The Limits of Instruction Level Parallelism in SPEC95 Applications

Matthew A. Postiff, David A. Greene, Gary S. Tyson and Trevor N. Mudge
Advanced Computer Architecture Lab
The University of Michigan
{postiffm,greened,tyson,tnm}@eecs.umich.edu

Abstract

This paper examines the limits to instruction level parallelism that can be found in programs, in particular the SPEC95 benchmark suite. It differs from earlier studies in removing non-essential true dependencies that occur as a result of the compiler employing a stack for subroutine linkage. This is a subtle limitation to parallelism that is not readily evident as it appears as a true dependency on the stack pointer. In this paper we show that its removal exposes far more parallelism than has been seen previously. We refer to this type of parallelism as "parallelism at a distance" because it requires impossibly large instruction windows for detection. We conclude with two observations: 1) that a single instruction window characteristic of superscalar machines is inadequate for detecting parallelism at a distance; and 2) in order to take advantage of this parallelism the compiler must be involved, or separate threads must be explicitly programmed.

1 Introduction

True dependencies can be classified as program dependencies (specified in the algorithm) and what we will refer to as *compiler-induced dependencies.* These dependencies are true dependencies introduced during compilation to support the high-level language abstractions. While they cannot be distinguished from program dependencies by the processor, modification of the run-time support generated during compilation can eliminate many of them. In this study we extend previous limit studies of register and memory renaming [Wal91, Wal93, AS92, TA97]. We also examine the effects of eliminating perhaps the most critical compiler-induced dependency—the allocation of activation records on a stack.

In Section 2 we explain the presentation of our results. We examine the effects of eliminating false dependencies through renaming in Section 3 and discuss additional improvements through recognition and elimination of compiler-induced dependencies in Section 4. In Section 5 we discuss methods of exploiting the parallelism exposed in this study and in Section 6 we conclude and present several areas of further research. Table 1 shows the summary results. For a full discussion of previous work and the results of this study, see [PGTM98].

2 Methodology

In order to examine the available parallelism, we constructed an execution driven simulator based on the the Simplescalar simulation environment. We use the SPEC95 becnhmark suite with train inputs.

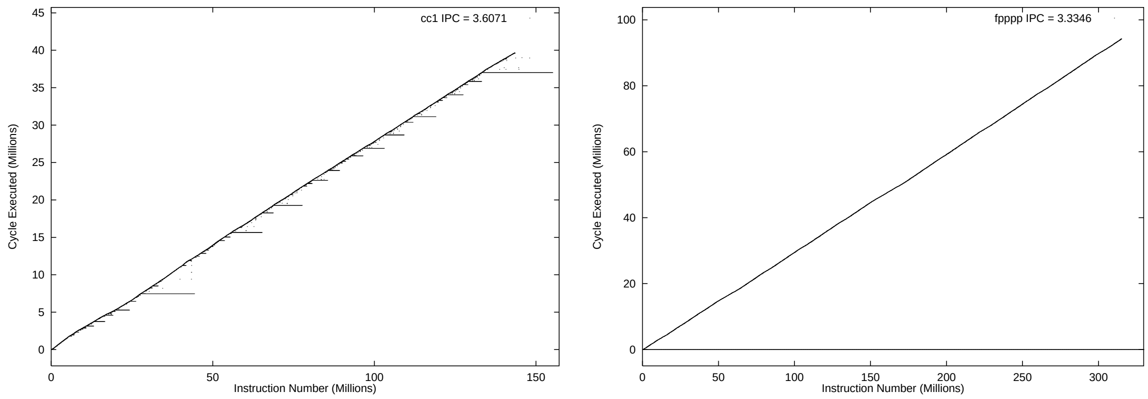
In order to evaulate the performance of different configurations, we rely primarily on the measure instructions per cycle (IPC). In addition, to understand what is happening during benchmark execution, we constructed graphs which we call "execution profiles." The execution profile plots instruction execution cycle on the y-axis versus the instruction's dynamic number on the x-axis. This representation shows when each instruction in a program is free of dependencies and can be executed.

Each point represents the execution of a particular instruction at a particular cycle. All the points intersecting a horizontal slice through the graph represent instructions that are executed in parallel in the simulation, and thus indicate parallelism that could be exploited by an ideal processor. There is only one dot on the graph for any given vertical slice, since each instruction has a unique dynamic number.

In the actual graphs shown in the succeeding sections, individual points cannot be discerned because the long x- and y- axes are compressed to fit conveniently onto the page.

3 Effects of Renaming

In this section we examine the effect of renaming on ILP. Figure 1 displays execution profiles for gcc and fpppp with no renaming. The IPC is shown in the upper right corners of the graphs. The gcc profile contains shading to the right of the main cluster. This indicates that even without renaming, instructions that



(a) gcc

(b) fpppp

Figure 1: Execution Profiles without Renaming. Note the y-axis scales.

are far apart in program order (on the order of millions) can in fact be executed simultaneously.

