Dynamic Dependency Analysis of Ordinary Programs [1]

Todd M. Austin and Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
{austin; sohi}@cs.wisc.edu

Abstract

A quantitative analysis of program execution is essential to the computer architecture design process. With the current trend in architecture of enhancing the performance of uniprocessors by exploiting fine-grain parallelism, first-order metrics of program execution, such as operation frequencies, are not sufficient; characterizing the exact nature of dependencies between operations is essential.

This paper presents a methodology for constructing the dynamic execution graph that characterizes the execution of an ordinary program (an application program written in an imperative language such as C or FORTRAN) from a serial execution trace of the program. It then uses the methodology to study parallelism in the SPEC benchmarks. We see that the parallelism can be bursty in nature (periods of lots of parallelism followed by periods of little parallelism), but the average parallelism is quite high, ranging from 13 to 23,302 operations per cycle. Exposing this parallelism requires renaming of both registers and memory, though renaming registers alone exposes much of this parallelism. We also see that fairly large windows of dynamic instructions would be required to expose this parallelism from a sequential instruction stream.

1 Introduction

Two things generally affect the advance of computer architectures: a better understanding of program execution, and new or better implementation technologies. It is therefore very important to understand the dynamics of program execution when considering the design of future-generation architectures.

To date, most processors have either executed instructions sequentially, or have overlapped the execution of a few instructions from a sequential instruction stream (via pipelining). For such processors, the relevant characteristics of dynamic program execution included operation frequencies, branch prediction accuracies, latencies of memory operations in cache-based memory systems, amongst others. The continuing trend in processor architecture is to boost the performance (of a single processor) by overlapping the execution of more and more operations, using fine-grain parallel processing models such as VLIW, superscalar, decoupled, systolic, or dataflow (in a complete or a restricted fashion). To aid the design of such processors, simple first-order metrics of the dynamic execution, such as operation frequencies and operation latencies, are not sufficient by themselves. What is needed is a thorough understanding of how the operations of a program interact, *i.e.*, what is the nature of dependencies between operations in the dynamic execution graph, how these are impacted by the processor model, and how they impact performance.

The dynamic execution of a program can be best described as a graph, where the nodes of the graph represent operations (or instructions), and the edges in the graph represent dependencies between operations. This paper focuses on a methodology for constructing and analyzing the graph representing the dynamic execution of a program, and considers its application to parallelism studies. In Section 2 we introduce and discuss the various dependencies that exist in program execution, show how they form a graph representing the order of computation in a program, and discuss some of the interesting metrics that can be obtained from this graph. In Section 3 we detail the extraction of these dynamic dependency graphs from the execution trace of a sequential program. Previous work is cited and the implementation of *Paragraph*, our tool for extracting and analyzing the graph, is discussed. Section 4 presents results of parallelism studies on the SPEC benchmarks, and Section 5 presents a summary and conclusions.

2 Program Dependencies and the Dynamic Dependency Graph

Program executions contain many dependencies. These dependencies force a specific order on the execution of the operations. Some dependencies are inherent to the execution of the program and cannot be removed, others can be removed, but usually not without costs in storage and possibly

1

[1] This work was supported by National Science Foundation Grant CCR-8919635.

execution speed.

2.1 Types of Dependencies

True Data Dependencies
Two operations share a true data dependency if one operation creates a value that is used by the other (also called a Read-After-Write or RAW dependency). The dependency forces an order upon operations such that source values are created before they are used in subsequent operations.

Storage Dependencies
Dependencies can also exist because of a limited amount of storage. Such storage dependencies require that a computation be delayed until the storage location for the result is no longer required by previous computation. Storage dependencies are often further classified and referred to as Write-After-Read (WAR) and Write-After-Write (WAW) dependencies. Since many different data values can reside in a single storage location over the lifetime of a program, synchronization is required to ensure a computation is accessing the correct value for that storage location. Violation of a storage dependency would result in the access of an uninitialized storage location, or another data value stored in the same storage location.

Control Dependencies
A control dependency is introduced when it is not known which instruction will be executed next until some (previous) instruction has completed. Such dependencies arise because of conditional branch instructions that choose between several paths of execution based upon the outcome of certain tests.

Resource Dependencies
Resource dependencies (sometimes called structural hazards) occur when operations must delay because some required physical resource has become exhausted. Examples of limiting resources in a processor include functional units, window slots, and physical registers (when renaming is supported.)

2.2 The Dynamic Dependency Graph

The *dynamic dependency graph (DDG)* is a partially ordered, directed, acyclic graph, where the nodes of the graph represent the computation that occurred during the execution of an instruction, and the edges represent the dependencies between the instructions in a dynamic execution of the program.

Figure 1 illustrates a simple program execution trace and the DDG extracted from the trace. The execution trace shows the instructions executed to evaluate the statement `S := A + B + C + D`. In the DDG, the operation at the tail of an edge depends on the execution of the operation at the head of the edge. The only dependencies that exists in this program trace are true data dependencies. This dependency requires that an operation not execute until its source values have been created. For example, the instruction `r4 <- r0 + r1` cannot execute until the value of `A` is loaded into `r0` by the instruction `load r0,A` and the value `B` is loaded into `r1` by the instruction `load r1,B`.

True data dependencies can only be removed by changing the algorithms used in the program and/or with compiler transformations that rewrite the code. Examples of algorithmic changes include the use of parallel (most likely logarithmic) algorithms instead of sequential algorithms (the algorithm used in Figure 1 is actually a parallel summation algorithm). Examples of compiler transformations include loop unrolling, invariant loop code motion, loop interchange, and strength reduction, amongst others.

