A Technique for Multiprocessor Memory Resource Estimation[*]

J. D. Bullard M. A. Thornton D. L. Andrews

Department of Computer Systems Engineering, University of Arkansas, Fayetteville AR 72701

{jdb2,mat1,dla}@engr.uark.edu
(501) 575-5159
(501) 575-5339 (Fax)

Abstract

Keywords: *Resource Estimation, Data Dependency, Multiprocessor, Performance Analysis, Multiprocessor Architecture, Parallel Processing*

An approach for estimating the required memory resources to execute a program on a multiprocessor system is presented. The technique relies on the information contained in a data dependency graph representing a program to be executed. Data dependency graphs can be generated by a compiler as an intermediate representation of the program without accounting for any specific details of the target machine architecture. Therefore, this approach can be used in the early design phases of multiprocessor architectures for performance analysis of targeted or benchmark applications. Memory resource estimates are separated into two categories; algorithmic and run-time requirements. Experimental results of this technique are presented for several scientific benchmark code fragments.

1.0 Introduction

Processor execution speeds are increasing dramatically due to advancements in integrated circuit manufacturing technology and engineering design methods and tools. However, the rate of speed increase for memory is much smaller. Currently, memory access versus processor latency is at a ratio of approximately 10:1. It is predicted that this ratio will increase to 100:1 in 20 years, a problem known as the processor-memory performance gap [7]. This implies that future architectures must efficiently deal with memory latencies in order to continue to provide machines with performance increases that have been common in the past.

A result of the processor-memory performance gap is that new architectural approaches for designing and implementing multiprocessor computer systems must be utilized. A common theme among several of the various approaches is to distribute and integrate memory with each processor to reduce bus contention, thus allowing concurrent local memory accesses. This leads to the question of determining the amount of memory needed for each processor. Too much memory results in a waste of resources in terms of available chip area and power consumption [7]. Too little memory could impact program runtime. For these reasons an analytical tool for estimating memory resources for given application programs has been developed.

Traditionally, memory requirements have been obtained by 1) assuming an architecture, 2) building or simulating the architecture, and 3) executing an application

[*] This research is supported in part by a grant from the Arkansas Science and Technology Authority under contract 97-B-12.

on the architecture and monitoring memory usage. This approach is disadvantageous for the designer since the design must be in place before the memory requirements can be found.

Other methods for modeling and predicting memory include non-deterministic statistical models [2] [6] which unfortunately still assume some type of architecture. The method discussed here has novelty in that the only requirement is a representation of the application program itself.

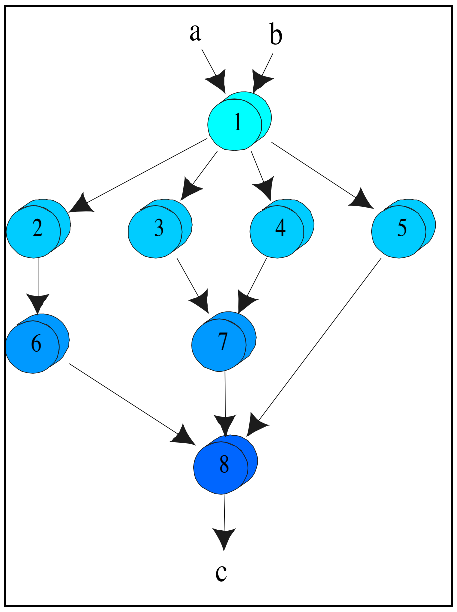


Figure 1 Data dependency graph of $c = (1/a^2) + b^2 - (b + a) - 1$.

The next section describes data dependency graphs and how they are executed on multiprocessor systems. The following sections discuss parallelism and the proposed approach for memory resource estimation. The results of the memory estimation tool are then discussed followed by conclusions and directions of future efforts.

2.0 Data Dependency Graphs

Any given application program can be viewed as a collection of sequential instruction threads to be executed as soon as their input data is available. An abstract representation of a program is then a directed acyclic graph where vertices correspond to computational instructions to be executed and edges represent data dependencies. The edges of the data dependency graph correspond to the transfer of data from the output of a *producer* instruction thread to a *consumer* instruction thread. In a multiprocessor system, each thread is executed on a single processor; therefore, threads can execute concurrently as long as the required input data are available. With this viewpoint, representing an application program as a data dependency graph allows us to exploit the available parallelism.

As an example, Figure 1 shows a data dependency graph which computes a value from the formula $c = (1/a^2) + b^2 - (b + a) - 1$. The graph shows the data dependencies inherent in the computation. For example, node 2 cannot execute until it receives the value of a from node 1. When node 2 executes, it will produce the value of a^2. Node 2 then passes this value to node 6, and so on.

