Finding Parallelism for Future EPIC Machines

Matthew Iyer[†], Chinmay Ashok[†], Joshua Stone[†*], Neil Vachharajani[‡], Daniel A. Connors[†], and Manish Vachharajani[†]

[†]Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO 80309
{Matthew.Iyer, Chinmay.Ashok, Joshua.Stone, dconnors, manishv}@colorado.edu

[‡]Department of Computer Science
Princeton University
Princeton, NJ 08544
nvachhar@cs.princeton.edu

ABSTRACT

Parallelism has been the primary architectural mechanism to increase computer system performance. To continue pushing the performance envelope, identifying new sources of parallelism for future architectures is critical. Current hardware exploits local instruction level parallelism (ILP) as hardware resources and information communicated by the instruction-set architecture (ISA) permit. From this perspective, the Explicitly Parallel Instruction Computing (EPIC) ISA is an interesting model for future machines as its primary design allows software to expose analysis information to the underlying processor for it to exploit parallelism. In this effort, EPIC processors have been more or less successful, however the question of how to identify (or create) additional parallelism remains. This paper analyzes the potential of future relationships of compilers, ISAs, and hardware resources to collectively exploit new levels of parallelism. By experimentally studying the ILP of applications under ideal execution conditions (e.g., perfect memory disambiguation, infinite instruction-issue window, and infinite machine resources), the impact of aggressive compiler optimization and the underlying processor ISA on parallelism can be explored. Experimental comparisons involving an Itanium-based EPIC model and an Intel x86-based CISC (Complex Instruction Set Computing) model indicate that the compiler and certain ISA details directly affect local and distant instruction-level parallelism. The experimental results also suggest promising research directions for extracting the distant ILP.

1. INTRODUCTION

Current processors exploit instruction-level parallelism (ILP) in single threads by using aggressive control speculation, data speculation, large instruction issue windows, and high-bandwidth, low-latency instruction caches. Unfortunately, these now classic design features are not scaling well given trends in fabrication technology; feasible extensions of these methods are showing vanishingly small returns. For instance, despite advanced branch predictors, increasingly large misprediction penalties strongly limit speedup for control intensive programs [1]. Likewise, the instruction window currently consumes a large percentage of total processor core power [2], and is very difficult to enlarge [3]. Larger instruction windows require larger multi-ported register files and larger reorder buffers to realize their potential, and these also are difficult to design [4]. Finally, the added complexity of new microarchitectural concepts has led to ever larger design teams to manage design complexity and correctness. The consequence of this poor scalability and diminishing returns in the recent past is that caches now dominate the area of typical high-end processor chips [5].

Currently, the challenges of meeting continual demands for performance improvements have spurred designers to move to chip multithreaded (CMT) [6] and chip-multiprocessor (CMP) [7] designs. In these designs, replication of processor cores and hardware contexts reduces design effort per transistor and provides a way of delivering more computational power without the scalability problems described above. While these systems provide higher overall throughput, single-thread application performance sees no direct improvement, and sometimes even experiences slowdown due to the overhead in multiprocessor operating systems. In response, there is an effort to explore the use of multi-threading multi-processing to exploit parallelism within a single thread [8, 9].

Two broad classes of evaluating the level of parallelism available in modern single-threaded applications exist: near-ILP (exploitable by kilo-instruction window processors [10]) and distant ILP (due to instructions more well over 2K instructions ahead in execution trace [11]). Both classes of parallelism are evaluated by considering ideal execution conditions (unconstrained microarchitectural resources). However, most compiler and architecture studies have focused on near ILP as it relates more closely to scaling existing processor models [12, 13]. On the other hand, there are more avenues to exploit distant ILP and these may unlock substantially more parallelism. As exploiting distant ILP will likely require compiler and/or run-time system support, it is critical to examine the influence that compiler technology has on distant ILP. Similarly, the information communicated by different instruction-set architecture (ISA) models also directly influences the ability of hardware-based techniques to exploit distant ILP. This paper presents an integrated study of distant ILP by simultaneously examining multiple ISAs and the impact of compiler optimization.

Results show that for both EPIC (Explicitly Parallel Instruction Computing) and CISC (Complex Instruction Set Computing) architectures, a large amount of useful ILP exists beyond the range of current instruction windows, much of it beyond the range of kilo-instruction processors. At the same time, results also show, surprisingly, that local transformations implemented in modern aggressively optimizing compilers have a significant influence on the amount of distant ILP available. In many cases the compiler sacrifices distant ILP in exchange for better exposure of near ILP on real hardware. Additional results from this study suggest that the

[*]Now at Intel Corp., Santa Clara, CA

character of distant ILP is quite similar across different architectures (in this case Itanium, PowerPC, and x86), though the absolute IPC and other details vary greatly between the platforms. The study also determined that while the ISA does not have a dominant role in distant ILP patterns, specific, seemingly minor, platform details do play a major role. During the course of this investigation, several key properties of distant parallelism such as dependence criticality and parallel dependence chains were identified. Overall, these results motivate potential new design techniques to exploit distant ILP on multi-core platforms.

The remainder of this paper is organized as follows. Section 2 describes the experimental infrastructure and setup for this study, including Adamantium, an ideal trace scheduler. Section 3 details the results of the limit study and presents our analysis. Section 4 briefly describes previous ILP limit studies and highlights how the this study differs. Finally, Section 5 concludes by discussing the implications of our results and suggesting future directions.

2. LIMIT STUDY METHODOLOGY

Studies have previously shown the impact of hardware resources on the the potential level of parallel execution. This study is focuses on the upper bound of available parallelism with the primary consideration of inherent program information, the effect of *instruction-set architecture* (ISA) and the compiler transformed by the compiler. In particular, the study:

- Characterizes the amount and type of natural parallelism in an application.
- Evaluates the role of a single-thread compiler in uncovering different types of parallelism.
- Examines the role of instruction-set architecture (ISA) in permitting parallelism.

In conducting the limit study of available ILP, a range of conditions, a variety of ISAs, and a variety of compilers are examined. The following sections describe the techniques used to perform the limit study and the space of applications, compilers, and the targeted ISAs. In addition, the mechanisms used to address the challenges in implementing the infrastructure to perform the study are discussed.

2.1 ILP Limit Studies

The process of assessing limits to parallelism generally involves several steps: collecting application execution traces from an architecture, determining true (flow) dependences of instructions within the trace, and scheduling the instructions based on unconstrained hardware resources. An example of an execution trace and its respective ILP-limit schedule is shown in Figure 1a and Figure 1b.

To identify the limit of available ILP, the scheduler should assume single cycle latency, ignore all false dependences including false memory dependences (i.e., those that can be avoided via renaming), assume perfect branch prediction allowing instructions to be moved across branches, and assume perfect memory disambiguation. Notice, in Figure 1c, that because the scheduler ignores all false dependences, instruction number 10 need not wait for dynamic instruction number 6 to complete since the value of `r1` defined by 10 is a new value for `r1` indicating that the dependence on r1 between 6 and 10 is false. Furthermore, note that although dynamic instruction 11 is a branch, instructions 12, 13, and 14 need not wait for it

```
1         li  r1, &fibarr
2         mov r2, 8      // cnt=8
3         mov r3, 1      // prev1=1
4         mov r4, 1      // prev2=1
5   loop: add r5,r3,r4   // tmp=prev1+prev2
6         st 0(r1),r5    // fibarr[i]=tmp
7         mov r3,r4      // prev1=prev2
8         mov r4,r5      // prev2=tmp
9         addi r2,r2,-1  // cnt--;
10        addi r1,r1,4   // i++;
11        bnz r2, loop
```

(a) Fibonacci Program to be Analyzed

Dynamic Instruction Number	Instruction	Memory Address Accessed
1	`li r1, &fibarr`	
2	`mov r2, 8`	
3	`mov r4, 1`	
4	`mov r3, 1`	
5	`add r5,r3,r4`	
6	`st 0(r1),r5`	0xbee0
7	`mov r3,r4`	
8	`mov r4,r5`	
9	`addi r2,r2,-1`	
10	`addi r1,r1,4`	
11	`bnz r2, loop`	
12	`add r5,r3,r4`	
13	`st 0(r1),r5`	0xbee4
14	`mov r3,r4`	
	...	

(b) Execution trace

Cycle	Dynamic Instruction Numbers
1	1 2 3 4
2	5 7 9 10
3	6 8 11
4	12 14
5	13

(c) Ideal trace schedule

Figure 1: Stages of an ILP limit experiment

Platform	gcc	icc	glibc
x86	3.3.2	8.1	2.2.5
Itanium	3.2.3	8.1	2.3.2
PowerPC	3.4.1	n/a	2.3.3

Table 1: Compiler versions used to build SPEC benchmarks

due to perfect branch prediction. They are scheduled after instruction 11 due to their dependence on dynamic instructions numbered 8, 12, and 8, respectively.