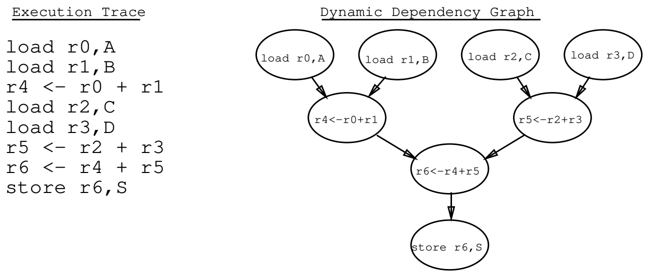

Figure 1: Dynamic Dependency Graph Example

Short of rewriting the program, the true data dependency is the only dependency in a program execution that cannot be removed. If all other dependencies are removed, the resulting graph is simply the *dynamic dataflow graph*.

Figure 2 shows the same computation as Figure 1 but augmented with storage dependencies. The storage dependencies are introduced by the reuse of register locations for different values. The storage dependency edges are annotated with a small, gray bubble. As shown in the figure, the order of execution is further constrained because the subexpression `C + D` cannot begin execution until the subexpression `A + B` has completed using the registers `r0` and `r1`.

Storage dependencies can always be removed by assigning a new storage location to each value created. This is commonly called *renaming*. When using unlimited renaming (assumed for the analysis in this paper), each datum created is a token, and each token is bound to a unique storage location. This results in the program execution having the property of *single-assignment*; that is, a location is assigned to at most once. Figure 1 uses a different storage location for each value created, thus no storage dependencies are created.

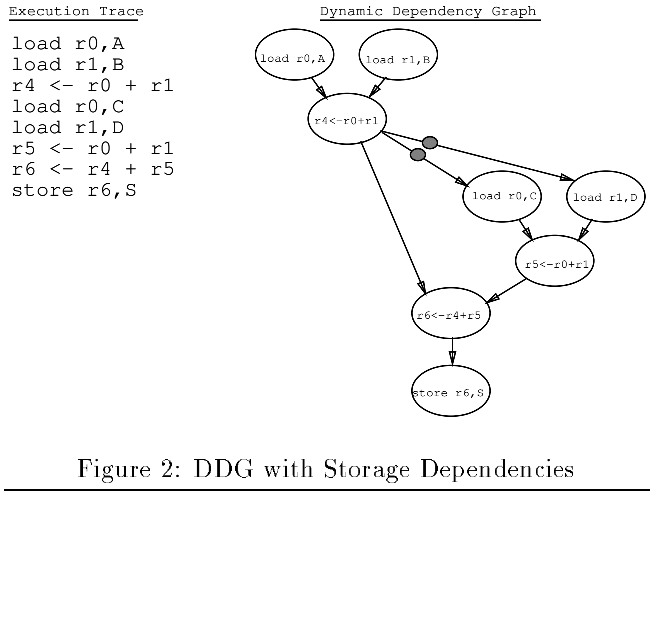

Figure 2: DDG with Storage Dependencies

Removing storage dependencies, via renaming, however, is not without cost. It is likely that during the life of a program, more values are created than could be stored in unique locations in a conventional, high-speed memory. Single-assignment storage allocation would not only be slow and expensive, it would also be wasteful, since a storage location could certainly be reused after the value contained within it was no longer required by another computation. Much research has been done attempting to efficiently implement single-assignment semantics on top of a conventional memory system. Most has been performed in the dataflow literature where the machines directly support single-assignment semantics [1, 6, 11]. The primary techniques used have been self-cleaning graphs, garbage collection, and explicit allocation/deallocation. Self-cleaning graphs can release token storage when the extent is limited to a single entry/single exit body of code; the technique is similar to stack allocation of local variables in imperative languages. Garbage collection is more appropriate when the extent of a variable is not known. The technique tends to be very expensive, and is generally avoided if possible. Often it is avoided by forcing the programmer or compiler to explicitly allocate and deallocate dynamic storage.

To understand how a control dependency could affect the placement of a computation in the DDG, consider Figure 3. The control dependency, shown as a dashed line, is introduced because it is not known which instructions will be executed after the compare and branch instructions until the execution of the `read r1` instruction. The `read r1` instruction reads a value from an input device into the register `r1`. Removing this dependency would require *a priori* knowledge of the input. If a reliable branch prediction method is not available, the `ble` instruction will cause the fetching of instructions to cease until the input has been read. This is illustrated in Figure 3 by the delay of the computation `C + D` until after the `read r1` instruction has completed. Since the compare and branch instructions only provide a mechanism to change the flow of control, and do not create any values, they are not included in the DDG. Operations that are not executed (*e.g.*, `r2 <- r0 + r1`) are also omitted from the DDG.

Figure 4 shows an example of resource dependencies. The processor executing the code fragment contains only two generic functional units (one is required for any instruction execution), thus at most two operations can coexist in any single level of the DDG.

Analyzed Program

```
           load r0,A
           read r1
           cmp r1,0
           ble subtract
           r2 <- r0 + r1
           br continue
subtract:  r2 <- r0 - r1
continue:  store r2,S
           load r3,C
           load r4,D
           r5 <- r3 + r4
```

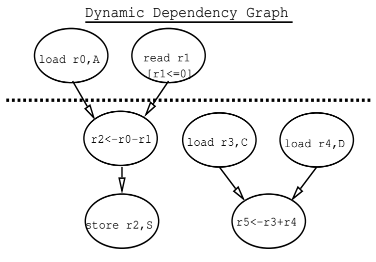

Figure 3: DDG with Control Dependencies

Execution Trace

```
load r0,A
load r1,B
r4 <- r0 + r1
load r2,C
load r3,D
r5 <- r2 + r3
r6 <- r4 + r5
store r6,S
```

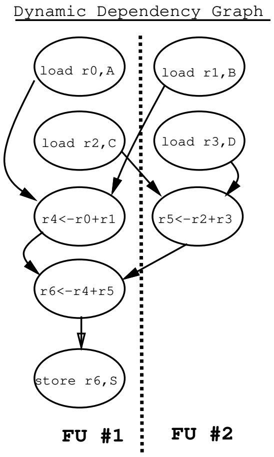

Figure 4: DDG with Resource Dependencies

2.3 DDG Analysis

Given the dependence relations between operations, we can construct the DDG, as illustrated in the previous section. Now let us consider what information can be obtained from the DDG.