Register renaming is used to eliminate write-after-read (WAR) and write-after-write (WAW) dependencies. While this is effective for removing false dependencies in a limited range, it does not remove such dependencies when they occur through memory. To model memory renaming we eliminate WAR and WAW dependencies through memory. Only true dependencies are considered when calculating the issue cycle of an instruction.

Figure 2 presents the execution profiles of gcc and fpppp when both register and memory renaming are enabled. Notice that the y-axis scale for both applications has decreased by over an order of magnitude when compared with 1. This shows that register and memory renaming uncover significant parallelism.

4 Compiler Interactions

After eliminating all false data dependencies (along with all resource and control dependencies), it is reasonable to assume that *a* basic limit in the parallelism inherent in a program has been achieved. This is true from the perspective of the processor, which sees only the binary representation of the program. However, to support language abstractions, the compiler includes additional instructions which can, in themselves, limit the parallelism in aggressive machines.

Chief among the compiler generated dependence chains is the stack pointer register used to allocate function activation records. The use of a stack pointer generates a very long true dependence chain as the stack register is continually modified — twice for each function allocating space for an activation record.

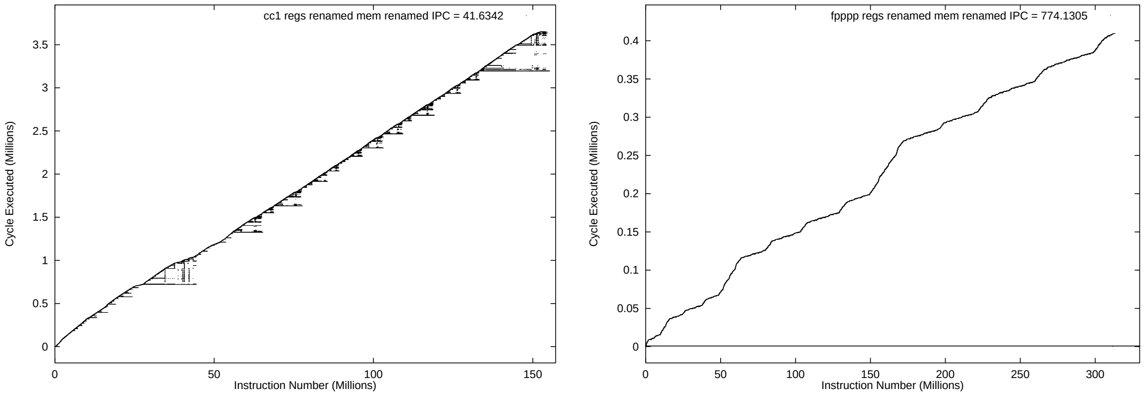
Note that the stack has two separate effects on exploitable ILP. The first, false dependencies through reuse of stack space, is solved with memory renaming. This section deals with the second, namely the artificial true dependence chain generated by the compiler to decrement and increment the stack pointer for each function call.

In Figure 3, the execution profiles for gcc and fpppp are again plotted; this time in addition to the elimination of anti-dependencies and output dependencies to both the register file and memory, true dependencies updating the stack pointer register (R29 in Simplescalar binaries) are removed. The parallelism uncovered by removing the stack pointer is far beyond that uncovered by perfect prediction and renaming. The large shaded area under the gcc graph indicates that instructions must be handled from very distant regions in the dynamic instruction stream.

Table 1 summarizes the IPC values found in each of the infinite instruction window configurations. The final column shows the effect of limiting the instruction window (to 10,000) in the least constrained model (both register and memory renaming with all stack register dependencies removed). With a limited instruction window, the IPC of most applications is limited to a value approximately equal the register-renaming-only model. Even this impossibly-large single window cannot take advantage of the parallelism exposed by memory renaming and elimination of stack dependencies — independent instructions are simply too distant for even an unrealistically large, single instruction window to capture. Multiple smaller windows may be able to capture this parallelism.

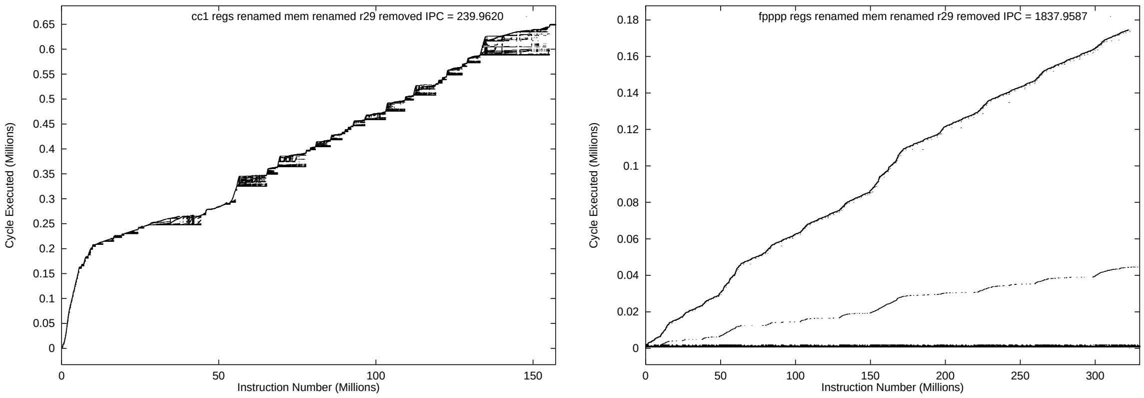
5 Finding Distant Parallelism

In the previous sections, we have shown that the available parallelism in the SPEC benchmarks is considerably higher than previous limit studies and orders of



(a) gcc (b) fpppp

Figure 2: Execution Profiles with Register and Memory Renaming. Notice the change in y-axis scales.