In the scheduling infrastructure, perfect branch prediction is easy to achieve, the scheduler simply ignores all control dependences. Ignoring false register dependences is also straight-forward, the scheduler simply renames architected registers. Typically, this renaming is done by keeping a map of architected locations to a set of virtual "physical" names. The location of the instruction that produces a value in the execution trace is guaranteed to be unique and usually suffices. Perfect memory disambiguation is more difficult. In Figure 1, perfect memory disambiguation is achieved by recording in the execution trace the address used by every memory instruction. The scheduler then treats these memory locations just like architected registers, renaming them by creating a new label for each write to the location. Of course, managing a map that potentially contains every virtual address can be tricky. Furthermore, including the memory address with every execution trace can increase the size of the trace prohibitively. The mechanisms used to address these issues are discussed in Section 2.3.

2.2 Study Configuration Space

For this limit-study, the integer applications in the SPEC benchmark suite (i.e., SPEC CPU INT 2000 benchmarks) were examined. These applications were selected because they are considered particularly difficult to parallelize, especially when attempting to extract distant-ILP. Floating point applications have challenges but efforts at automatic parallelization of these codes have had success when compared to integer codes in the SPEC suite [11].

A subset of the SPEC CPU INT 2000 benchmarks were compiled for Linux running on Intel's x86 instruction set architecture (ISA), IBM's PowerPC ISA, and Intel's Itanium ISA. Benchmarks were compiled using both the GNU Compiler Collection (gcc), and when available, Intel's optimizing compiler (icc). Table 1 shows the compiler versions used for various platforms. All benchmarks were compiled with the `-O3` flag. Traces were generated for the train input set using Intel's Pin [14] tool for the x86 and Itanium traces. The PowerPC traces were generated using a modified version of the Liberty Simulation Environment's [15] PowerPC emulator [16].

All the resulting benchmarks were scheduled using the Adamantium trace scheduler developed for this study. A variety of scheduling window sizes were evaluated to examine the effect of window size on parallelism and to allow a comparison to prior limit studies that used finite windows. The window sizes used were 64 dynamic instructions, 128 dynamic instructions, 256 dynamic instructions, 2048 instructions, 16384 instructions, 102400 instructions, and an infinite window. All generated schedules at all window sizes ignore no-op instructions and discount instructions whose qualifying predicate is false (for Itanium).

The 3-tuple, $(\mathrm{arch}, \mathrm{compiler}, \mathrm{window})$ is used in the remainder of this paper to refer to the ILP results for a particular benchmark or set of benchmarks. Table 2 shows the values for each field of the tuple.

arch	x86	ia64 (Itanium)	ppc (PowerPC)
compiler	gcc	icc	
window	64	128	256
	2k (2048)	16k (16384)	100k (102400)
	inf (infinite)		

Table 2: Values for configuration tuple fields

2.3 Challenges

While the overall concept of the ideal limit study is simple, there are a number of practical challenges that arise in the implementation. Of these, there are three main challenges: limiting trace size on disk, managing renaming for memory locations, and generating meaningful output. The remainder of this section discusses how these issues are addressed.

2.3.1 Trace Size

The SPEC CPU INT 2000 benchmarks often execute in excess of 1 billion instructions. Assuming 33% of integer application instructions are memory operations with a single memory address, the trace file, naively stored would be at least 2GB on Itanium (8 bytes per address). This large size still ignores the space to store the actual instructions executed and their register operands. To keep the trace file size manageable, two separate mechanisms were deployed. First, the trace file is partitioned into two parts, a static trace file (called the program trace file) and the dynamic trace file (called simply the trace file). The static trace file stores a map of instruction address to decoded instruction information including the opcode, architected register operands, and some category information to aid in later scheduling (e.g., whether the instruction is a nop). In this way, the dynamic trace need only store the qualifying predicate (for Itanium), an instruction pointer with which the scheduler can index the program file, and any memory operand addresses. To manage the excessive size of the memory operand addresses, the TCgen [17] tool was used to transform the traces into highly compressible streams. The streams are then compressed using bzip2. Trace compression ratios were observed similar to those reported by the TCgen authors.

This trace-size reduction strategy poses a special challenge for the Itanium architecture. The Itanium supports compiler controlled renaming, and thus, the architected register numbers are not sufficient to properly track dependences. To avoid writing renamed register numbers to the trace (greatly expanding its size), the full REN stage of the Itanium pipeline was implemented in the trace scheduler. This allows us to translate the architected registers in the program trace file to physical register numbers consistent with the renaming requested by the code sequence during trace scheduling. The values of the CFM register and PFS register must be tracked throughout the trace in order to support the REN stage [1]. To support this the Itanium trace generator writes a side-trace file that stores every IP (instruction pointer) that modifies the CFM or PFS, and the corresponding register value. This side-trace is compressed using bzip2.

2.3.2 Managing Memory Renaming

[1]The CFM and PFS registers track how the registers have been rotated for software pipelining, as well as which register stack frame is currently in use. See the Itanium architecture reference manuals for more detail.

In order to perform register renaming, the Adamantium trace scheduler maintains a map of every architected register to the ideal cycle number in which the last producer of that register was scheduled. This map is simply a fixed size array of 64-bit cycle numbers indexed by architected register number. This strategy allows for fast scheduling (recall that Adamantium is scheduling well over a billion instructions for many benchmarks), but does not work for memory since a 32-bit machine would require over 32 GB of RAM. A 64-bit address space could not be mapped on any machine. To remedy this, but still allow fast lookup of dependences, Adamantium uses an inverted page table backed by a multi-level page table with large page sizes (between 32k and 128k 64-bit time entries per page). Nodes in the page table are only allocated on first reference limiting the total size of the structure to just as many pages as the application being scheduled needs. The hash table in the inverted page table has 1024 entries and two hash functions: a primary hash and a secondary hash for use on collision. With this infrastructure 98% of the memory dependence lookups hit in the hash table, leaving only 2% to page table walks.

2.3.3 Generating Meaningful Output

The final challenge for trace scheduling is identifying a useful set of output for analysis. Outputting the full schedule is impractical since it will be as large as the original trace. Compressing this schedule as described earlier prevents reasonable analysis because random access is disallowed. In general, the question of how to best store and analyze the schedule remains unaddressed. In this paper, analysis was limited to schedule representations generated incrementally as scheduling occurred. This data includes:

- A histogram that shows the idealized IPC of the application over ideal scheduling cycles (i.e., time).
- A plot of the time an instruction was scheduled versus dynamic instruction number (DIN) (i.e., the instruction's location in the serial execution trace) (DIN vs. ready-time plot).
- A plot of the static instruction number (SIN), the instruction address, of each instruction vs. the time it was scheduled in the ideal scheduler (SIN vs. ready-time plot).

Although simple, these three representations provide a wealth of information about the available ILP in a program and allow interesting comparisons between compilers and architectures as will be discussed in Section 3. As will be seen, they also guide us to interesting phenomena to investigate regarding ILP.

3. RESULTS AND ANALYSIS

This section presents a representative subset of the interesting information revealed by the limit study described in Section 2. The section begins by giving an overview of the IPC available on both the x86 and Itanium architectures. It continues by characterizing various phenomena regarding the IPC, especially the distant-ILP. Next, results for the two compilers (icc and gcc) are compared and interesting features identified. Finally, with an understanding of the role of the compiler, the section concludes with an examination of the ISA's role in the far-ILP available in a program binary.

3.1 Overview of IPC

Figure 3 shows the average IPC values for the x86 and the Itanium architectures given an infinite instruction window for the SPEC

```
1  .function foo
2  foo: subi $sp,$sp,242 //Grow stack
3       ...
4       stw  r1,28($sp)  // Spill code
5       ...
6       ld   r1,28($sp)  // Fill code
7       addi $sp,$sp,242 // Shrink stack
8       jr   $ra         // Return
```

(a) Arithmetic SP Update

```
1  .function foo
2  foo: stw  0($bp),$sp
3       subi $sp,$sp,242 //Grow stack
4       ...
5       stw  r1,28($sp)  // Spill code
6       ...
7       ld   r1,28($sp)  // Fill code
8       ld   $sp,0($bp)  // Shrink stack
9       jr   $ra         // Return
```

(b) Save-restore SP Update

Figure 2: Stack pointer manipulation code.

CINT 2000 benchmark suite[2]. This graph shows that Itanium comparitively performs far better across the entire benchmark suite, more than intuition would suggest. This finding suggests that there is a naive set of unnecessary true dependences on the x86 architecture that causes this drop in performance. As discussed in the literature we found this to be a dependence based on arithmetic update of the stack pointer [18, 19].