The first, and perhaps most valuable, piece of information that we can obtain relates to the true nature of parallelism in program execution. If the DDG is topologically sorted, its height shows the minimum number of steps required to execute the program. This is a function of the dependencies in the graph and is termed the *critical path length* of the particular execution. For example, the DDG in Figure 1 has a critical path length of four, and the DDG of Figure 2 has a critical path length of six. Plotting the number of operations by level in the topologically sorted DDG yields the *parallelism profile* of the DDG. The average number of operations per level for the parallelism profile is the *available parallelism* in the application, and can be viewed as the speedup that could be attained by an abstract machine capable of extracting and executing the DDG from the program's execution trace. For example, the parallelism profile for Figure 1 has four operations in level one, two operations in level two, and one operation in levels three and four; the parallelism profile for Figure 2 has two, one, two, one, one and one operations in levels one, two, three, four, five and six, respectively.

We can also obtain the distribution of value lifetimes from the DDG. The value lifetimes are useful in determining the amount of temporary storage required to exploit the parallelism in the DDG. Likewise, we can also determine the resource requirements of an abstract machine that executes the DDG.

Next, we can obtain the distribution of the degree of sharing of each computed value (or token). Such a distribution is important not only for a possible dataflow realization of our abstract execution engine (by indicating how many operations can be "fired" when a token is created in an explicit token store machine, for example), but also for more conventional multiprocessor execution of the program. In a multiprocessor, different processors are responsible for execution of different parts of a (suitably constrained) DDG of a program. By measuring how much data flows from the nodes in one subgraph to another (albeit in a very constrained form of the DDG that matches the processor execution model), we can measure the degree of data sharing amongst the processors, for example.

By placing suitable constraints on the execution order, or the resources available, we can throttle the DDG to match a particular machine model. Thus, for example, we can obtain parallelism profiles for machines that cannot do storage renaming, or rename only registers, or for machines that have a limited number of ALUs, or for machines that have the ability to look at a fixed-size, contiguous window of instructions.

In Section 4 we shall see how the DDG analysis of the SPEC benchmarks can be carried out along these lines. There we shall consider parallelism studies, and how parallelism is impacted by various constraints placed on the execution of instructions, and consequently on the construction of the DDG.

3 DDG Extraction

Having introduced the concept of the dynamic dependency graph and illustrated the importance of DDG analysis, we now consider how we could construct the DDG of ordinary programs (*i.e.*, programs written in imperative languages), subject to various constraints, and how we could carry out its analysis. Before we describe our methodology, we briefly discuss the previous work in the area.

3.1 Previous Work

There has been a plethora of work in measuring the average parallelism in a (sequential) instruction stream [5, 9, 12, 13, 15]. These studies typically find the length of the critical path through the computation, and compute the average parallelism as the total number of instructions divided by the length of the critical path. (Some also measure the maximum width to determine the maximum number of resources required.) Because they are interested in only a single measure, namely the average or available parallelism, and not in other aspects of the DDG, they do not need to construct the entire DDG, or even parts of it. These studies typically evaluate how the average parallelism changes under various constraints such as register renaming, various branch prediction strategies, memory disambiguation strategies, changes in operation latencies, instruction window sizes, resource constraints, etc. Changes in a parameter result in changes to the critical path length (and the width of the DDG), and consequently to the average (and maximum) parallelism.

To the best of our knowledge, Kumar presented the first work that actually gathered the distribution of parallelism, *i.e.*, the parallelism profile, and the memory requirement profile of serial programs [8]. Kumar extracts parallelism profiles by rewriting FORTRAN programs such that the profile is generated during the execution of the program. Although the method would lend itself to any imperative language, it was only applied to FORTRAN. The operations in the DDG are FORTRAN statements that are assumed to execute in one unit of time. A notable, earlier work is that of Kuck et al. [7] in which program dependencies of FORTRAN programs were analyzed statically, and the resulting available parallelism was estimated from the analyzed code fragments.

A number of papers from the dataflow literature have included examples of DDG analysis [2, 3, 4, 10]. For example, Culler and Arvind [2] provide detailed parallelism profiles and waiting token (or storage usage) profiles for some dataflow programs. Their dataflow processor and language environment lends itself well to dataflow analysis. Instrumenting the dataflow processor's execution is sufficient to generate the parallelism profiles and critical path. Yet their environment lacks storage and most control dependencies, so their results will not include the effects they have on available parallelism. Therefore, it is not clear how their results would extend to programs written in imperative languages such as C or FORTRAN, and also how they could be applied to processors that have more restricted computation models.

Our method of extracting DDGs from serial traces is very similar in spirit to Kumar's method, though the actual implementation is completely different. While Kumar placed FORTRAN statements into the DDG, we instead place machine instructions into the DDG. This allows more precise control over the relative time taken for operations in the DDG, and finer-grain parallelism (such as that found *within* FORTRAN statements) will show up in our results. Moreover, since our technique builds the DDG from a serial execution trace, without modifications to the source program (like Kumar's method), it can be applied widely.

As we shall see in Section 4, our results agree with the conclusions of [3] and [8] that ordinary programs, not intended to execute in parallel environments, do indeed have a significant amount of fine-grain parallelism.