The graph also shows the available parallelism in the computation. For example, given sufficient resources, nodes 2, 3, 4, and 5 can be executed in parallel depending only on the results of node 1. Table 1 summarizes the operations performed by each instruction thread (or graph vertex).

Thread	Operation	Result
1	Retrieve	a,b
2	Square	a^2
3	Subtract	-1-b
4	Square	b^2
5	Negate	-a
6	Inverse	$1/a^2$
7	Add	$b^2+(-1-b)$
8	Add	$(1/a^2)+b^2-(b+a)-1$

Table 1 **Operations for the example data dependency graph in Figure 1.**

3.0 Parallelism Estimation

Consider the case where all threads in a data dependency graph have the same execution time of one clock cycle and execute on a multithreaded multiprocessor with no delay from interprocessor communications or synchronization. If this machine also has unlimited resources (i.e. the ideal parallel machine), then all available parallelism in the program can be exploited. Furthermore, the number of threads executing in parallel at each clock cycle will represent the maximum available parallelism in the program. In this simplified case, the data dependency graph may be viewed as having *levels* of execution, where a level is the collection of nodes executing concurrently during a given clock cycle. This observation has been exploited and a stochastic model has been created to estimate the number of utilized processors per clock cycle [1].

As an example, in Figure 1 the first level would contain node 1 and would execute in one clock cycle. The second level would contain nodes 2 through 5 and would execute in the second clock cycle. Likewise, the third level would contain the nodes 6 and 7 and would execute in the third clock cycle. The fourth and final level would contain only node 8 and would complete execution in the fourth clock cycle.

Another useful metric which can be obtained directly from the data dependency graph is the number of graph edges which enter and leave particular nodes on a per level basis. This information can be used to estimate the memory and bandwidth a system requires to efficiently execute a program.

4.0 Memory Requirements Estimation

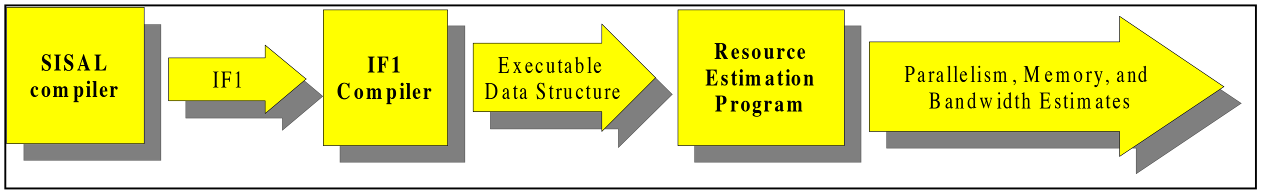
The amount of required local memory can be estimated for a given processor by noting the maximum amount of intermediate storage used during the execution of a program. However, it is important to note that the actual code does not need to be executed to perform this estimation. The required intermediate storage can be obtained by traversing the data dependency graph structure by application of a “graph walk” algorithm.

Consider the case when a data producing instruction thread completes execution but a corresponding consumer instruction thread requires data from the finished thread as well as another independent producer thread that has not yet completed execution. In this case, the data from the finished producer thread must be stored until the consumer thread has all available data and is scheduled for execution. Based on this premise, we have begun developing a tool to estimate the required memory to execute an algorithm on a generic multiprocessor system.

Figure 2 shows the sequence of steps to produce resource estimates from available source code as we have currently implemented the tool. The first step is to compile the source code into IF1, a text file representing a data dependence graph. A SISAL[5] to IF1 compiler exists, as well as IF1 compilers from other high-level languages. The IF1 file is then used as input to the IF1 compiler/profiler described in [8]. The profiler tool extracts the necessary statistics used as input to the memory resource estimation tool.

Level	Parallelism	Nodes	Incoming Arcs	Outgoing Arcs
1	1	1	2	4
2	4	2, 3, 4, and 5	4	3
3	2	6 and 7	3	2
4	1	8	3	1

Table 2 The parallelism and incoming and outgoing arc counts for each level for the example data dependency graph in Figure 1.


SISAL compiler
IF1
IF1 Compiler
Executable Data Structure
Resource Estimation Program
Parallelism, Memory, and Bandwidth Estimates

4.1 Algorithmic Memory Requirements Estimation

We define two types of memory requirements; those due to machine dependent details of program execution, the *run-time memory requirements*, and those due to the structure of the application program's data dependency graph, the *algorithmic memory requirements*. Algorithmic memory requirements are unavoidable and pertain to the structure of the program only. Run-time memory requirements contain the algorithmic memory requirements in addition to the extra amount of memory needed for processor synchronization, communication, and other operating system needs.