The x86 stack pointer is updated arithmetically as shown in the RISC-like assembly pseudo-code in Figure 2a. This pair of arithmetic updates to grow and shrink the stack occur on each call and return on x86 architecture. As a result all instructions in the function body that use the stack pointer (and their dependents), such as register spill and fill code, are unable to execute until after the corresponding instruction to grow the stack pointer. However, each grow instructions is pegged behind prior shrink instructions. This dramatically limits ILP. However, consider the save-restore based stack pointer update code in Figure 2b. In this figure, there is no such dependence between the grow and prior shrink instruction because renaming is able to disambiguate the different stack pointer definitions. Based on this observation, for the x86 architecture, we ignore all dependences based on the stack pointer. This marginally inflates the IPC of the application because the SP grow instructions are not serialized, but this effect is minor on the whole. There is no need to ignore the SP on Itanium since the Itanium architecture has a stacked general purpose register file with hardware backed fill and spill when the stack overflows the physical register file. This is discussed in more detail in Section 3.5.

Figure 4 shows the same data as Figure 3 while ignoring stack pointer dependences on x86. This demonstrates that if the stack pointer dependences are ignored, the x86 architecture does better in comparison with the Itanium architecture on an average. Figure 5 demonstrates that, on the x86 architecture, far fewer dynamic instructions are executed than on the Itanium architecture. This observation suggests that there is a greater amount of parallelism

[2]Benchmarks 256.bzip2, 186.crafty and 252.eon have been omitted. On Itanium, trace generation for 256.bzip2 has been excessively slow and traces were unavailable at the time of this writing. Technical difficulties were experienced for 252.eon and 186.crafty.

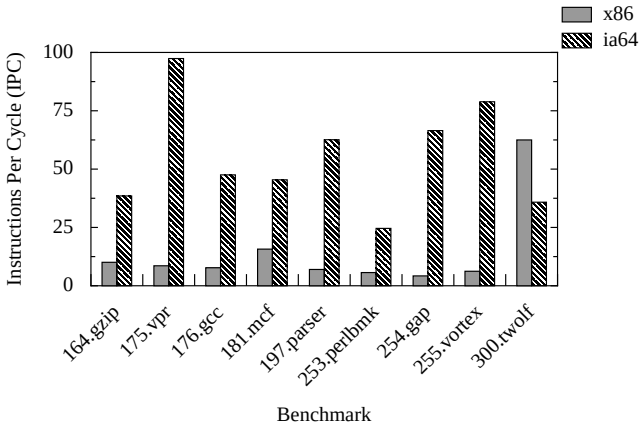


Figure 3: Average IPC for (x86,gcc,inf) and (ia64,gcc,inf)

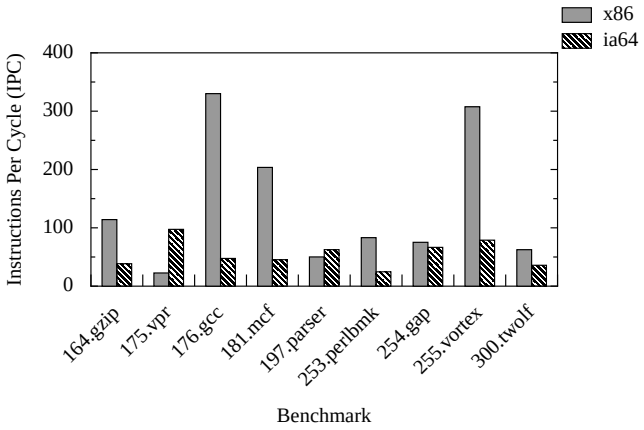


Figure 4: Average IPC ignoring SP on x86

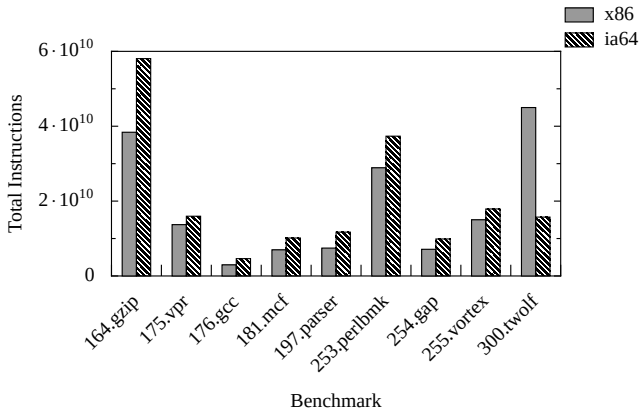


Figure 5: Number of operations by architecture

available for the x86 architecture over the entire execution, if the stack pointer dependence is ignored. This is an counter intuitive which should be explored in greater detail. Another possible cause for this effect is discussed in Section 3.5, but further investigation is still warranted. Notice that 300.twolf executes far more instructions on an x86 architecture as compared to the Itanium architecture. In addition, it is apparent that ignoring the stack pointer does not offer an advantage in the case of 300.twolf. This is because the number of calls made within the program are very few, minimizing the effect of the stack pointer dependence.

3.2 Characterizing ILP

Many prior limit studies limit the window size when considering "ideal" ILP. For example, the studies by Wall [13], limit the instruction window to two thousand instructions. Given this, and the recent work advocating kilo-instruction processors [10], terms near and distant-ILP are loosely defined. Near-ILP is parallelism found within approximately two thousand dynamic instructions, and distant ILP (i.e., far-ILP) is parallelism considerably farther than this. In this section two phenomena that characterize distant parallelism are presented: critical dependences and independent dependence chains.

3.2.1 DIN Plots

Before beginning a discussion of these effects, the visualization of ILP used throughout the remainder of this paper must be discussed. Consider Figure 6 for 254.gap (ia64,gcc,inf). The top graph of this figure is a plot whose horizontal axis represents time as cycles in which the scheduler schedules instruction. The vertical axis represents dynamic instruction number (DIN). Each instruction in a trace is plotted in the graph according to the time that it is scheduled by Adamantium and its dynamic instruction number. The lower graph in the figure shows the instantaneous IPC of the application over scheduling cycles. This pair of plots will be called a DIN plot or a DIN vs. ready-time plot.

These plots are useful for discussing far-ILP because they easily show interesting phenomena. For example, a line with positive slope in this plot corresponds to a chain of locally dependent instructions. We know this because the plot is for ideal ILP, instructions are scheduled as early as possible. The sloped line forms because the execution of the instructions earlier in the line satisfy the dependences for those in the very next cycle. The slope of this line gives a rough indication of the IPC of the dependence chain. Note *dependence chain* is used loosely in this context. There is not necessarily a chain of fully dependent instructions, instead there are groups of independent instructions that are nearby in the trace (giving rise to near-ILP) satisfying dependences for the next group of nearby independent instructions.

Far-ILP is represented by vertical stacks of points. We know this is far ILP since the vertical distance between the points is large (indicating very different DINs and thus distance in the serial execution trace), however the instructions are scheduled in the same cycle. With this basic understanding of DIN plots, we can now discuss critical dependences and independent satisfying dependences for the next group of independent instructions.

3.2.2 Critical Dependences

Our study indicates that an important characteristics of distant parallelism is a tendency for large numbers of instructions to be data dependent upon a single result. When resolved, these *critical dependences* unlock vast quantities of ILP. Notice how the DIN plot

for 254.gap (Figure 6) exhibits staircase behavior. At the front of the "stair" (e.g., at 7.5×10^7 cycles), there is a fair amount of ILP since the line has decent slope. However, at some point the parallelism is exhausted and a very serial chain of dependences remains (with little near ILP and almost no distant ILP). This can be ascertained by the "top-of-the-stair" effect (e.g., from cycle 8.75×10^7 to 1×10^8). The final results generated by this long dependence chain satisfy a large number of other dependences giving rise to the start of another "stair." The set of dependences that unlocks this new parallelism is what we call a critical dependence. In this study, almost all benchmarks exhibit this staircase behavior with an infinite window (a few show a single long dependence chain executing throughout the program).

A more extreme example of critical dependences is shown in Figure 7 shows critical dependences in the DIN plot for benchmark 181.mcf (ia64,gcc,inf). 181.mcf uses an iterative method to minimize network flow, where results of each successive iteration of the flow computation algorithm depends upon the previous one. In the figure, each iteration of the algorithm (seen as a black wedge) begins execution with a large amount of far-ILP (indicated by the height of the wedge). As the iteration progresses, newer instructions are not available for execution. In fact, the flat horizontal top of each wedge strongly indicates that there are key dependences that cannot be satisfied (and thus more distant instructions are unable to execute.) At the thin end of a "wedge" a key section of code is executed. Once its execution is complete, a new wedge begins. This indicates that at that point in the schedule the critical dependences were satisfied and more distant ILP is unlocked. Once again, this DIN plot supports the notion that critical dependences within each iteration of the flow computation algorithm prevent execution of future iterations, and therefore cause a bottleneck in distant ILP.