3.2 Paragraph: A Tool for DDG Analysis

All DDGs analyzed in this paper were extracted from serial program execution traces using *Paragraph*. This form of extraction was selected because it is the simplest to perform, and the results are very applicable to the development of processors that must extract parallelism from sequential instruction streams. Yet, extraction from serial traces has caveats that must be made apparent before interpreting the results in Section 4 or comparing them to similar studies that may be done. First, because the trace was generated by a serial program, it is impossible to create totally independent loop iterations (unless loop control instructions are specifically marked and the DDG analyzer can be instructed to treat them differently). Separate iterations of a loop will be connected by at least one recurrence from the loop counter. This results in reducing loop parallelism as the successive independent iterations unroll around the loop counter recurrence. The second caveat involves program transformations performed by compilers. The compiler can actually create a second order effect on the parallelism in the program. For instance, the MIPS compiler commonly performs loop unrolling which tends to decrease the recurrences created by loop counters, thus increasing the parallelism in the program.

Paragraph was developed to extract and analyze DDGs from serial traces produced on DECstation 3100/5000 workstations. The traces were captured with *Pixie*, a basic block execution profiler. *Paragraph* is fully parameterizable. The following parameters can be combined in any combination to see their effects on the parallelism profiles and critical paths.

- System Calls Stall: If this switch is set, an encountered system call is assumed to modify all live values in the program. This causes the placement of all computation after the system call to be in a lower level in the topologically sorted DDG than the system call. If the switch is not set, the system call instructions are assumed to modify nothing, and are ignored.
- Rename Registers: If this is switch is turned on, the storage dependencies on registers are not incorporated into the DDG.
- Rename Data: If this switch is turned on, the storage dependencies for the non-stack segments are not incorporated into the DDG.
- Rename Stack: If this switch is turned on, the storage dependencies for the stack segment are not incorporated into the DDG.
- Window Size: This indicates the number of contiguous instructions in the serial trace that can be considered at any one time when placing values into the DDG. The DDG is created by sliding the window across the entire trace. Once an instruction leaves the instruction window, due to displacement by instructions farther into the trace, it can no longer affect the placement of future instructions into the DDG.

Every trace analysis produces two metrics: the parallelism profile, and the critical path length. The parallelism profile is generated by recording in a hash table the DDG level of each live value. This hash table is called the *live well*. When an instruction is processed, its source values are located, by address or register number, in the hash table (because they are being referenced, they are alive and will be in the hash table). The value created by the newly encountered instruction will be available in the DDG level L_{dest} where:

$$L_{dest} = MAX(L_{src1}, L_{src2}) + t_{op}$$

L_{src1} and L_{src2} are the DDG levels in which the source values are first available for use by another computation, and t_{op} is the time in abstract machine steps (or DDG levels) to complete the operation. If the instruction has no dependencies, *e.g.*, a load immediate, it is placed in the topologically highest level in the DDG. The parallelism profile distribution is updated by incrementing a distribution entry indexed by L_{dest}. When the range of L_{dest} becomes too large to represent each possible value in a distribution, a range of L_{dest} values is mapped to each distribution entry, and in the final output, the average number of operations per level within the range is computed.

Paragraph's implementation is complicated by the need to continually release or reuse value storage in the live well. This is required because a typical program's DDG can contain billions of operations, and thus billions of values will be entered into the live well; many more than can be stored in conventional memories. When a value becomes *dead*, it can be removed from the live well. A value is dead when it will never again be referenced by an instruction in the trace. Two methods can be used to determine if a value is dead.

1. Process the trace in two passes, first in the reverse direction and then in the forward direction. If the instructions are processed in reverse, the first occurrence of a value is its last use, and value lifetime information can be easily inserted into the trace for use on a second, forward pass through the trace. The disadvantage of this approach is that the entire trace must be stored for reverse analysis.
2. Process the trace in a single pass in the forward direction. This method can only determine that a value has become dead after its storage location is reused. While this is not a problem for registers, many memory locations will be used during a program's execution necessitating the need to track many potentially dead values.

All the analyses of the SPEC benchmarks in Section 4 were performed in the forward direction, but a very large memory (32 MBytes) was required to hold the working set of *Paragraph*. A very space efficient hash table was used to minimize the per value memory overhead.

Two special cases must be handled when processing instructions. First, if an instruction uses a value that existed when the program began execution, *i.e.*, a pre-initialized register value, or a value from the DATA segment, the value is placed in the live well such that it was created in the level immediately preceding the topologically highest level in the DDG (the variable *highestLevel* records the topologically highest level in the DDG). This placement prevents a preexisting value from delaying any future computation placed in the DDG. Any values created using only the preexisting value will still be placed in the topologically highest level.

The second special case is the handling of system calls. *Paragraph* does not determine the exact side effects of a system call. Therefore, it must either assume that the system call modified all live values in the program, or that it modified nothing. The conservative assumption (all values modified) is implemented by placing a *firewall* in the DDG. A firewall prevents any later encountered instruction from executing (or firing) higher in the DDG than the firewall. The firewall must be placed immediately after the deepest computation in the DDG. This location is recorded by the variable *deepestLevelYetUsed*, and is possibly updated each time a computation is placed in the DDG. After placing a firewall in the DDG, the variable *highestLevel* is updated to the level immediately following the firewall. To ensure no computation is placed in the DDG prior to the last firewall, the placement function is redefined to be:

$$L_{dest} = MAX(L_{src1}, L_{src2}, highestLevel) + t_{op}$$

The firewall can also be used to represent the effect of a mispredicted conditional branch, resulting in all operations after the conditional branch being placed into the DDG with a control dependency to the firewall. Figure 5 shows the state of the live well after processing the trace in Figure 1. The edges between the records in the live well are not explicitly stored. They are only included to show the order in which values were placed into the live well. The value at the head of an edge was used as a source value to the operation that created the value at the tail of the edge. Each record contains