The following method is used to estimate algorithmic memory requirements for a given program in a high level language:

1. Compile the source code to IF1.
2. Use the IF1 compiler and profiler to produce a parallelism profile that includes incoming and outgoing arc counts (see Table 2 for an example).
3. Begin a count of memory usage at zero.
4. Step through each level in the parallelism profile, adding the outgoing arcs and subtracting the incoming arcs to find the net memory usage by level. Accumulate these values during each step to determine the current, total memory usage.
5. The maximum (peak) value of the accumulated memory usage is then the memory requirement of the algorithm.

```
% LOOP 1
% Hydro Fragment
% Parallel Algorithm

Define  Main

type double = double_real;
type OneD   = array[double];

function Loop1( n:integer; Q,R,T:double; Y,Z:OneD returns OneD )
   for K in 1,n
       X := Q + (Y[K] * (R * Z[K+10] + T * Z[K+11]))
   returns array of X
   end for
end function

function Main( rep,n:integer; Q,R,T:double; Y,Z:OneD returns OneD )
   for i in 1, rep
      X := Loop1( n, Q, R, T, Y, Z );
   returns value of X
   end for
end function
```

Figure 3 SISAL code for Livermore Loop 1 [2].

Level	Parallelism	Incoming Arcs	Outgoing Arcs	Memory Usage
1	1	0	7920	***4950***
2	2970	2970	2970	3960
3	1980	3960	1980	1980
4	1980	3960	1980	1980
5	990	1980	990	99
6	990	1980	990	0
7	990	1980	990	0
8	1	990	1	0
9	1	1	0	0

Table 3 Parallelism and algorithmic memory requirements for Livermore Loop 1 when the loop index N = 990.

5.0 Results

Table 3 shows the results of the algorithmic memory requirement estimation tool for the SISAL code shown in Figure 3. After the first level of instruction threads is executed, the number of outgoing arcs which must be stored for level 1 is 7,920. Level 2 has only 2,970 incoming arcs leaving 4,950 data items to be stored. The amount of memory required decreases throughout the rest of the execution as those arcs are consumed by other instruction threads, so 4,950 is the peak amount of storage required for execution under these ideal conditions. Therefore, this value represents the algorithmic memory requirements for Livermore Loop 1.

Figure 4 shows the results of the algorithmic memory requirements estimation when analyzing a set of benchmark applications, the Livermore Loops in SISAL [3]. The graph shows only the amount of temporary storage required by the algorithm. The technique does not

