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. Apart from using a more recent version of the SPEC 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. Other methods can be used that do not employ a stack to remove this dependency. 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

There are three principal types of dependencies that limit parallelism in programs: resource dependencies, control dependencies, and data dependencies. Resource dependencies include such things as the number of function units, size of caches, degree of pipelining, and the instruction window size. They are the most difficult class of dependencies to characterize, because limits associated with them are subject to the rapid improvement that we have come to expect with semiconductor technology.

Control dependencies arise from the need to evaluate conditional branches and computed jumps before execution can proceed. The usual approach for reducing this dependency is to predict the outcome of branches. In past studies the effects of different success rates for prediction have used actual predictors or Monte Carlo techniques. Limit studies in which the predictor can refer to an oracle to give it perfect prediction have also been studied. Our studies will assume perfect prediction.

Data dependencies are induced by the need to preserve the order of reads with respect to writes to storage locations. After anti-dependencies and output dependencies are removed by renaming, true dependencies remain. These true dependencies can be further classified as program dependencies (resulting from data interaction in the particular algorithms employed by the application) 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 will examine the effects of eliminating perhaps the most critical compiler-induced dependency—the allocation of activation records on a stack.

Section 2 discusses previous limit studies on ILP. In Section 3 we present our methodology and a detailed simulator description. We examine the effects of eliminating false dependencies through renaming in Section 4 and discuss additional improvements through recognition and elimination of compiler-induced dependencies in Section 5. In Section 6 we discuss methods of exploiting the parallelism exposed in this study and in Section 7 we conclude and present several areas of further research.

2 Previous Studies

Early studies by Tjaden and Flynn showed that 2-3 instructions per clock (IPC) were possible [TF70]. However, these studies did not consider the possibilities of looking past a branch and were therefore bound by basic block sizes, which are typically four to eight instructions. The importance of branches as a limit on parallelism was more fully explored by Riseman and Foster [RF72]. The recognition of the importance of branch prediction as enabling parallelism lead to a significant effort in branch prediction that continues today [LS84, YP92]. It is now possible to achieve prediction rates that are correct more that 95% of the time. The possibility of high prediction rates also prompted limit studies that considered speculative execution scenarios that extended beyond the basic block boundaries.

Many of the earlier limit studies [Wal91, JW89, Wal93, BYP+91] also removed WAR and WAW dependencies in the case of registers by modeling renaming. The use of register renaming hardware in current processors has given added relevance to these studies [Kel75].

Work reported by Wall [Wal93] showed that there was a potential for IPC greater than 60 if perfect prediction was assumed. To achieve this limit it was also necessary to assume that caches were perfect, that data dependency analysis could be done essentially instantaneously and that register renaming was supported.

Removing false memory dependencies is practically difficult for two reasons: 1) addresses are usually computed, so the task of determining whether two memory references are aliased to the same location is difficult; and 2) memory is large so renaming schemes like those employed for registers would be impractical. The use of algorithm information by the compiler can make the problem tractable for large classes of address calculation, and thus it has made sense to assume a solution to the aliasing problem in limit studies. Perfect alias analysis was studied in [Wal93].

Memory renaming was considered in a limit study in [AS92]. IPC as high as several thousand was reported. Recent developments that employ relatively small value files to rename locally live memory addresses have shown that memory renaming is not as impractical as was thought [TA97]. Such studies have shown promise in reducing the effect of false memory dependencies and allowing more memory references to execute out-of-order. We extend previous work in memory renaming to the limit case to explore the possible gains of an unrestricted memory renaming model.

These limit studies gave an optimistic picture. Other studies that considered the complexities of the hardware needed to detect data dependencies between instructions, the complexities of fetching noncontiguous instructions from memory, and gathering multiple data items from memory in single cycles, arrived at much more pessimistic results that suggested 2-3 as a realistic limit for IPC [JW89, SJH89]. Nevertheless, the limit studies showed that the limitations were one of physical implementation, not logical limitations, and thus provided a realistic goal for implementers.

In fact, before some of these studies there was already a significant body of research that proposed to eliminate the implementation complexities through compiler analysis. The VLIW work was one of the best examples of this approach. Scheduling techniques were developed in which branch prediction was essentially done by the compiler so that it could assemble long execution traces. In [NF84] IPC of tens and in some cases hundreds are reported for benchmarks of DO-loops style programs. Even earlier studies by Kuck's group at the University of Illinois showed that 16 or more processors could be kept busy on workloads characterized by FORTRAN DO-loops [KBC+74].

