine primitive operation $\}$ *is the set of simple nodes.* $N_c = \{n_c | n_c \in$ compound operation $\}$ *is the set of compound nodes where each node n_c in N_c is composed of a set of function definitions $\{p_1, p_2, \ldots, p_n\}$. $B = \{\uparrow, \perp\}$ is the set of boundary nodes. Boundary nodes are special lexical separator nodes. Note that a compound node may be composed of any combination of simple, compound, and boundary nodes.*

Definition 4 *[N, $\leq$] is a partially ordered set (poset) consisting of the set N and the partial ordering $\leq$ representing data and control dependencies between nodes in the node set N.*

Definition 5 $P = \{(n_i n_j) | n_i \leq n_j, i, j\}$ *is the edge set of the precedence relation in Definition 4.*

Definition 6 $N_c : N \rightarrow Z^+$ *such that n_c is equal to the cost of node n_c. For simple nodes, n_c is the (relative) number of machine cycles required to perform the operation. For compounds and the boundary node $\uparrow$, n_c represents the total number of machine cycles required to perform all operations in C_i such that $n_c \in C_i$.*

Definition 7 $E_c : P \rightarrow Z^+$ *such that $e_c(p_{ij})$ represents the communication cost of edge $p_{ij} = (n_i, n_j)$.*

The set of simple nodes represent processor primitive operations including arithmetic, control, boolean, and other indivisible sequential operations. The set of compound nodes consists of conditional constructs, parallel constructs, and iterative constructs. The boundary nodes provide separation between logical groupings of nodes. The precedence relation provides an explicit interpretation of both data and control dependencies. Precedence constraints are directly visible in languages with explicit parallelism, and are easily determined from functional languages with no side effects from explicit data dependencies. Although translation of computations expressed in imperative languages expressible in the λ calculus into a graph form is guaranteed, the translation may require substantial dependence analysis to reveal possible parallelism. The analysis may also include a transformation process that first assumes sequential execution, and removes those dependencies which are provably redundant.

Communication edges represented by the precedence relation $\leq$ are graphical translations of the data dependencies explicitly determined in functional languages adhering to the single assignment rule. However, a communication transformation has to be applied to imperative languages using global storage and reassignment of data values.

As an example, consider the code sequence given in Figure 2 from the Livermore Loops [4] written in the *SISAL* language [6]. The parallelism in the original source language is not obvious. However, the parallelism is immediately obvious from the data dependence graph version of the same program shown in Figure 1. The graph is presented hierarchically, with the graph for the main program showing the calls to function `Gen(args)` and function `Mmult(args)`. Function `Mmult(args)` contains nested `forall` constructs, implying all code contained within the body of the construct can be executed in parallel. A range generator computes the instantiation numbers for each parallel copy of the body code.

3 Architecture Definition

The block diagram of a simplified tightly coupled architecture is shown in Figure 3. Each processing element, PE, contains a local data cache connected across a single interconnection network. Data is shared using global memory. Accessing data from a L1 cache is assumed to introduce no communication overhead. Accesses from the L2 cache incurs an overhead of a cache miss from the L1 cache, and any delays due to the contention across the interconnect network. Accesses from the data memory incur a miss

Graph Analysis and Transformation Techniques for Runtime Minimization in Multi-Threaded Architectures

M. A. Thornton, D. L. Andrews
University of Arkansas
Fayetteville, AR 72701-1201

Abstract

This paper describes a method of analysis for detecting and minimizing memory latency using a directed data dependency graph produced from a compiler. These results are applicable to the development of methods for the optimal generation of instruction threads to be executed on a multi-threaded, data-driven architecture. The resulting runtime reductions are accomplished by minimizing memory access times by individual processing elements. Additionally, these analysis methods can be used to predict measures of achievable parallelism for a given program graph which can be exploited by a reconfigurable, multi-threaded architecture.

1 Introduction

This paper describes a method for the detection and minimization of memory latencies in a data-driven, multi-threaded architecture using a directed graph which contains program data dependency information. This method consists of first analyzing the data dependency graph followed by the application of transformation rules that modify the content of various instruction threads. The data dependency graph is a directed acyclic graph (dag) where the vertices consist of instruction threads and the edges correspond to interdependencies among the threads. By accounting for memory latencies due to the transfer of data among the vertices, the content of the threads can often be modified. The overall result is that instead of relying only upon the inherent parallelism present in the directed graph for runtime minimization, the additional analysis and subsequent program graph transformations allow for further reductions in program runtime by decreasing overall memory access times by individual processing elements.

The analysis and graph transformation methods are shown to have acceptable costs in terms of computational and spatial complexities and are thus practical to implement in the optimization stage of the compiler/loader portion of the system software. Further, this method is devised such that runtime will never be increased through its application; in the worst case the overall program runtime will remain the same.

These results are directly applicable to the development of methods for the optimal generation of instruction threads for the architecture discussed here by statically scheduling the threads prior to program execution through the graph analysis phase and by partitioning the instruction threads to minimize memory latency penalties. Additionally, these analysis methods can be used to predict measures of achievable parallelism for a given program graph before execution. This information can be valuable for the determination of a good organization of processor assets in a reconfigurable, multi-threaded architecture. Similar approaches have been used by other researchers for related applications [1] [8] [9] [13].

The remainder of the paper is organized as follows. The next section will present the definitions used for the mathematical description of the directed acyclic graph used to represent data dependencies among various instruction threads. Next, a brief description of the architecture the methods developed here apply to is included. Sections 4 and 5 discuss the dag analysis methods used to calculate the ideal runtime and the compile/load time optimizations respectively. The application of the analysis results is described in the following section entitled "Thread Scheduling". Finally, conclusions and areas of further work are included.

2 Graphical Representation

This section presents the mathematical foundation taken from [2] [3] for representing programs as directed acyclic graphs (dag). Several advantages exist for expressing programs as directed graphs, including the ability to apply standard graphical analysis