The prevalence of critical dependences indicates that, in order to extract far-ILP, the compiler must break critical dependences, or some form of value prediction must be used to unlock far-ILP for execution as separate threads on different cores in a multi-core system. This is less necessary in the case of 181.mcf (versus 254.gap) because of the large amount of far-ILP present within an iteration. However, the relatively few dependences between iterations of the algorithm may make the *more distant* ILP more amenable to threading techniques.

3.2.3 *Independent Dependence Chains*

The DIN plot in Figure 6 also exhibits multiple divergent lines during periods that execute with high IPC (e.g., around cycle number 5×10^7). Each line represents a group of instructions with near ILP that form a dependence chain and must be serialized. Multiple lines with differing slopes suggests independent dependence chains executing in parallel with distant ILP. This phenomenon of independent dependence chains was present across most of the SPEC CPU INT 2000 benchmarks. In fact, for the benchmark 175.vpr (x86,gcc,inf) (whose DIN plot is shown in Figure 8) these chains continue for the entire execution of the program. These dependence chains could be harnessed by spawning a separate thread to execute each dependence chain in parallel on a multiprocessor system. This allows the chip-multiprocessor system to dynamically adapt its behavior to the available program. Furthermore, since these are dependence chains with relatively little IPC, large speedups could be achieved even with relatively simple cores. Of course, identifying the critical dependences for these chains (and thus identifying the point at which to spawn threads) remains an open questions, as

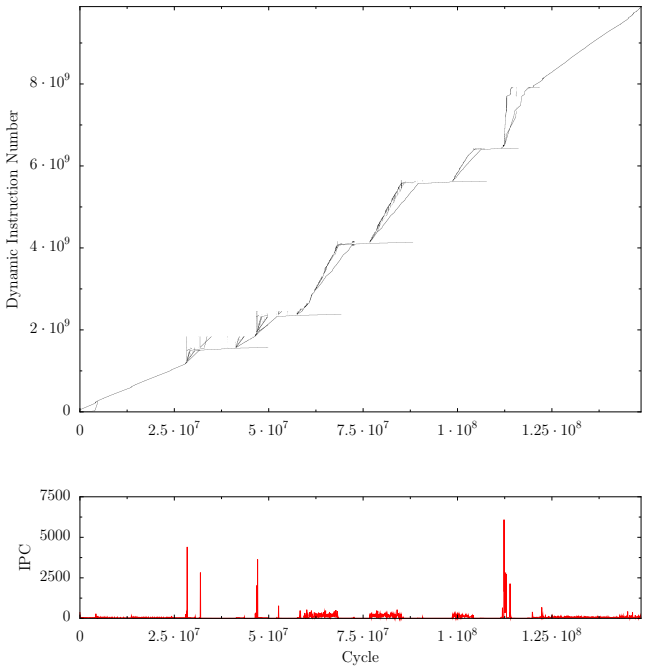


Figure 6: DIN versus ready-time for 254.gap (ia64,gcc,inf)

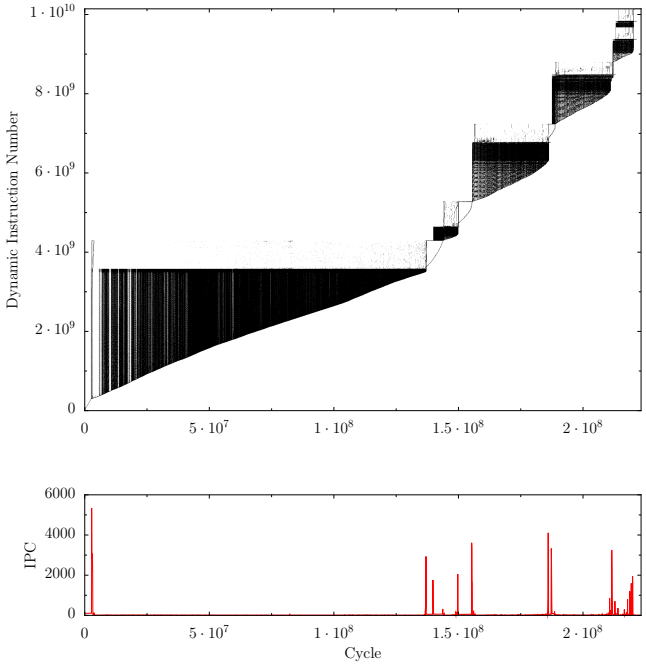


Figure 7: DIN versus ready-time for 181.mcf (ia64,gcc,inf)

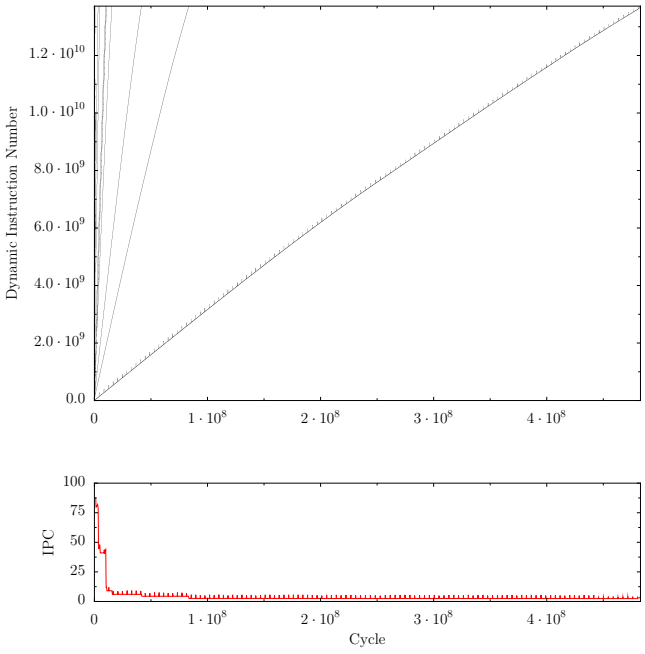


Figure 8: DIN versus ready-time for 175.vpr (x86,gcc,inf)

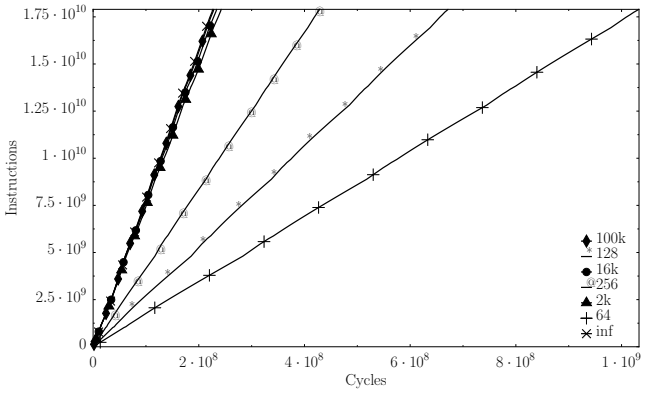


Figure 9: IPC for varying instruction-window sizes on 255.vortex (ia64,gcc)

does the mechanism to create the code for the threads in the first place.

3.3 Effect of Instruction Window Size on IPC

Section 3.2 shows the definitive existence of distant ILP in multiple forms. Conventional instruction windows lack scalability and are therefore unsuitable to exploit this distant ILP [2, 3]. Nevertheless, it is important to know how much nearby parallelism remains to be harnessed by incrementally larger instruction windows in current processors.

Figure 9 and Figure 10 show the effect of instruction window size on ILP for the benchmarks 255.vortex (ia64,gcc) and 197.parser (ia64,gcc). The graphs plot instructions-completed versus cycles for all the different instruction window sizes. In these plots, steep slopes represent high IPC, and flat areas represent low IPC. A single line is drawn for each window size and given a unique symbol. Notice the drastic differences in distant ILP for the two benchmarks. In 255.vortex, increasing the window size from 64 to 2k instructions exposes most of the available ILP; negligible gains are seen for 2k,16k,100k, and infinite windows. However, in 197.parser window sizes larger than 2k instructions continue to unveil parallelism, with the only apparent limitation being the total number of dynamic instructions. In addition to 255.vortex, benchmarks 253.perlbmk and 300.twolf demonstrate miniscule amounts of distant ILP beyond 2k instructions. The remaining benchmarks studied reveal substantial amounts of distant ILP beyond 2k instructions similar to 197.parser. (Recall that the interaction between predication and Adamantium in the Itanium architecture presents an additional limitation to ILP by converting control dependences to data dependences, as discussed in detail in Section 3.5).