Furthermore, not all researchers who considered the attenuation that results from implementation complexities were as pessimistic as [JW89] and [SJH89]. In [BYP+91], as the title suggests, the authors argue a strong case for implementation complexities being less limiting. Indeed, in the past few years manufacturers have now started to produce machines with the ability to issue six instructions per cycle, with more on the horizon [Gwe97].

This work is focused on logical limits rather than on implementation issues (we use unlimited window size and perfect branch prediction, for example). We demonstrate a form of "parallelism at a distance" which is the possibility to execute instructions which are hundreds of thousands or millions of instructions away from each other. A single window model of parallelism is not scalable to this level of parallelism.

Research on very wide processors have shown that large instructions windows can provide significant wins in performance [SV97]. However, no single instruction window, no matter how large, will be able to capture the parellelism we demonstrate in this paper–the distances between independent instructions is simply too vast.

As mentioned earlier, branch prediction allows a machine to look further down a (potential) instruction stream to extract parallelism. Non-perfect branch prediction has the same effect as a reduced instruction window size, restricting the machine to utilize local parallelism exclusively. With imperfect branch prediction, the instruction window will be filled with useless instructions instead of more distant instructions that could be executed in parallel independent of the branch outcome. This is especially true when multiple branch predictions are attempted in a single cycle.

Other work has shown the potential of multiple instruction windows to ease implementation by decoupling reference and computation streams [JT97]. Though the effective window size of the two separate windows proposed in that work can exceed the sum of their individual sizes, the two instruction streams are very tightly coupled (by a single program counter) compared to the separate instruction streams discovered here.

This parallelism at a distance supports much previous work in multi-threaded execution [TEL95], and provides some insight into how multiple threads may be found from a single "sequential" program.

3 Methodology

In order to examine the available parallelism, we constructed an execution driven simulator based on the the simplescalar simulation environment [BA97]. A graphical image is generated based on information produced by the simulator. This image provides insight into the available parallelism in the benchmarks.

3.1 Simulation

The simulator is derived from simplescalar's sim-safe.c (a simple functional simulator) and modified to track the cycle in which each instruction could be issued according to data, control and resource dependencies. In this section, the construction of the simulator will be discussed and the benchmark applications will be presented, with the goal that readers can easily duplicate our results.

The primary data structures used to track data dependencies in the simulator include:

1. REG[64] – a register file consisting of 32 integer and 32 floating point registers. This structure holds the current register data during functional simulation.
2. MEM[4GB] – the memory system containing program data allocated in memory. Logically, this is treated as a 4GB array of bytes.
3. REG.DEF.CYCLE[64] – a cycle count associated with any register definition (write) indicating when that data could first be available during execution.
4. MEM.DEF.CYCLE[4GB] – a cycle count associated with any memory definition (store) indicating when that data could first be available during execution.
5. REG.USE.CYCLE[64] – a cycle count associated with any register use (read) indicating when that register was last read.
6. MEM.USE.CYCLE[4GB] – a cycle count associated with any memory use (load) indicating when that register was last read.

The actual execution of instructions within the simulator is performed in original program order. As each instruction is processed it is labeled with its position in the program order (INST.NUM) and the earliest cycle in which it can be executed is calculated (ISSUE.CYCLE). In the initial simulation, all data dependencies are calculated as follows:

- `RAW = max (REG.DEF.CYCLE(in1),`
  `REG.DEF.CYCLE(in2),`
  `REG.DEF.CYCLE(in3))`
- `WAR = max (REG.USE.CYCLE(out1),`
  `REG.USE.CYCLE(out2))`
- `WAW = max (REG.DEF.CYCLE(out1),`
  `REG.DEF.CYCLE(out2))`
- `if (load instruction)`
  `MRAW = MEM.DEF.CYCLE[addr]`
- `if (store instruction)`
  `MWAR = MEM.USE.CYCLE[addr]`
- `if (store instruction)`
  `MWAW = MEM.DEF.CYCLE[addr]`

where *in1*,*in2* and *in3* are source register operand specifiers, *out1* and *out2* are destination operand specifiers and *addr* is the effective address calculated for memory operations[1]. Once input dependencies have been calculated, the earliest issue cycle can be calculated; when no renaming is performed (and resource and control dependencies are ignored) the earliest issue cycle is calculated as:

$$\text{ISSUE.CYCLE} = \max(\text{RAW}, \text{WAR}, \text{WAW}, \text{MRAW}, \text{MWAR}, \text{MWAW}) \quad (1)$$