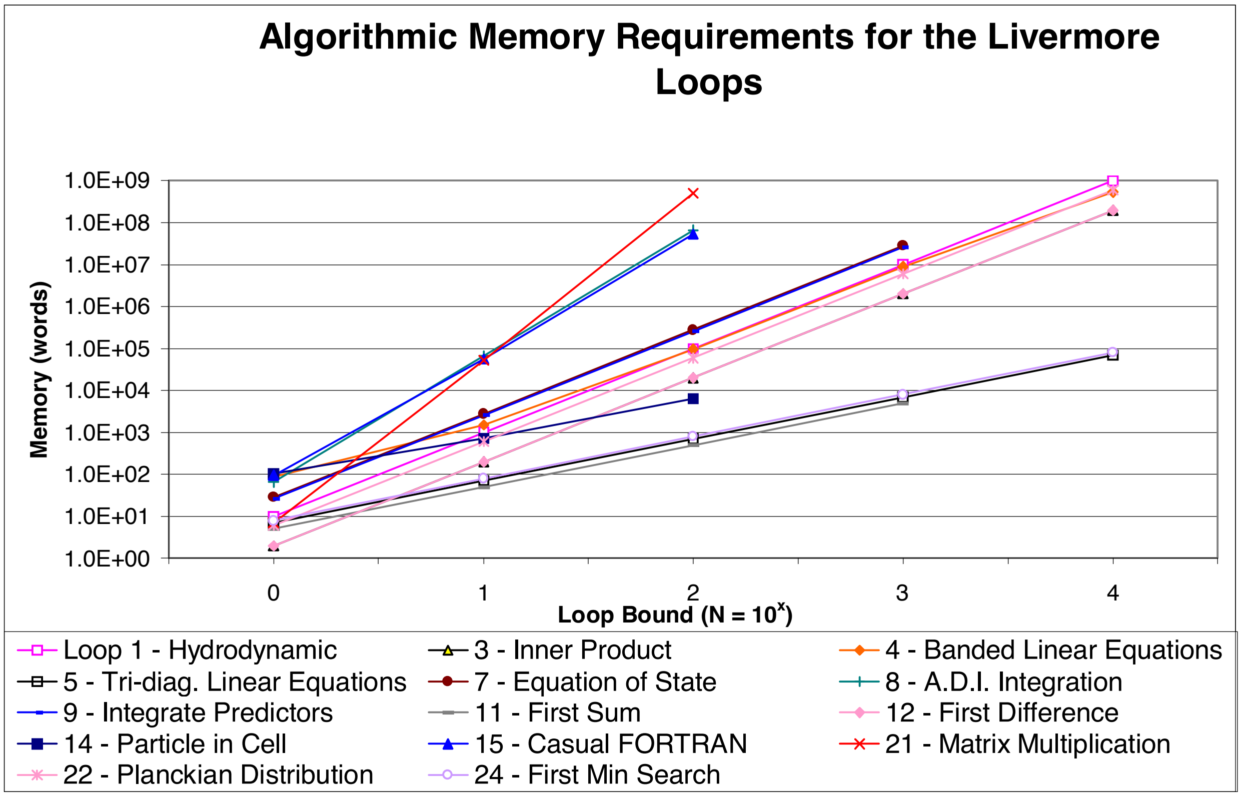


Figure 4 The number of storage locations required to execute several parallel algorithms.

include memory estimates for the storage of machine instructions and other synchronization and communication overhead. Therefore, these are algorithmic estimates, not runtime memory estimates.

The results of the technique illustrate the relationship between memory usage and the loop bound for the Livermore Loops. In all cases this relationship is linear with respect the loop bound, N. This trend is not surprising since we varied only a single bound. We would expect a non-linear relationship if more than one loop bound were varied. It is interesting to note that the different applications in Figure 4 can be characterized by the slope of the memory usage curves, thus validating the notion of algorithmic memory requirements.

6.0 Conclusion

This paper presented an approach for multiprocessor memory resource estimation using only an application's data dependency graph. This approach was implemented leading to the experimental results given. The methodology is suitable for inclusion in a high-level system architecture design package for estimating required memory resources for targeted or benchmark applications. Also, this technique could be incorporated into a "smart" scheduler to utilize available memory efficiently.

The development of a multipurpose resource estimation package has been initiated. To date, a profiler has been developed that produces information containing data structures from an input application's data dependency graph represented in IF1 [8]. A stochastic model based simulator has also been developed based on the profiling information [1] produced by the IF1 tool for estimating processor element work loads.

The current version of the memory resource estimation tool is limited to resource estimation for the ideal case of unlimited available processing elements and equal instruction thread length for all graph nodes. This version is being extended to estimate required memory resources for limited processing elements and variation in execution times for each type of node in IF1.

In addition, the arcs between nodes in the data dependency graphs represent data transfers and require bandwidth between individual processing elements. The memory estimation tool will also be extended to estimate the minimum required total system bandwidth to efficiently execute application software.

Bibliography

[1] D. L. Andrews, M. A. Thornton, and J. D. Bullard. "Multiprocessor Resource Estimation Using a Stochastic Modeling Approach." Eighth Symposium on Parallel and Distributed Processing, *Proceeding of the Workshop on Resource Estimation*, New Orleans, October 1996.

[2] J.P. Diguet, O. Sentieys, J.L. Philippe, and E. Martin, "Probabilistic Resource Estimation for Pipeline Architecture", in *VLSI Signal Processing VIII*, IEEE Press, October 1995, and *Proceedings of the 1995 IEEE Workshop on Signal Processing*, Osaka, Japan, October 16-18, 1995.

[3] John T. Feo. "The Livermore Loops in SISAL." Technical Report, UCID-21159, Lawrence Livermore National Laboratory, August 1987.

[4] John. T. Feo. "An Analysis of the Computational and Parallel Complexity of the Livermore Loops." Elsevier Science Publishers B.V., Series on Parallel Computing 0167-8191/88, #7, 1988.

[5] J. McGraw, S. Skedzielewski, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, R. Thomas. "SISAL: Streams and Iteration in a Single Assignment Language." *Language Reference Manual, Version 1.2, M-146 Rev. 1*, University of California-Davis, March 1985.

[6] H. Jonkers, A. J. C. van Gemund, G. L. Reijns. "A Probabilistic Approach to Parallel System Performance Modelling." *Proceedings 28th Annual Hawaii International Conference on System Sciences*, Vol. II (Software Technology), Wailea, Hawaii, January 1995.

[7] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. "A Case for Intelligent RAM: IRAM." URL: http://iram.cs.berkeley.edu/publications.html. To appear in *IEEE Micro*, April 1997.

[8] Suwanto. *Implementation of Compiler, Viewer, and Parallelism Analysis Software for the IF1 Language*. Master of Science thesis at the University of Arkansas at Fayetteville, May 1997.