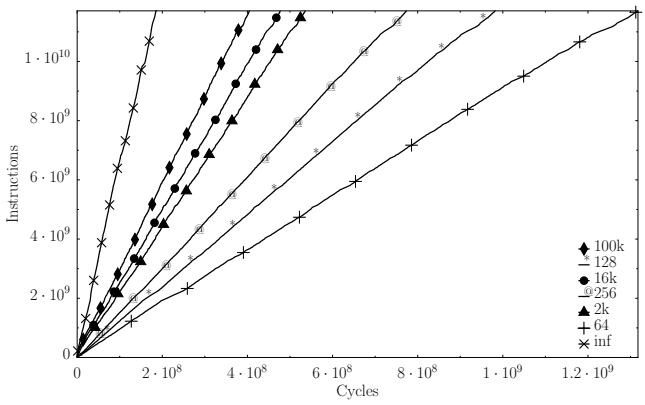


Figure 10: IPC for varying instruction-window sizes on 197.parser (ia64,gcc)

Our results indicate that for all benchmarks, a large amount of nearby parallelism remains untapped. However, this untapped parallelism will be difficult to extract in practice [13]. However, this lends merit to run-ahead and speculative techniques such as those of Mutlu et al. [20], Balasubramonian et al. [21], as well as proposals for larger instruction windows [22] and kilo-instruction processors [10]. Additionally, huge amounts of parallelism beyond 2k instructions exists, and exploring methods to distant parallelism remains important.

3.4 Effect of the Compiler on IPC

Compiler code transformations directly effect the inherent parallelism within the dynamic execution of a program. However, compiler transformations are primarily local, target increased parallelism, and attempt to reduce overhead within small regions of code. In fact, most architecture specific optimizations are implemented as peephole optimizations. This suggests that better optimizations applied by the compiler should improve near ILP, but distant ILP should remain relatively unaffected. This matches the intuition that distant ILP is a property of application characteristics such as data structures and algorithms which are intrinsicly affected by local optimizations. As will be seen, however, the results of this study contradict both these notions.

Figure 11 shows a comparison between Intel's C and C++ compiler (icc) and the GNU C compiler (gcc), for the x86 architecture, using different instruction window sizes, for the 181.mcf benchmark. The plots in grey represent icc runs, while the plots in black represent gcc runs. As expected, the more aggressive icc compiler completes more instructions per cycle than the gcc compiler for all window

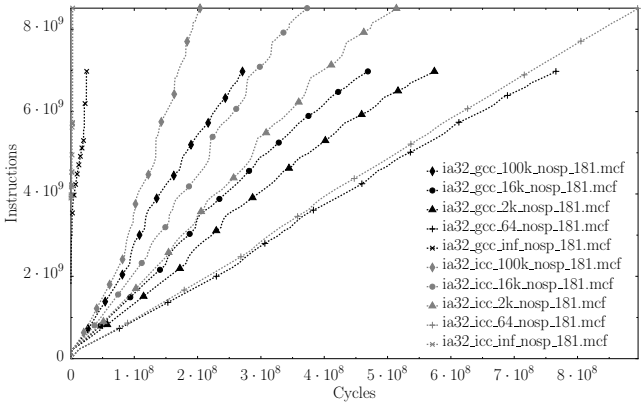


Figure 11: Compiler comparison for 181.mcf on x86 while ignoring the stack pointer

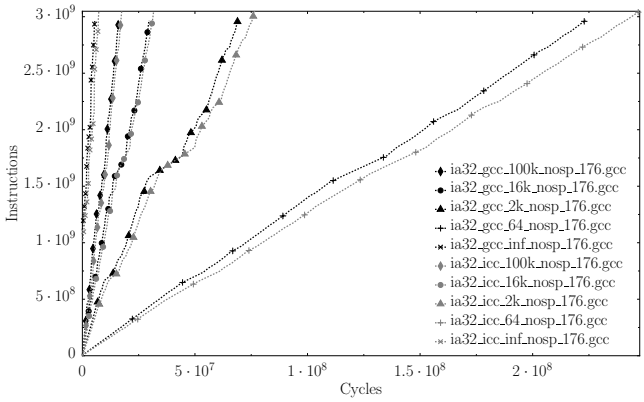


Figure 12: Compiler comparison for 176.gcc on x86 while ignoring the stack pointer

sizes. Note, however, that to do so icc executes 22% more instructions than the gcc compiler. Note, further, that the graph shows that there is a very large amount of distant parallelism available, which is evident from the large difference in completion time when using the infinite window. While 181.mcf is fairly well behaved, we shall see that a number of other benchmarks behave in a counter-intuitive fashion.

Figure 12 shows a similar compiler comparison for the 176.gcc benchmark. In contrast to 181.mcf, for 176.gcc the more aggressive optimizations performed by the icc compiler in order to exploit near ILP destroys the distant parallelism in the application. On the other hand, the less aggressive optimizations performed by the gcc compiler preserve the distant ILP. This causes the gcc schedules to out-perform the icc schedules in the ideal case. Figure 13 demonstrates a similar counterintuitive trend for the 254.gap benchmark. In fact more often than not (in 6 of 9 benchmarks), the attempt to utilize near ILP undermines the ability of a compiler to expose distant parallelism.

Note that this result also indicates that the compiler can have a significant influence on distant ILP, despite only local optimizaitons. This gives hope that compiler techniques (along with special hardware assists) will aid in extracting far ILP for next generation multicore systems. With that said, at this time, the reason for this effect

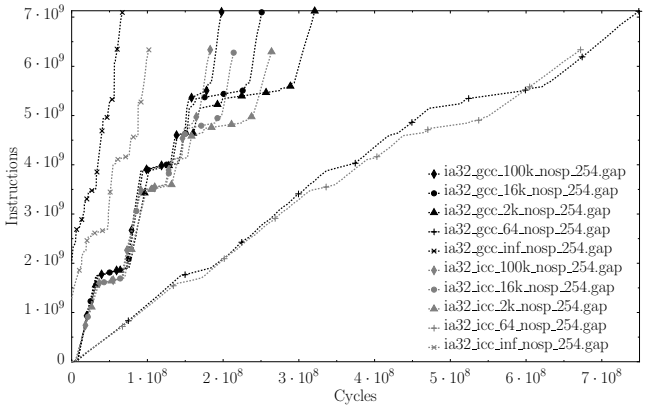


Figure 13: Compiler comparison for 254.gap on x86 while ignoring the stack pointer

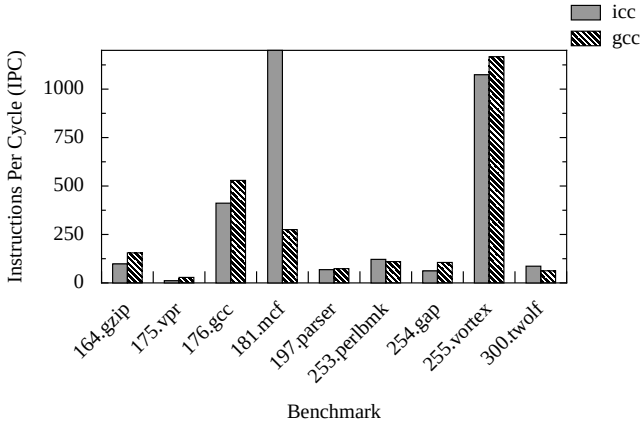


Figure 14: Compiler effect on average IPC for x86

is unclear, however, the effect is interesting and warrants further exploration. One possible explanation is that the local compiler transformations affect the critical dependences which have a dramatic effect on available ILP.