The simulation considers system call instructions to be serialization points but since the simulation is only of user-level instructions, system call modifications to machine state cannot be directly modeled. Instead, the simulator employs an additional implicit input dependency on each instruction which is the cycle number of the last system call. In effect, this assumes that all machine state is modified at the system call and so serializes execution.

After an instruction has been simulated, the data dependence structures are modified to accommodate the new register and memory references:

- `REG.DEF.CYCLE[out1] = ISSUE.CYCLE + 1`
- `REG.DEF.CYCLE[out2] = ISSUE.CYCLE + 1`
- `REG.USE.CYCLE[in1]  = ISSUE.CYCLE`
- `REG.USE.CYCLE[in2]  = ISSUE.CYCLE`
- `REG.USE.CYCLE[in3]  = ISSUE.CYCLE`
- `if (store instruction)`
  `MEM.DEF.CYCLE[addr] = ISSUE.CYCLE + 1`
- `if (load instruction)`
  `MEM.USE.CYCLE[addr] = ISSUE.CYCLE`

Table 1 shows the initial IPC measurements for the SPEC '95 benchmark suite. Simulations were terminated at completion or after 1 billion instructions for some of the long running floating point applications. Column 1 identifies the application, column 2 specifies the input file(s) used, and columns 3 and 4 show the total number of instructions and memory references executed during simulation. Column 5 of Table 2 shows

1. The simplescalar ISA includes three source operands (for future expansion) and some instructions may modify two destination registers (e.g. load with post-increment).

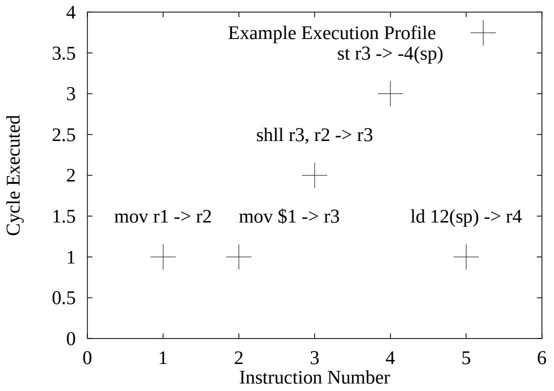


Figure 1: An example execution profile.

the IPC for the application when using equation (1) to calculate ISSUE.CYCLE. The IPC is obtained from:

$$\text{IPC} = \frac{\text{MaxINST.NUM}}{\text{MaxISSUE.CYCLE}} \qquad (2)$$

The results in Column 1 of Table 2 are consistent with earlier studies performing no register or memory renaming.

3.2 The Execution Profile

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 the execution time (ISSUE.CYCLE) on the y-axis versus the instruction's dynamic number (INST.NUM) on the x-axis. This representation of the data shows when each instruction in a program is free of dependencies and can be executed.

Figure 1 shows a very small example execution profile. Each point represents the execution of a particular instruction at a particular cycle. For example, the two mov instructions and the ld instruction at positions 1, 2, and 5 can execute in cycle 1 because they are independent. The shll instruction depends on the two previous instructions and therefore cannot execute until cycle 2. In general, 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 its own INST.NUM.

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 a page.

4 Effects of Renaming

In this section we examine the effect of renaming on ILP. We repeat previous register renaming studies for SPEC95 to provide a comparison and a validity check. We then consider memory renaming. Table 2 shows the summary results. In addition we include execution profiles for our running case studies, gcc and fpppp.