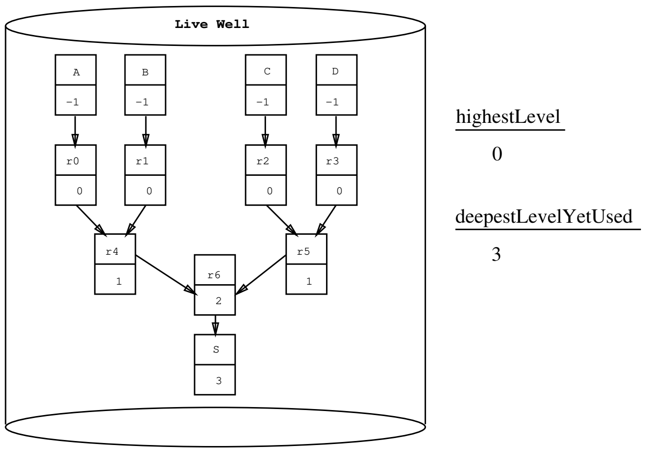

Figure 5: Live Well State After Processing Trace

the name of the location and the level in which the value was created. The variables `A`, `B`, `C`, and `D` are pre-initialized values from the DATA segment, and therefore are placed into the live well at level -1, the level immediately preceding the topologically highest level in the DDG.

The addition of storage dependencies can be implemented by further constraining the placement of computation in the DDG:

$$L_{dest} = MAX(L_{src1}, L_{src2}, highestLevel, D_{dest} + 1) + t_{op}$$

D_{dest} is the deepest level in the DDG of any computation that used a previous value stored in the same location as the destination value. Given this strategy of placement in the DDG, values stored in the same location will have non-intersecting lifetimes in the DDG.

Paragraph can optionally limit the number of viewable instructions when building the DDG. Figure 6 shows a conceptual view of the instruction window. The instruction window passes along the entire trace allowing at most W instructions to be viewed at any one time. As instructions enter the window, they are placed into a DDG being built in the window. Instructions ahead of the window are not considered for placement until they enter the instruction window. The first level available for placement is always the level at the bottom of the instruction window. As the instruction window moves along the trace, instructions displaced from the window can no longer affect the placement of other instructions. This is implemented by including a firewall with the operations displaced from the instruction window. The firewalls are shown as vertical dashed lines in Figure 6. The resulting DDG cannot contain more than W operations in any single level, *i.e.*, the case where all the instructions in the window are independent.

4 SPEC Benchmark Analysis

To demonstrate DDG analysis of ordinary programs, we present the results of some parallelism availability studies, using a DDG analysis of the SPEC benchmarks [14]. The effects on available parallelism due to system calls, renaming, and instruction window size are explored.

All traces were generated using *Pixie* on DECstation

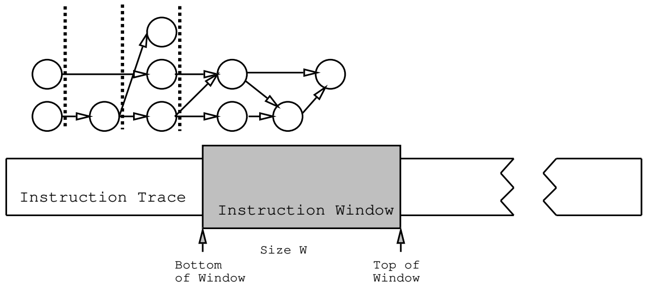

Figure 6: Paragraph Instruction Window

3100/5000 workstations. All the benchmarks were compiled with the MIPS 'C' and FORTRAN compilers, both version 2.1, at optimization level '-O3'. Table 1 shows the instruction latencies (in DDG levels) for each operation class in the MIPS processor. These values are used to determine how many levels an operation will span in the DDG before the value it creates is available for use by subsequent operations.

Table 1: Instruction Class Operation Times

Operation Class	Steps
Integer ALU	1
Integer Multiply	6
Integer Division	12
Floating Point Add/Sub	6
Floating Point Multiply	6
Floating Point Division	12
Load/Store	1
System Calls	1

Table 2 details the SPEC benchmarks, their arguments, and the inputs used. At most 100,000,000 instructions were traced due to time restrictions during the writing of this paper. Analyzed instructions are taken from the start of the program's instruction trace. The benchmarks *cc1* and *espresso* were completely analyzed.

To give an upper bound on the available parallelism, we first analyze DDGs containing only true data dependencies. Table 3 shows the critical path length, and available parallelism for the SPEC benchmarks. Figure 7 shows graphically the parallelism profiles for all benchmarks analyzed (note that the x and y scales differ for each benchmark.) The results in Table 3 are given for both conservative and optimistic system call assumptions; the results in Figure 7 are for conservative system call assumptions. System calls are relatively infrequent, the most frequent case being cc1 with a system call roughly every 14,861 instructions. Comparing the *Conservative* and *Optimistic* columns indicates the range in which the actual value of available parallelism lies. The conservative system call assumption does not hide much parallelism. For most of the benchmarks, the maximum absolute measurement error is very small, with *cc1* being the largest at 32%. For each analysis, all renaming is enabled, the window size is the same size as the trace (no control dependencies), and no functional unit resource restrictions are imposed.