Figures 14 and 15 compare the 2 compilers on x86 and IA64 archi-

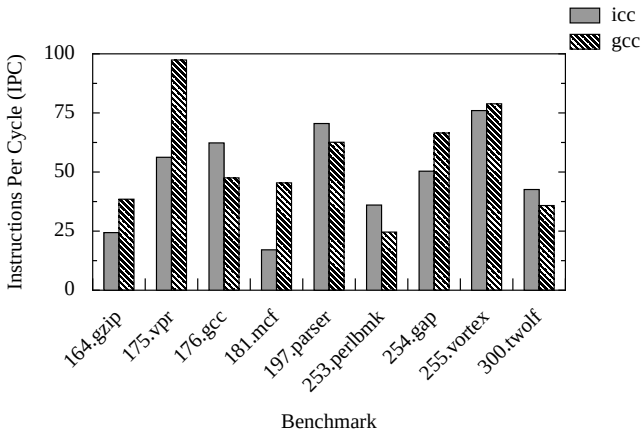


Figure 15: Compiler effect on average IPC for IA64

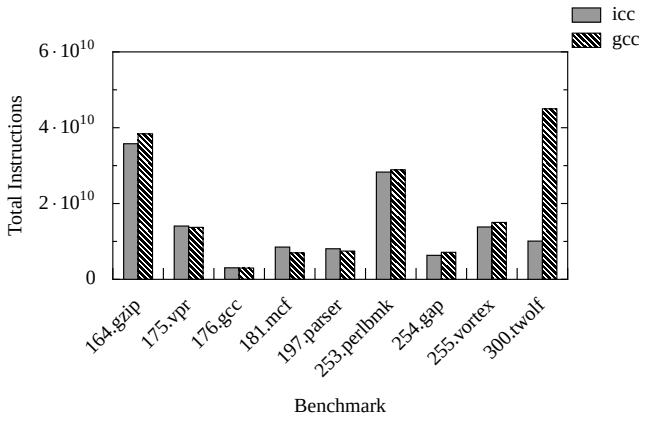


Figure 16: Compiler effect on total instructions for x86

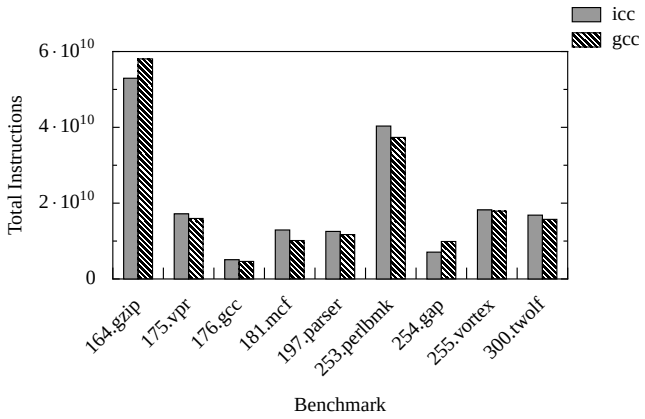


Figure 17: Compiler effect on total instructions for IA64

tectures based on average IPC values for the ideal case across all benchmarks in this study (for reference, Figures 16 and 17 show the total ops executed for each binary).

These figures show that, with x86, for most benchmarks the icc compiler exhibits lower average IPC values, except in the case of 181.mcf. The reason for the large amount of overall parallelism that the icc compiler is able to garner (compared to gcc) for the 181.mcf benchmark needs to be examined in detail to establish the nature of the compiler effects described above. It also remains to be determined why this advantage is absent for the other benchmarks. In the case of IA64, across all benchmarks, the performance of both icc and gcc compilers are similar and it appears that neither compiler diplays a clear advantage. Here, it is important to note that the values presented for x86 are for runs that ignore the stack pointer dependence, thus control dependences are eliminated. Because predication converts control dependences (which are ignored by Adamantium in this study) into data dependences (which are not ignored in Adamantium), there is an unfair disadvantage for more aggressive compilers on Itanium. This effect is discussed further in Section 3.5.

Before concluding this section, it should be noted that icc does do a better job than gcc in extracting performance. Figures 18 and 19 compare the two compilers based on the execution time on real hardware (900 MHz Itanium 2 processors and 2.4 Intel Xeons with

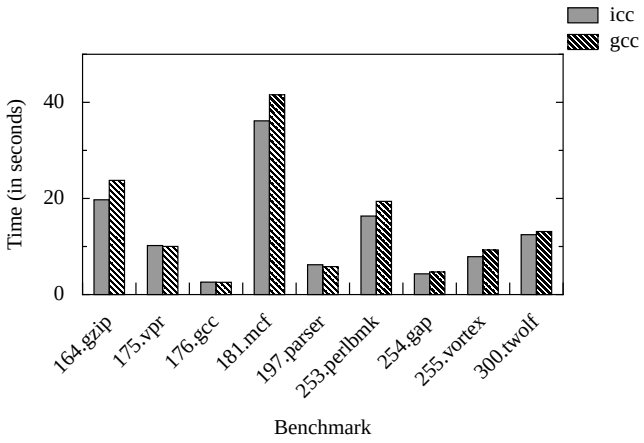


Figure 18: Compiler effect on real execution time for x86

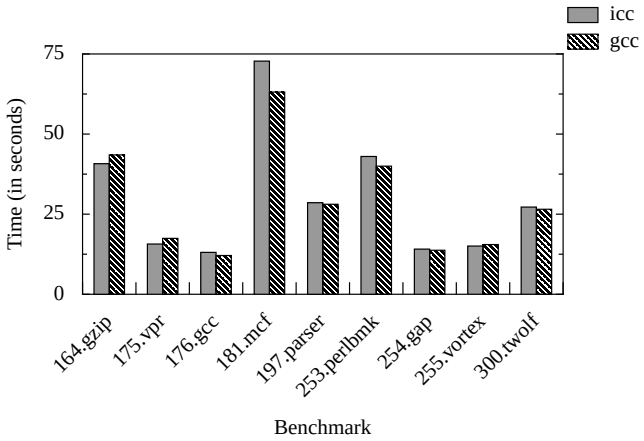


Figure 19: Compiler effect on real execution time for IA64

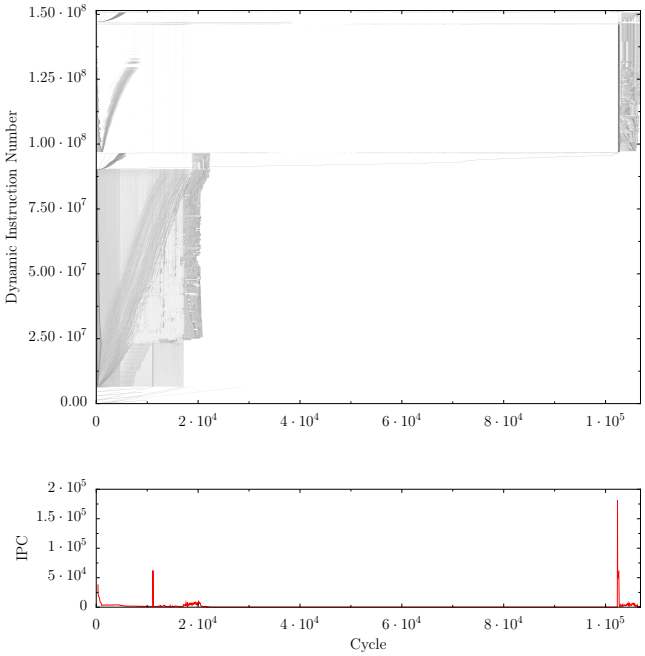


Figure 20: DIN versus ready-time plot for 181.mcf (x86,gcc,inf)

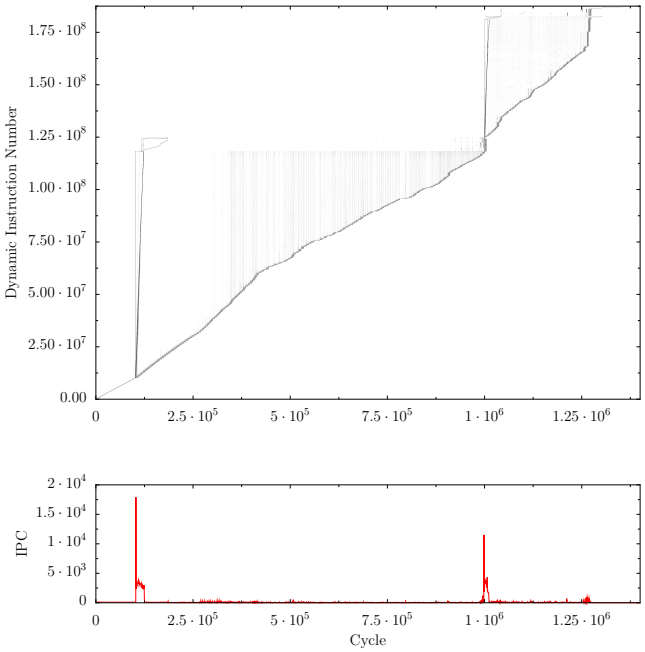


Figure 21: DIN versus ready-time plot for 181.mcf (ia64,gcc,inf)

2 GB RAM on both platforms). The icc compiler has a clear advantage, and almost always executes code faster than the gcc compiler, for x86. For Itanium, on the other hand, neither compiler is clearly better (though we expect a dramatic advantage for icc on floating point codes).