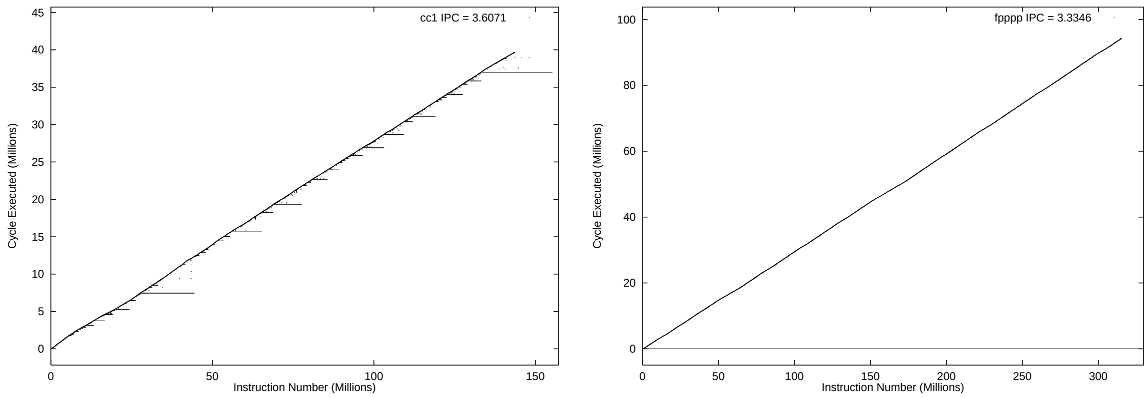
Table 1: Benchmark Descriptions (train inputs)

Benchmark	**Input**	**Insts (M)**	**Mem Refs (M)**
compress	small.in	95.2	33.8
gcc	jump.i	157	63.4
go	2stone9.in	151	41.7
ijpeg	vigo.ppm	1000	254
li	train.lsp	183	77.8
m88ksim	ctl.big	120	37.0
perl	scrabbl.pl	40.5	18.5
vortex	vortex.in	214	115
applu	applu.in	532	136
apsi	apsi.in	1000	315
fpppp	natoms.in	330	175
hydro2d	hydro2d.in	1000	260
mgrid	mgrid.in	1000	363
su2cor	su2cor.in	1000	308
swim	swim.in	797	235
tomcatv	tomcatv.in	1000	278
turb3d	turb3d.in	1000	216
wave5	wave5.in	1000	256

Figure 2 displays execution profiles for gcc and fpppp with no renaming. This corresponds to the gcc and fpppp IPC entries in Table 1 (bold). The IPC is shown in the upper right corners of the graphs. The horizontal axis plots each instruction in numerical order (from 1 to 157,242,271 in the case of gcc or 329,849,256 for fpppp).

In Figure 2, a horizontal slice would intersect, on average, 3.6 instructions for gcc or 3.3 for fpppp. In addition, the gcc profile contains shading to the right of the main cluster. This indicates that even without renaming, instructions that are far apart in program order (on the order of millions) can in fact be executed simultaneously. The fpppp graph exhibits a horizontal line at cycle zero extending through the entire execution. All of these instructions are executed early in the program run because they have no dependencies. These instructions include unconditional jumps, nops and instructions that write to registers the first time (so that output dependencies are not a factor).

In the next section we explore the effect of register renaming on the execution profiles of our example benchmarks.



(a) gcc (b) fpppp

Figure 2: Execution Profiles without Renaming. Note the y-axis scales.

4.1 Register Renaming

As noted earlier, register renaming is used to eliminate WAR and WAW dependencies. Accordingly, we calculate the issue cycle for an instruction as:

$$\begin{aligned} &\text{ISSUE.CYCLE} = \\ &\quad \max(\text{RAW}, 0, 0, \text{MRAW}, \text{MWAR}, \text{MWAW}) \end{aligned} \tag{3}$$

Register renaming eliminates WAR and WAW dependencies, so these are zeroed in the equation.

Figure 3 shows the execution profiles of gcc and fpppp when register renaming is enabled. Note that the y-axis scale is significantly reduced, indicating a substantial increase in performance. For gcc, the IPC improves by an order of magnitude, and fpppp shows an even greater improvement. There are many more horizontal lines in the graphs, indicating a great deal of parallelism is available. Most of this parallelism appears over great distances.

The third column of our summary table (Table 2 at the end of the paper) presents the IPC figures for our benchmarks when register renaming is applied. Register renaming results in a substantial increase in potential IPC.