The critical path lengths in Table 3 show the absolute

Table 2: SPEC Benchmarks Analyzed

Benchmark Name	Source Language	Benchmark Type	Arguments	Input	Total Instructions In Trace	Instructions Analyzed
cc1	C	Int	-O	explow.i	59,313,327	59,313,327
doduc	FORTRAN	FP	none	doducin	1,619,374,300	100,000,000
eqntott	C	Int	-s -.ioplte	int_pri_3.eqn	1,241,913,236	100,000,000
espresso	C	Int	-t	opa	119,134,865	119,134,865
fpppp	FORTRAN	FP	none	natoms4	2,396,679,406	100,000,000
matrix300	FORTRAN	FP	none	none	2,766,534,109	100,000,000
nasker	FORTRAN	FP	none	none	919,571,920	100,000,000
spice2g6	FORTRAN	Int and FP	none	greycode.in	28,696,843,509	100,000,000
tomcatv	FORTRAN	FP	none	none	1,872,460,468	100,000,000
xlisp	C	Int	none	li-input.lsp	1,234,252,567	100,000,000

Table 3: SPEC Benchmark Dataflow Results

Benchmark	Number of	Conservative		Optimistic		Maximum
Name	System Calls	Critical Path Length	Available Parallelism	Critical Path Length	Available Parallelism	Measurement Error
cc1	3,991	1,321,698	36.21	903,622	52.95	0.32
doduc	428	877,872	103.59	848,052	107.22	0.03
eqntott	44	109,088	782.52	78,774	942.35	0.16
espresso	91	742,678	132.97	560,225	176.26	0.25
fpppp	30	49,240	1,999.86	48,484	2,032.78	0.02
matrix300	34	4,191	23,302.60	2,839	33,748.58	0.31
nasker	23	1,885,077	50.97	1,884,388	50.99	0.00
spice2g6	1,849	746,124	111.45	600,633	138.44	0.19
tomcatv	24	17,008	5,806.13	14,559	6,800.33	0.15
xlisp	3,470	5,650,548	13.28	5,640,833	13.30	0.00

minimum number of steps required to evaluate the computations in each benchmark. The available parallelism (and the parallelism profile) is computed only from instructions that create values (and thus are placed in the DDGs and parallelism profiles,) and does not include control instructions such as jumps and branches. (The average parallelism would be higher if these were included.) The results indicate that there is a significant amount of parallelism available in the applications. In fact, significantly more parallelism than was shown in [15] (compare the results with the "Perfect" results in [15]). However, as we will see, exploiting the available parallelism will require memory renaming and very large instruction windows for many of the benchmarks. Of course, perfect control flow and memory disambiguation is assumed in the dataflow analysis of Table 3 and Figure 7, whereas [15] places some limitations on their analysis that are representative of the current state-of-the-art.

Xlisp stands out as the benchmark with the least amount of parallelism. Further investigation revealed that the benchmark executes a Lisp program (li-input.lsp) that spends nearly all of its time in a *prog* structure. This construct provides an imperative programming framework in Lisp. The Lisp interpreter implements an abstract serial machine for the prog structure, thus re-introducing the control dependencies that are normally removed by *Paragraph*. The control dependencies show up as recurrences in the updating of the prog structure program counter. The resulting program's DDG is riddled with data dependencies, and thus provides little parallelism.

Time constraints prohibited us from running all the benchmarks to completion. Had we done so, we believe that the benchmarks with large amounts of parallelism (*i.e.*, few inter-instruction dependencies) would have continued to show an increase in the available parallelism, as independent instructions would be placed into the same DDG levels as earlier placed instructions. Benchmarks with smaller amounts of parallelism (*i.e.*, many inter-instruction dependencies) would probably reveal approximately the same amount of available parallelism, as dependent instructions would be placed into new levels of the DDG. Of course, later phases of a program could be very much unlike earlier phases, possibly exhibiting much more, or much less parallelism. This issue remains to be investigated.

Obtaining the parallelism shown in Table 3 will not be possible if storage and control dependencies cannot be removed. First consider the necessity to remove storage dependencies in order to expose parallelism. Table 4 shows the available parallelism when all storage dependencies (register and memory) are left in the DDG, when only memory storage dependencies are left in the DDG (registers renamed), when only non-stack memory storage dependencies are left in the DDG (registers and stack renamed), and when no storage dependencies are in the DDG. For all experiments, system calls force the insertion of a firewall, the instruction window size is the same size as the trace, and there are no functional unit resource restrictions.

The results illustrate that renaming must occur, otherwise the available parallelism in the trace will be substantially reduced. Without register renaming, very little parallelism is detected. This is to be expected since the use of a limited number of storage elements, namely 32 registers, for buffering the results of computations is a severe impediment to