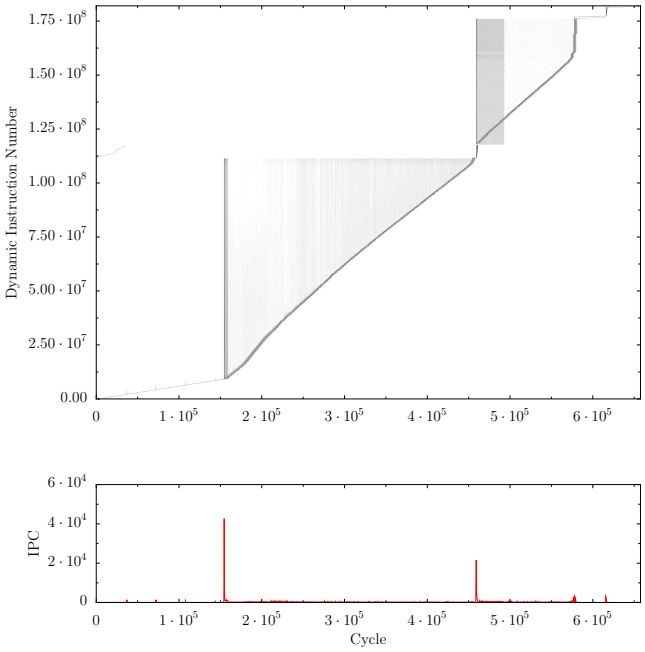
3.5 Effect of ISA on IPC

Intuition tells us that the instruction-set architecture targeted by the compiler may have a drastic effect on ILP. However, this intuition is built upon a working knowledge of application-ISA interaction in the presence of microarchitectural constraints. For example, Intel's x86 architecture is usually considered ill-suited for ILP because its dearth of architected registers results in excessive memory delay due to register-spill and fill code. However, in the ideal case, memory operations are no more costly than other operations, and thus spill-fill code may not have a significant effect on the ideal ILP. (This is not to say that excessive memory operations are not a problem, but they are not the focus of this paper). In this section we examine the role that the ISA and ISA-related factors play in ideal ILP.

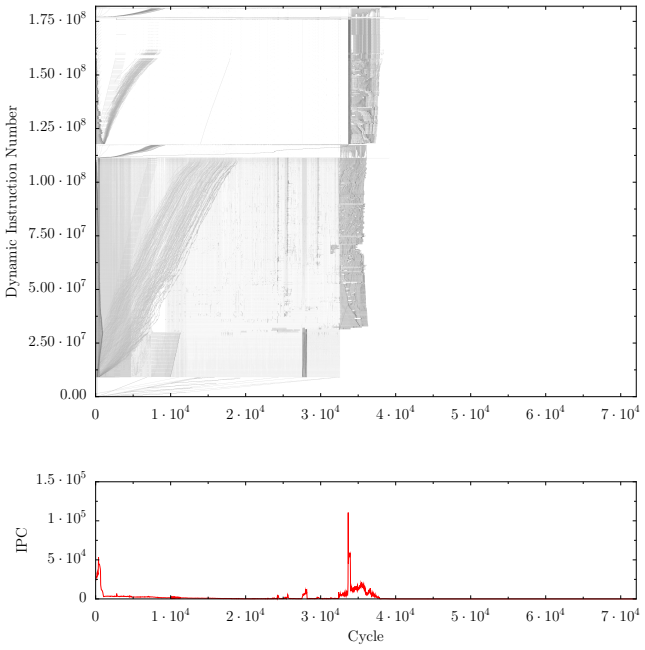
Figure 20 and Figure 21 show the DIN versus. ready-time plots and IPC histograms (seen earlier in the paper) for the 181.mcf benchmark and its test input with configurations (x86,gcc,inf) and (ia64,gcc,inf). From this perspective it appears that the overall IPC available in the two benchmarks is quite different, especially in terms of the far-ILP. This contradicts the intuition that far-ILP ought to be a product of high-level application properties and not implementation details.

To examine the dependence on far-ILP further, consider Figure 22a. This figure shows the DIN versus ready-time plot for the same run of the 181.mcf benchmark and test input using the PowerPC ISA, in particular (ppc,gcc,inf). Notice that this plot is, overall, a match for the same 181.mcf plot on the Itanium architecture. Now, consider Figure 22b. This is the same 181.mcf run for (ppc,gcc,inf), but ignoring the stack pointer (as described for the x86 data). Notice here, that this plot is nearly identical in shape to the x86 DIN plot. A similar match between plots is observed for other benchmarks. This matchup arises because, like x86, PowerPC and Itanium using arithmetic stack pointer updates. The difference is that PowerPC and Itanium still have considerable ILP, even in the presence of the arithmetic stack pointer updates. Due to the small number of registers on x86, however, the stack pointer serializes much of the computation due to the register spill and fill code in the binary.

Unfortunately, this explanation is still not satisfactory. Itanium too has a stack pointer but, intuitively, the hardware stacked registers and large architected register file should greatly reduce the number of instructions that need the stack pointer. Indeed, generating the same 181.mcf run's DIN vs. ready-time plot when ignoring r12 (the stack pointer on Itanium) shows no major difference from the plot in Figure 21. On the other hand, consider Figure 23, which shows the same DIN vs. ready-time plot for Itanium but this time ignoring all predicate registers. Notice that this plot looks very similar to to the same plot for x86 (Figure 20) and PowerPC (Figure 22). Thus, the predicate registers on Itanium are somehow limiting the ideal case far-ILP. The reason for this is that the Adamantium scheduler is doing perfect branch prediction which allows much more parallelism than would otherwise be possible. On Itanium, however, predication converts control dependences into data dependences on the predicate registers. Ignoring the predicate registers is tantamount to assuming perfect predicate prediction, and thus brings the Itanium plots in line with the observed PowerPC and x86 data. In fact, the critical dependences shown in Section 3 for 181.mcf are based on predicate registers. It is tempting to attribute the influence the compiler has on far-ILP to this effect alone, but note that in Section 3.4, the compiler can influence far-ILP even for x86 code which does not support predication, and thus compiler effects still warrant further investigation.



(a) Normal Plot



(b) Plot Ignoring Stack Pointer

Figure 22: DIN versus ready-time plot for 181.mcf (ppc,gcc,inf)

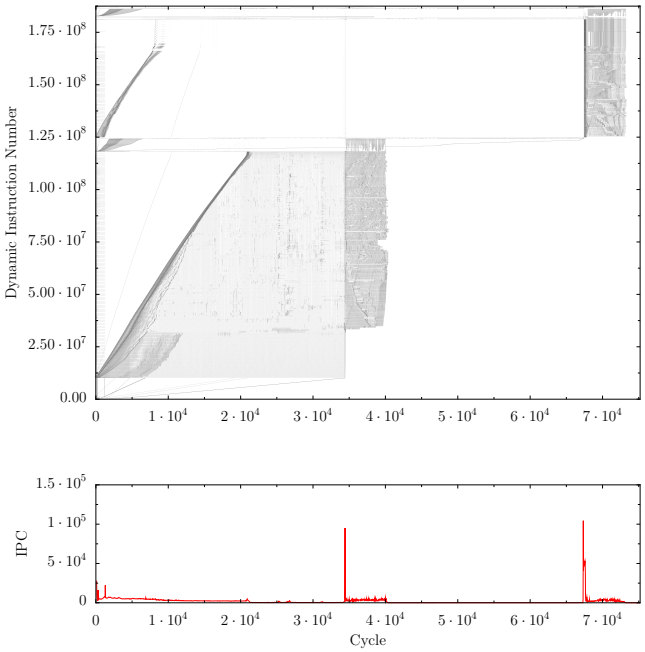


Figure 23: DIN versus ready-time plot for 181.mcf (ia64,gcc,inf), ignoring predicate registers.

In summary, we can see from the above overall IPC numbers presented earlier that the architecture can make a difference in the overall maximum amount of ILP available. However, when a few key bottlenecks are removed (e.g., the stack pointer dependences for spill-fill code on x86 and lack of predicate prediction on Itanium), it is clear that the ISA does *not* have a profound influence on the overall pattern of available far-ILP in a program. This matches the notion that far-ILP ought to be a product of application characteristics, not ISA particulars. Notice, however, that subtle differences in the treatment of architectural features and implementation details in the application-binary interface (ABI) can play a significant role in the availablity of far-ILP and the overal ILP pattern. For x86 and PowerPC, the decision to reduce memory operations by implementing arithmetic stack pointer updates dramatically limited the availability of ILP, especially far-ILP. On Itanium, treating predicate register based "control-flow" differently than branch based control flow dramatically limited far-ILP. Thus, when designing systems to extract far-ILP, the compiler and ABI implementation must be carefully tuned to avoid artificially limiting far-ILP for little-to-no gain in near-ILP.

4. RELATED WORK

As we have seen, understanding the limits and origins of ILP, especially distant ILP, can be instrumental to future research towards better performance and utilization of emerging chip multi-threaded and chip multi-processor architectures. Numerous researchers have presented ILP limit studies in the past [23, 18, 12, 13, 24, 19, 25, 26]. Unfortunately, many of these studies are out-of-date, and provide only limited information about distant ILP and the role of compiler and ISA on the availability (and origins) of this ILP.