Register renaming works because false dependencies often determine the execution time of an instruction. This is apparent in Figure 4, where a breakdown of the limiting dependencies is presented. For each type of dependency, the simulator tracks how many times that particular dependency type determined the execution time of the instruction, i.e. how many times it was the maximum value in equation (1). WAR and WAW dependencies through registers account for approximately 55%-60% of the limiting dependencies. Register renaming eliminates these dependencies.

Register renaming also uncovers another class of false dependencies: those through the memory system.

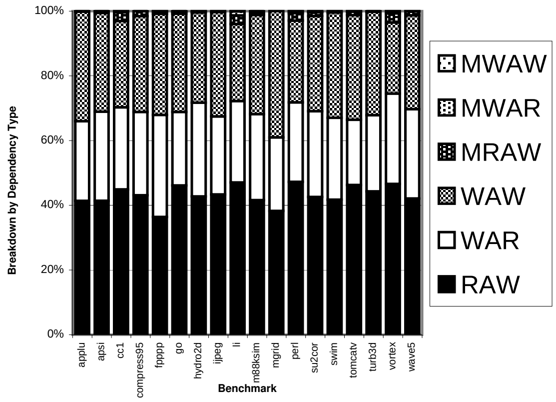


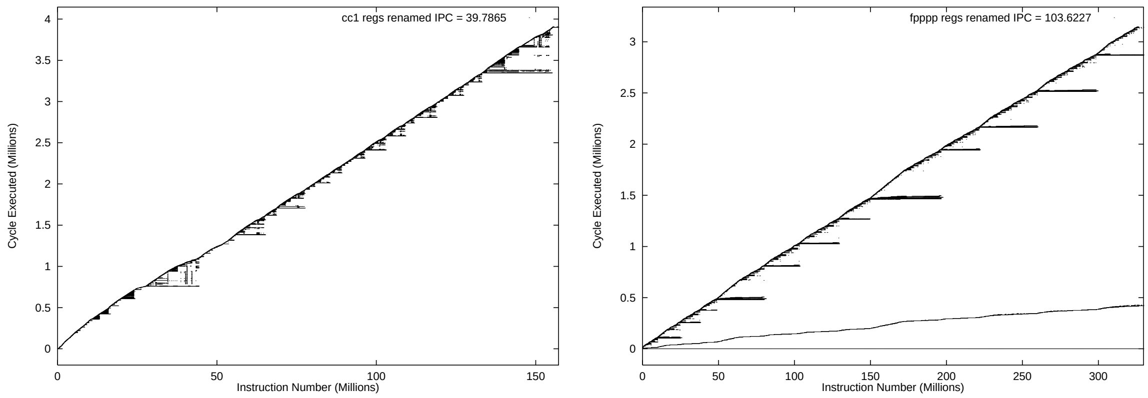
Figure 4: Dependency Breakdown by Type.

Although these dependencies seem small in Figure 4, they often have much larger penalties associated with them and their removal can significantly improve IPC. We discuss this in the following section.

4.2 Memory Renaming

While register renaming is effective for removing false dependencies in a limited range, it does not remove such dependencies when they occur through memory. Memory dependencies are much more difficult to detect and eliminate because of the resource needs of any detection mechanism and the aliasing problem of determining whether two computed addresses reference the same memory location.

For this experiment we eliminate MWAR and MWAW dependencies, in addition to register renam-



(a) gcc

(b) fpppp

Figure 3: Execution Profiles with Register Renaming. Notice that the y-axis scales are an order of magnitude smaller than in the corresponding graphs of Figure 2.

ing. Our issue cycle calculation now becomes:

$$\text{ISSUE.CYCLE} = \max(\text{RAW}, 0, 0, \text{MRAW}, 0, 0) \quad (4)$$

Only true dependencies are considered when calculating the issue cycle of an instruction.

Figure 5 presents the execution profiles of gcc and fpppp when register and memory renaming are enabled. The y-axis scale for gcc has not changed, indicating that gcc is not limited by false dependencies through memory. On the other hand, fpppp execution time has improved by almost another order of magnitude. In addition, many of the interior points have been collapsed to lower cycles, often to cycle zero.