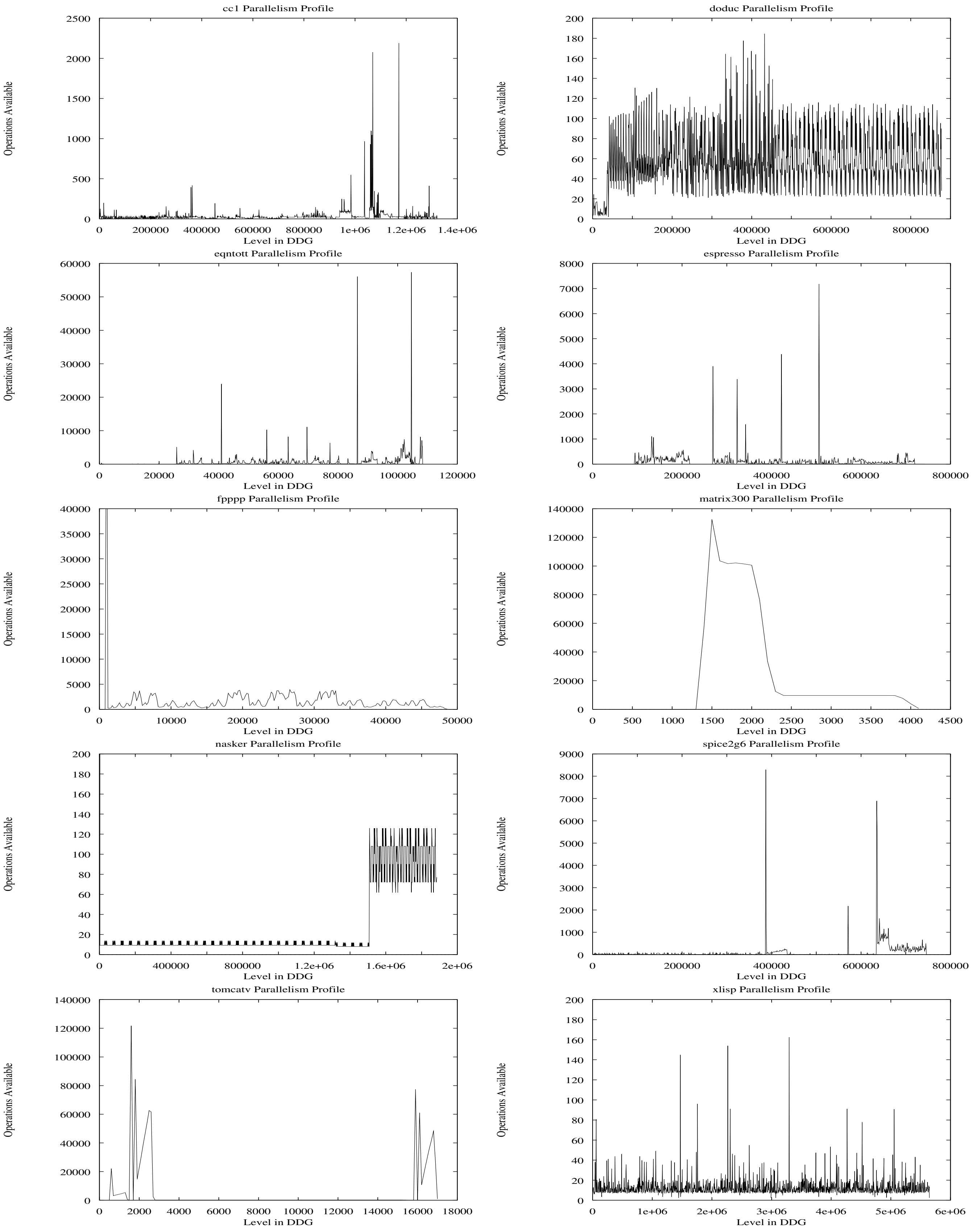

Figure 7: Parallelism Profiles for the SPEC Benchmarks

Table 4: SPEC Benchmarks under Different Renaming Conditions

Benchmark Name	Available Parallelism			
	No Renaming	Regs Renamed	Regs/Stack Renamed	Reg/Mem Renamed
cc1	3.65	33.70	36.19	36.21
doduc	1.62	29.97	103.59	103.59
eqntott	3.67	532.69	538.87	782.52
espresso	2.53	42.46	42.49	132.97
fpppp	1.69	18.34	81.32	1,999.86
matrix300	2.05	1,235.74	23,302.59	23,302.60
nasker	2.58	50.84	50.85	50.97
spice2g6	1.85	39.67	57.36	111.45
tomcatv	1.52	66.63	5,772.38	5,806.13
xlisp	3.32	13.27	13.28	13.28

parallelism. In most cases, renaming registers is enough to expose a sizable fraction the parallelism in the trace. The exception being *matrix300* and *tomcatv* where many of the values (vectors) used are not allocated to registers. The register renaming analysis assumed an infinite number of physical registers were available to rename logical registers.

As mentioned earlier, it is generally easier to rename variables resident on the stack, since their extent is known to be the same as the procedure using the values. If the stack and register values are renamed, only slightly more parallelism is exposed except for *tomcatv* and *matrix300*. Both *tomcatv* and *matrix300* manipulate arrays allocated on the stack. Since all values stored on the stack are renamed, including array accesses, the results are optimistic because it is unlikely that a stack based renaming technique could effectively rename array accesses.

The parallelism values presented in Table 3 and Figure 7 also assume that the entire trace (100,000,000 instructions in most cases) can be searched to find independent operations. For our next study, we vary the window size. This will demonstrate how many instructions in the dynamic instruction stream will have to be exposed before significant parallelism is realized. Figure 8 shows the percent of total available parallelism exposed as a function of instruction window size (note both axes are logarithmic scales.) Each point in the graph represents a full DDG extraction and analysis of up to 100,000,000 instructions (and requires approximately 10 hours on a DECstation 3100.) For each result, system calls force the insertion of a firewall, all renaming is enabled, and there are no functional unit resource restrictions. The instruction window size, described in Section 3, determines how much of the instruction trace is viewable when building the DDG.

Figure 8 indicates that very extensive look ahead, on the order of 100,000 instructions, will be required to garner all the available parallelism in *cc1*, *doduc*, *espresso*, *nasker*, and *spice*. For the applications with a large amount of available parallelism, *i.e.*, *eqntott*, *fpppp*, *matrix300*, and *tomcatv*, the number of instructions needed in the window greatly increases. For *matrix300*, only 3.8% of the total available parallelism (an average of 875 operations per level or cycle in the DDG) is exposed with a window size of 1,000,000 instructions. However, modest levels of parallelism (about 7-52 operations per cycle), certainly enough to fuel the next several generations of superscalar processors, can be obtained for all benchmarks with window sizes as small as 100 instructions.