Though outdated, previous research provides a rich set of information regarding the behavior of near ILP. Wall studies the effect of branch windows, memory disambiguation, load-latency, branch prediction, and a variety of other factors on the ILP extractable

from applications, assuming an instruction window capable of holding 2000 instructions (henceforth referred to as a 2k-instruction window). Wall shows that in the absence of near perfect branch prediction and memory disambiguation, extracting much of the ILP within the 2k-instruction window is very difficult, leading to supporting the claim in Section 1 that extracting near-ILP is difficult. A number of later studies confirm Wall's findings [23, 19, 25]. When the study in this paper examined conditions similar to the prior work, we observed similar results. Unfortunately, Wall's research has an extremely limited instruction window and thus does not characterize distant ILP.

For near ILP, control flow is known to be a major limiting factor and the effect on ideal ILP of methods to overcome these effects have also been studied [18]. Postiff showed that certain non-essential dependences (such as those based on arithmetic stack pointer updates) are also extremely harmful to near ILP [19]. The results contained in this paper confirm Postiff's findings. Once again, these studies do not examine the effects of compiler and ISA on the ILP available in an application, making it difficult to sort out application properties from these other factors.

Increasing the instruction issue window size unlocks more-distant ILP. To fully characterize the available parallelism for future architectures, it is important to examine this unlocked parallelism, and to determine its utility. Previous analyses [23, 24] demonstrate the increase in available parallelism with larger window sizes but fall short at explaining the source of this parallelism with respect to the application under consideration. Furthermore, these studies, once again, do not characterize how different compiler optimizations influence ILP, especially distant ILP, nor do they examine multiple ISAs under similar experimental setups.

Finally, the availability of ILP based on increasing the window size, shows a marked dependence on application characteristics, yet no attempt has previously been made to explain this observation. Previous studies fail to categorize the available parallelism. Our study compares architectures, compilers and window sizes, while attributing the increase or decrease in ILP to one of these factors. Such an extensive comparison allows a deeper understanding of the underlying issues that need to be immediately addressed in order to exploit the stated parallelism gains.

In summary, prior ILP limit studies show that there is much parallelism available, but that near-ILP will be difficult to extract. None of the studies described above examine features of the ILP that are needed to guide research into extracting this ILP for use on multi-core systems. The study presented in this paper begins just such an analysis by examining the role of compiler and ISA on distant ILP, and presenting an initial method to characterize the origins of that ILP.

5. CONCLUSIONS AND FUTURE WORK

Parallelism, historically, has been the primary architectural means for enhancing performance of computer systems. Unfortunately, existing means of exploiting near instruction level parallelism (ILP) such as more aggressive hardware and more advanced peephole compiler optimizations are reaching their limits. This is evidenced by the emergence of multi-core and multi-threaded systems as the design path of choice. These multi-core and multi-threaded systems improve threaded applications, but do little for single-threaded programs.

This paper presents an ILP limit study that examines the intrinsic distant ILP in applications with the aim of revealing promising directions for exploiting multi-core architectures for single-threaded programs. The study focuses on examining the role of the compiler in exposing and creating distant parallelism, as well as the role of the ISA in allowing the exposure of distant parallelism. Contrary to common wisdom and intuition, the results show that local compiler optimizations can have a dramatic effect on the overall ILP in an application binary. Furthermore, results indicate that the major features of an ISA only play a minor role in the character of distant ILP, however minor platform implementation details (e.g., the mechanism used to manipulate the stack pointer) can have a much larger effect. These counter-intuitive results warrant a more detailed explanation. In particular, a close examination of how different optimizations affect distant ILP is in order. The observation that compiler efforts to extract near ILP often destroys distant ILP is particularly interesting.

Most importantly, this study identifies several characteristics of distant ILP, two of which are of particular interest. The first is the notion of critical dependences (small sets of dependences that unlock large amounts of future parallelism), and parallel dependence chains (independent parallel chains of instructions unlocked by critical dependences). These results suggest several promising avenues of research for threading programs including value predictions of critical values and dynamic threading of applications when parallel dependence chains are unlocked. Though much work remains in fully characterizing the nature of distant ILP and exploiting this ILP in actual hardware, these initial results give promise for improved single-thread performance by exploiting multi-core platforms.

Acknowledgements

We thank Carole Dulong and Intel for the donation of resources and the Itanium workstations used in this study. We also thank Vijay Janapa Reddi and the Pin team at Intel for their support in using the Pin tool. We also thank the Liberty group and Jonathan Chang for the PowerPC emulator ans support in using it. Finally, we thank Martin Burtscher for his support with the TCgen trace compression utility. Computer time was provided by NSF ARI Grant #CDA-9601817, NSF MRI Grant #CNS-0420873, NASA AIST grant #NAG2-1646, DOE SciDAC grant #DE-FG02-04ER63870, NSF sponsorship of the National Center for Atmospheric Research, and a grant from the IBM Shared University Research (SUR) program. Other support was provided by donations from Intel.

6. REFERENCES

[1] D. N. Armstrong, H. Kim, O. Mutlu, and Y. N. Patt, "Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery," in *MICRO-37: Proceedings of the 37th International Symposium on Microarchitecture*, pp. 119–128, IEEE Computer Society, 2004.

[2] T. Moreshet and R. I. Bahar, "Power-aware issue queue design for speculative instructions," in *DAC '03: Proceedings of the 40th conference on Design automation*, pp. 634–637, ACM Press, 2003.

[3] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose, "Energy efficient co-adaptive instruction fetch and issue," in *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pp. 147–156, ACM Press, 2003.

[4] R. A. Ravindran, R. M. Senger, E. D. Marsman, G. S. Dasika, M. R. Guthaus, S. A. Mahlke, and R. B. Brown, "Increasing the number of effective registers in a low-power processor using a windowed register file," in *CASES '03: Proceedings of the 2003 international conference on Compilers, architectures and synthesis for embedded systems*, pp. 125–136, ACM Press, 2003.

[5] M. J. Flynn, P. Hung, and K. W. Rudd, "Deep-submicron microprocessor design issues," *IEEE Micro*, vol. 19, no. 4, pp. 11–22, 1999.

[6] L. Spracklen and S. G. Abraham, "Chip multithreading: Opportunities and challenges," in *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, pp. 248–252, IEEE Computer Society, 2005.

[7] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," in *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pp. 2–11, ACM Press, 1996.

[8] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 177–188, September 2004.

[9] P. Marcuello and A. Gonzalez, "Clustered speculative multithreaded processors," in *International Conference on Supercomputing*, pp. 365–372, 1999.

[10] A. Cristal, O. J. Santana, M. Valero, and J. F. Martinez, "Toward kilo-instruction processors," *ACM Transactions on Architecture and Code Optimization (TACO).*, vol. 1, no. 4, pp. 389–417, 2004.

[11] I. Martel, D. Ortega, E. Ayguad, and M. Valero, "Increasing effective ipc by exploiting distant parallelism," in *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pp. 348–355, ACM Press, 1999.

[12] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the 4th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188, April 1991.

[13] D. W. Wall, "Limits of instruction-level parallelism," Tech. Rep. 93/6, DEC WRL, November 1993.

[14] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 International Symposium on Programming Language Design and Implementation (PLDI)*, (Chicago, IL), June 2005.

[15] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, "Microarchitectural exploration with Liberty," in *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, pp. 271–282, November 2002.

[16] The Liberty Research Group, 2005. http://www.liberty-research.org/.

[17] M. Burtscher and N. Sam, "Automatic generation of high-performance trace compressors," in *Proceedings of the 2005 International Conference on Code Generation and Optimization*, 2005.

[18] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 46–57, May 1992.

[19] M. Postiff, G. Tyson, and T. Mudge, "Performance limits of trace caches," Tech. Rep. CSE-TR-373-98, University of Maryland, Department of Electrical Engineering and Computer Science, CSE, September 1998.

[20] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proceedings of the 9th International Symposium on High Performance Computer Arechitecture*, February 2003.

[21] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Dynamically allocating processor resources between nearby and distant ilp," in *Proceedings of the 28th annual international symposium on Computer architecture*, pp. 26–37, ACM Press, 2001.

[22] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A large, fast instruction window for tolerating cache misses," *SIGARCH Computer Architecture News*, vol. 30, no. 2, pp. 59–70, 2002.

[23] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi, "Dynamic dependency analysis of ordinary programs," in *Proceedings of the 19th International Symposium on Computer Architecture (ISCA-19)*, May 1992.

[24] P. Ranganathan and N. P. Jouppi, "The relative importance of memory latency, bandwidth, and branch limits to performance," in *Proceedings of the Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.

[25] H. H. Lee, Y. Wu, and G. Tyson, "Quantifying instruction-level parallelism limits on an epic architecture," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 21–27, April 2000.

[26] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proceedings of the 2004 International Symposium on Computer Architecture (ISCA)*, pp. 76–89, 2004 2004.