The fourth column of Table 2 presents the IPC figures for our benchmarks when register and memory renaming are applied. The improvement in IPC is highly benchmark dependent. As shown by Figure 4, memory-carried dependencies are a small percentage of those dependencies that determine execution time. However, in those benchmarks where the penalty for memory-carried false dependencies is large, significant improvement in IPC can be observed when the false dependencies are removed. In particular, applu is quite limited by false memory dependencies. Removing this limit results in an order of magnitude increase in IPC. Other benchmarks, such as gcc and go, are not limited by false memory dependencies.

Memory renaming can be beneficial when areas of memory are re-used often. A common source of false dependencies through memory is the use of a runtime stack for function linkage. The same memory area is continually re-used as the stack pointer is moved and activation records are pushed and popped. This use of a stack is an artifact of the compilation model; it is one possible method used to abstract the concept of a function and private local variables.

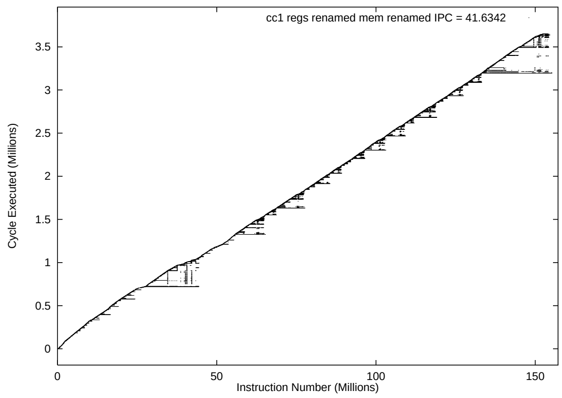
5 Compiler Interactions

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, this low-level representation is generated by a compiler from a more abstract high-level language. During translation, the compiler not only translates the original data dependencies of the application, but includes additional instructions supporting language abstractions (e.g. functions). These additional instructions create their own data dependencies which can, in themselves, limit the parallelism in aggressive machines.

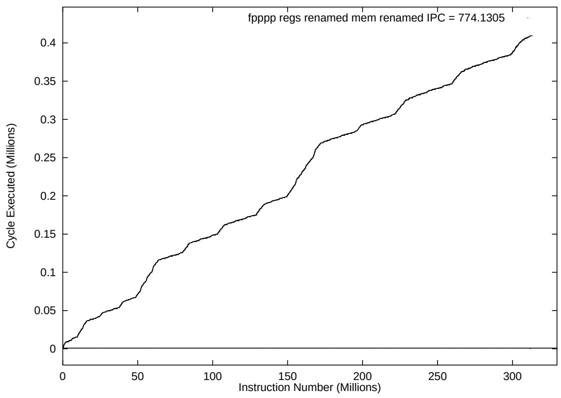
Chief among the compiler generated dependence chains is the stack pointer register used to allocation function activation records. The use of a stack pointer is very efficient in allocating the memory space for functions (local variables, parameters, etc.); it takes only a single instruction to allocate a function activation record of arbitrary size[2] and a second instruction to deallocate that space upon completion of the function call. While this is efficient in terms of space and instructions required to allocate space, aggressive, multi-issue architectures are less constrained by the number of instructions required to perform a task than by the lengths of the dependence chains connecting the instructions. 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 ex-

2. SP = SP - size_activation_record



(a) gcc



(b) fpppp

Figure 5: Execution Profiles with Register and Memory Renaming. Notice again the change in y-axis scales.

ploitable ILP. The first, false dependencies through reuse of stack space, is solved with memory renaming as explained in Section 4.1. 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.

Figure 6 plots the same execution profile shown in Figure 5 with only those instruction that update the stack pointer register displayed. Almost all of these instruction are of the form `(SP = SP - size_activation_record)`; however, in gcc the stack pointer is also modified by dynamically allocating a variable amount space on the stack with the `alloca()` function. Comparing the IPC values in Figures 5 and 6 shows that even if every real program dependency was eliminated, leaving only the compiler-induced dependencies, the execution time is almost unchanged.

In Figure 7, 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[3]. Table 2 shows that for many applications there is a significant increase in the number of instructions executed each cycle — nearly an order of magnitude (or more) for half of the applications. The parallelism uncovered by removing the stack pointer is far beyond that uncovered by perfect prediction and renaming. Our method of removing stack dependencies is overly optimistic (it does not account for any additional overhead in managing the heap, for instance, nor for recursion). Still, it is a limit which indicates that the compiler-induced stack dependency is significant.