It is unlikely that conventional superscaler designs could ever remove all control dependencies and completely exploit the parallelism in large instruction windows. This is because 1) the branch predictors currently available are not accurate enough to expose even hundreds of instructions, and 2) resolving dependencies between more than even a hundred instructions would require a prohibitive amount of associative logic. It is apparent that other methods of exposing independent instructions and resolving dependencies will be required.

5 Summary

In this paper, we presented a methodology for building and analyzing the dynamic dependency graph (DDG) of a program from a sequential execution trace of the program. The DDG of a program is a partially ordered, directed, acyclic graph where the nodes of the graph represent computation that occurred during the execution of the program, and the edges represent dependencies that force a specific order on the execution of the instructions. We also illustrated how the analysis of a (suitably constrained) DDG can yield valuable insight into the dynamics of program execution.

We then applied our DDG analysis methodology to study the parallelism in the SPEC benchmarks (for executions of 100 million instructions in most cases). We constructed parallelism profiles, measured the length of the critical path of computation through the programs, and measured the average (or available) parallelism. These studies indicated that there is a useful amount of parallelism in the benchmarks, ranging from 13 to 23,302 operations per cycle, but to fully expose this parallelism requires large instruction windows as well as the ability to rename both registers and memory. Renaming only registers exposed much of the parallelism in most of the cases. The parallelism profiles indicate that parallelism is bursty, with periods of lots of parallelism followed by periods of much less parallelism.

We also saw how the exposed parallelism was influenced by the size of the window of dynamic instructions. If we are interested in only small amounts of fine-grain parallelism (say less than 10 or 20 operations per cycle), then window sizes of a few hundred instructions are sufficient, but for larger levels of parallelism, much larger window sizes (on the order of thousands or tens of thousands of instructions) are required.

We feel that for a quantitative analysis of future-generation fine-grain parallel architectures, DDG analysis is

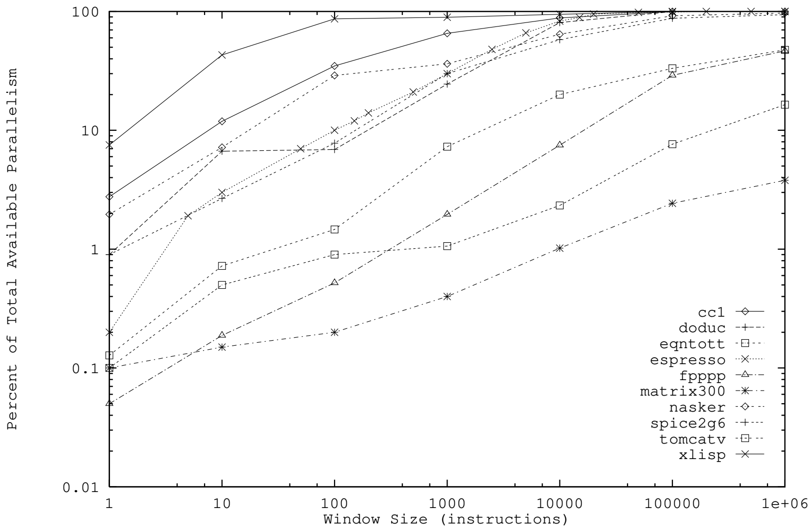

Figure 8: Window Size vs. Parallelism

going to be essential in understanding the dynamics of program execution on such machines, and strongly advocate the use of such analysis methods, despite their costs in execution time and resource requirements.

References

[1] Arvind, S. Brobst, "The Evolution of Dataflow Architectures from Static Dataflow to P-RISC," MIT Computation Structures Group Memo 316, August 1990.

[2] Arvind, D. E. Culler, "Resource Requirements of Dataflow Program," In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 141-150, May 1988.

[3] Arvind, D. E. Culler, G. K. Maa, "Assessing the Benefits of Fine-grained Parallelism in Dataflow Programs," In *Proceedings of Supercomputing 88*, pp. 60-69, November 1988.

[4] Arvind, R. S. Nikhil, "A Dataflow Approach to General-purpose Parallel Computing," MIT Computation Structures Group Memo 302, July 1989.

[5] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, M. Shebanow, "Single Instruction Stream Parallelism Is Greater than Two," In *Proceedings of the Eighteenth Annual Symposium on Computer Architecture*, pp. 276-286, May 1991.

[6] D. E. Culler, G. M. Papadopoulos, "The Explicit Token Store," In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 82-91, May 1990.

[7] D. J. Kuck, et al., "Measurements of Parallelism in Ordinary FORTRAN Programs," Computer, vol 27, pp. 37-46, January 1974.

[8] M. Kumar, "Measuring Parallelism in Computation-Intensive Scientific/Engineering Application," IEEE Transactions on Computers, C-37, 9, pp. 1088-1098, September 1988.

[9] A. Nicolau, J. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures," IEEE Transactions on Computers, C-33, 11, pp. 968-976, November 1984.

[10] R. S. Nikhil, "The Parallel Programming Language Id and its Compilation for Parallel Machines," MIT Computation Structures Group Memo 313, July 1990.

[11] G. M. Papadopoulos, "Implementation of a General Purpose Dataflow Multiprocessor," MIT Laboratory for Computer Science TR-432, December 1988.

[12] M. D. Smith, M. Johnson, M. A. Horowitz, "Limits on Multiple Instruction Issue," In *Third International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 290-302, May 1991.

[13] G. S. Tjaden and M. J. Flynn. "Detection and Parallel Execution of Parallel Instructions," *IEEE Transactions on Computers*, C-19 (10), pp. 889-895, October 1970.

[14] J. Uniejewski, "SPEC Benchmark Suite: Designed for Today's Advanced Systems," *SPEC Newsletter*, Fall 1989.

[15] D. W. Wall, "Limits of Instructional-Level Parallelism," In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176-188, April 1991.