(a) gcc (b) fpppp

Figure 3: Execution Profiles with Stack Pointer Removed. The y-axis scales have dropped by another order of magnitude.

magnitude better than can be achieved with current technology.

Current processor designs exploit instruction level parallelism across a narrow window of program execution; these systems have no ability to identify very distant independent instructions. What design changes must occur to enable this parallelism at a distance to be exploited? One option is to compile to a multi-threaded representation. This would enable a conventional multi-processor to exploit parallelism, but places significant burden on the compiler to identify independent threads. The compiler rarely has complete knowledge of data dependencies (especially through the memory system) so it is impossible to make a static guarantee of the independence of threads.

Many early languages (e.g. early versions of Fortran) used fixed allocation of function variables, while some current languages are implemented using heap allocated activation records to support multi-threaded execution models. The best known example is Sun's implementation of the Java virtual machine. While a fixed frame is not reentrant, heap allocation is a viable alternative to stack based activation records — trading higher overhead (in instructions required to allocate space) with the ability to perform allocations in parallel.

It may be possible to achieve the effects of heap-based allocation of activation records without abandoning the efficiency of stack-based allocation by determining the maximum stack depth a procedure call may require. Often the call depth is known at compile time. This information can be used to reserve the required maximum amount of stack space before initiating the call. Code following the call can be exe-

Table 1: Benchmark IPC for All Configurations

Benchmark	**IPC No Renaming**	**IPC Register Renaming**	**IPC Memory Renaming**	**IPC r29 Removed**	**IPC 10K Window**
compress95	3.12	26.25	73.88	226.33	18.89
gcc	**3.61**	**39.79**	**41.63**	**239.96**	**86.45**
go	2.50	49.15	53.77	141.46	70.71
ijpeg	2.41	55.47	93.60	94.11	52.94
li	3.56	19.60	19.61	81.45	27.70
m88ksim	2.76	19.93	62.06	363.26	20.50
perl	3.47	82.01	127.57	153.05	128.84
vortex	4.57	26.26	26.27	271.97	92.04
applu	2.82	106.65	2037.61	2076.06	78.67
apsi	3.6	54.89	183.44	1224.86	79.56
fpppp	**3.33**	**103.62**	**774.13**	**1837.96**	**134.62**
hydro2d	3.09	144.80	147.67	242.08	52.14
mgrid	3.34	1876.11	3933.03	4003.44	286.48
su2cor	3.22	38.21	34.81	55.56	47.60
swim	3.10	112.08	112.08	275.21	89.15
tomcatv	3.61	32.85	61.47	119.67	58.91
turb3d	3.42	370.98	482.24	3652.46	–
wave5	3.25	29.28	35.71	35.71	–

cuted immediately (assuming no other dependencies). In effect, the single runtime stack is partitioned into multiple stacks used by independent procedure invocations.

6 Conclusions

The limit studies reported here show that a marked increase in exploitable parallelism can be seen by successive addition of register renaming, memory renaming and removal of the compiler-induced dependency on the stack pointer.

These results provide new evidence that there is significant parallelism in applications which are traditionally thought to be sequential. However, the parallelism cannot be exploited by a traditional out-of-order superscalar microarchitecture because the distance between parallel instructions is too great for a single-window implementation to discover them. Therefore a processor that executes from multiple instruction streams will be required to uncover this parallelism. We feel that a hybrid approach exploiting some local parallelism along with the more distant parallelism is the only chance to dramatically increase the overall parallel execution within a single application.

This work was supported by DARPA contract DABT63-97-C-0047. The simulation facility was provided through an Intel Technology for Education 2000 grant.

References

[AS92] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In David Abramson and Jean-Luc Gaudiot, editors, *Proc. ISCA-19*, pages 342–351. ACM Press, May 1992.

[PGTM98] Matthew A. Postiff, David A. Greene, Gary S. Tyson, and Trevor N. Mudge. The limits of instruction level parallelism in spec95 applications. In *ASPLOS VIII INTERACT3 Workshop*, October 1998.

[TA97] Gary S. Tyson and Todd M. Austin. Improving the accuracy and performance of memory communication through renaming. In *Proc. Micro-30*, pages 218–227, December 1997.

[Wal91] D. W. Wall. Limits of instruction-level parallelism. In *Proc. ASPLOS-4*, volume 26, pages 176–189, April 1991.

[Wal93] David W. Wall. Limits of instruction-level parallelism. Technical Report DEC-WRL-93-6, Digital Equipment Corporation, Western Research Lab, November 93.