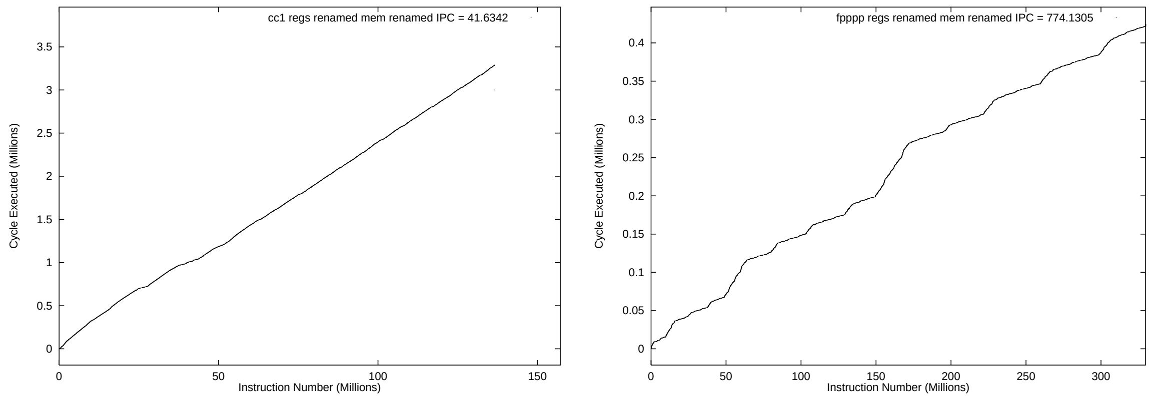
Table 2 summarizes the IPC values found in each of the infinite instruction window configurations. The final column shows the effect of limiting the instruction window (to 10,000) in the least constrained model (both register and memory renaming with all stack register dependencies removed). With a limited instruction window, the IPC of most applications is limited to a value approximately equal the register renaming only model. 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 instruction window to capture. This is the so-called "parallelism at a distance".

6 Methods for Extracting 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 magnitude better than can be achieved with current technology. In this section, we will discuss what capabilities some future processor must have in order to exploit the very distant parallelism which we have demonstrated.

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 analysis in sections 4 and

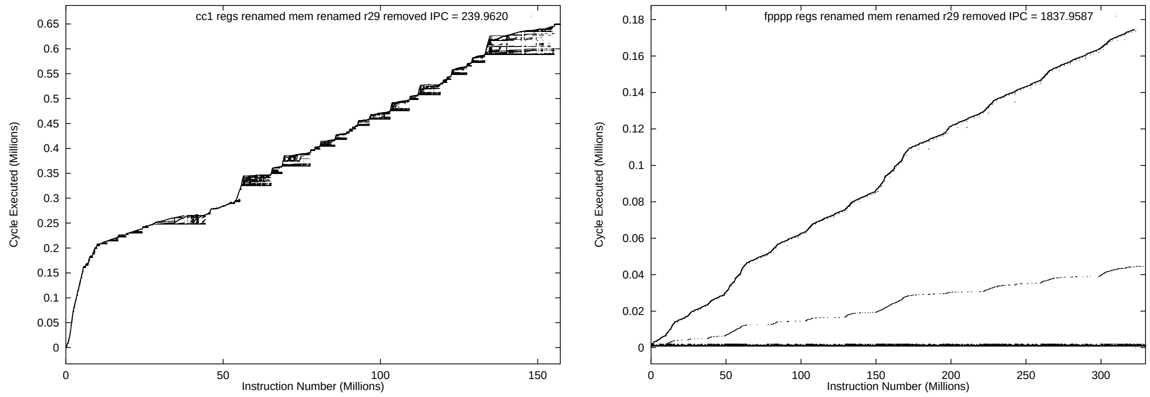
3. This is performed by maintaining REG.DEF.CYCLE[29] at zero.



(a) gcc

(b) fpppp

Figure 6: Stack Register Dependency Execution Profiles.



(a) gcc

(b) fpppp

Figure 7: Execution Profiles with Stack Pointer Removed. The y-axis scales have dropped by another order of magnitude.

5 show that threads do exist, but offers little help in identifying them statically. Furthermore, 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.

For some functions it may be relatively easy to generate threads — the C `printf` routine, for instance, is a long function with no significant side effects. It is very likely that the code immediately following the call to `printf` can be executed independently of the `printf` call (unless the return value is used), but no single-window machine would be able to do so since the `printf` call may require execution of tens of thousands of instructions. The widespread use of library routines makes them natural candidates for this type of thread creation since they are quite easy to analyze.

Executing threads at this granularity necessitates a very lightweight thread creation and dispatch mechanism. One aspect of this mechanism is to allocate space for the activation record of a thread or function. Stack allocation of activation records is a common technique to support local frame allocation, but other options are possible. 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 [LY97]. 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.

Table 2: 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
cc1	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	0
wave5	3.25	29.28	35.71	35.71	0

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 executed immediately (assuming no other dependencies). In effect, the single runtime stack is partitioned into multiple stacks used by independent procedure invocations. This can also be generalized to a multiple stack implementation if the machine ISA has appropriate support.

7 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.

This work supports previous studies of register renaming as a solution for false register dependencies. We extended the limit study to include memory renaming. In previous studies it was thought that memory renaming was infeasible. However, recent work has proposed a feasible, though limited, version of memory renaming. By extending our study to include memory renaming we exposed the constraints placed on parallelism by the linear nature of the stack allocation of activation records. Therefore we removed the decrement/increment dependency chain responsible for frame allocation. We discussed several possible mechanisms to do this. These techniques together expose much greater parallelism than has been seen in previous studies.

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 even a highly aggressive implementation to discover them. Therefore a processor that executes from multiple instruction streams will be required to uncover this parallelism. While it is clearly a difficult task to identify this distant parallelism, we feel that a hybrid approach exploiting some local parallelism along with a portion of the more distant parallelism is the only chance to dramatically increase the overall parallel execution within a single application.

Furthermore, compiler support will be required to eliminate the false dependencies introduced by the stack model used in the calling conventions of languages such as C and C++. Compiling for threads and heap allocation of activiation records seem to be the most obvious starting points.

Future work that eliminates the stack and other compiler-induced dependencies may open up better ways of performing inter-procedural analysis for parallelism. In addition, architectural changes that allow relaxation of compiler guarantees will allow a compiler without complete knowledge of program behavior to make "unsafe" assumptions that are valid through most or all of the program execution.

8 Acknowledgments

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.

[BA97] Douglas C. Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Tech. Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[BYP+91] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *Proc. ISCA-18*, volume 19, pages 276–286, June 1991.

[Gwe97] Linley Gwennap. Design concepts for merced. *Microprocessor Report*, 11(3):9–11, March 1997.

[JT97] G. P. Jones and N. P. Topham. A comparison of data prefetching on an access decoupled and superscalar machine. In *Proc. Micro-30*, pages 65–70, December 1997.

[JW89] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proc. ASPLOS-3*, volume 24, pages 272–282, May 1989.

[KBC+74] D. Kuck, P. Budnik, S. C. Chen, E. Davis, Jr., J. Han, P. Kraska, D. Lawrie, Y. Muraoka, R. Strebendt, and R. Towle. Measurements of parallelism in ordinary FORTRAN programs. *Computer*, 7(1):37–46, January 1974.

[Kel75] Robert M. Keller. Look-ahead processors. *ACM Computing Surveys*, 7(4):177–195, December 1975.

[LS84] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, January 1984.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1997.

[NF84] A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. In *IEEE Trans. Computers*, volume C-33, pages 968–976. 1984.

[RF72] E. M. Riseman and C. C. Foster. The inhibition of potential parallelism by conditional jumps. In *IEEE Trans. Computers*, volume C-21, pages 1405–1411. 1972.

[SJH89] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple instruction issue. In *Proc. ASPLOS-3*, number 5, pages 290–302, May 1989.

[SV97] James E. Smith and Sriram Vajapeyam. Trace processors: Moving to fourth-generation microarchitectures. *Computer*, 30(9):68–74, September 1997.

[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.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. ISCA-22*, pages 392–403, June 1995.

[TF70] G. S. Tjaden and M. J. Flynn. Detection and parallel execution of independent instructions. *Journal of the ACM*, 19(10):889–895, October 1970.

[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.

[YP92] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In David Abramson and Jean-Luc Gaudiot, editors, *Proc. ISCA-19*, pages 124–135, May 1992.