Quantifying Instruction-Level Parallelism Limits on an EPIC Architecture

Hsien-Hsin Lee[†‡] *Youfeng Wu*[‡] *Gary Tyson*[†]

[†]*ACAL, EECS Department*
University of Michigan
Ann Arbor, MI 48105
{linear, tyson}@eecs.umich.edu

[‡]*Microprocessor Research Labs*
Intel Corporation
Santa Clara, CA 95052
youfeng.wu@intel.com

ABSTRACT

EPIC architectures heavily rely on state-of-the-art compiler technology to deliver optimal performance while keeping hardware design simple. It is generally believed that an optimizing compiler has an enormous scheduling window to exploit instruction-level parallelism (ILP) since compiler orchestrates the entire program. In this paper, we will show that this presumption is overly optimistic when the scopes of compiler transformations are constrained to loop and function boundaries; many state-of-the-art compilers typically confine optimizations to loop boundaries (e.g. software pipelining, trace scheduling, and loop unrolling) and function boundaries (e.g. loop peeling, loop exchanges, invariant hoisting, and global optimizations). Although techniques such as function inlining and interprocedural optimizations can alleviate these constraints to a limited extent, loop and function boundaries are often the real scopes to the compiler scheduler. Several previous ILP studies have explored the limits in parallelism on dynamic superscalar machines; however, those results are not applicable to EPIC architectures since they rely on dynamic scheduling, not static code schedule by the compiler, to reorder instructions. In this paper, we evaluate the limits in ILP obtained through compiler scheduling alone. We quantify these limits as more restrictive scheduling constraints are imposed — starting from inter-procedural code scheduling, to intra-procedural and finally to loop-confined code scheduling.

I. INTRODUCTION

Supplying instructions and data in a timely manner are fundamental design issues in modern computer system. Many computer architecture and compiler researchers are investigating new techniques to enhancing the number of effective instructions issued into the execution core as well as reducing delays of data delivery. New microarchitecture features of superscalar processors, such as instruction prefetch, dynamic and speculative instruction issue, novel branch prediction mechanisms, trace caches, data prefetching, etc., have been proposed to address and/or resolve these issues, largely, in a dynamic fashion. These techniques rely on intelligent hardware designs to exploit the instruction-level parallelism (ILP) and maximize instruction supply during program execution. For a static implementation of the Intel/HP EPIC architecture (e.g. Itanium), compilers take a full responsibility to exploit instruction level parallelism from a given program. Innovative ISA features such as instruction predication and control speculation are also governed and generated by compilers to improve the execution efficiency. In this way, the design complexity of dynamic scheduling via hardware can be significantly reduced. This enables the processor frequency to increase due to simplified control logic in the hardware. It is generally believed that a large amount of ILP can be made available by the compiler (except for the dependencies posed by ambiguous memory addresses), since the compiler manipulates the entire program.

However, existing compilers often limit instruction scheduling to loop instances or function level (i.e. "global" transformations). Loop peeling and code hoisting can break the constraints of intra-loop instruction scheduling, while function inlining can enlarge the scheduling window by merging instructions from the inlined function call with the calling function. However, these optimization techniques are only applicable to a limited extent. For example, loop peeling and function inlining often significantly expand the code size increasing pressure on the I-cache. In this study, we investigate the ILP limits using IA64 code compiled by the Intel research compiler. Under perfect microarchitecture assumptions, we will show the ILP upper bounds constrained only by true data dependency for SPECint95 benchmarks as scheduling constraints are gradually imposed – starting from inter-procedural code scheduling, to intra-procedural and finally to intra-loop code scheduling. Using this approach, we increase our understanding of how and where the ILP decreases as more compiler scheduling constraints are imposed.

This paper is organized as follows. Section II gives background information on the relationship between language, compiler and instruction-level parallelism. Section III defines our technique of capturing ILP with different instruction scheduling windows. Section IV describes our simulation infrastructure. We discuss our results in Section V. Finally, we conclude this work in Section VI.

II. COMPILER SUPPORT FOR EXPLOITING ILP

A compiler translates the algorithm represented by a high-level language into a specific machine code. The performance of the compiler and the architecture of the target machine have a significant impact on the performance of the program (second only to the programmer). For a conventional dynamic superscalar machine, the processor has the capability to narrow performance gap of different compilers by reordering instructions on the fly to improve execution performance. The philosophy of the EPIC architecture is to reduce the complexity of the processor implementation by relying on a sophisticated compiler to exploit an enormous amount of instruction level parallelism by scheduling instructions across the entire program.

In this research, we investigate the ILP of ordinary programs in a processor design relying on a compiler to perform all instruction scheduling. Prior ILP studies [1] [4] [5] [10] are not applicable to an architecture like EPIC in which compiler's capability is crucial. Therefore, we propose new performance analysis techniques to model the ILP seen from a compiler's scheduling window. The new techniques impose extra scheduling constraints to prevent reordering across function and/or loop boundaries. This analysis is performed by simulating the binary after appropriately applying optimization techniques such as loop peeling, code hoisting and function inlining. We believe that the ILP measurements in this study more closely approximate the parallelism that a real EPIC compiler can achieve.

III. TOP-DOWN APPROACH TO INSTRUCTION SCHEDULING

We present three techniques to examine the limits in ILP obtained through compiler scheduling. In a top-down fashion, we quantify these limits as more restrictive compiler scheduling constraints are imposed — starting from interprocedural code scheduling, to intra-procedural and finally to loop-confined code scheduling.

A Interprocedural Scheduling Without Limitation

Given a program, we would like to understand its limit in ILP using the whole program as a single scheduling window. Similar to the methodology adopted in [7], this technique exploits the ILP limits of an ideal code schedule based on an execution-driven, functional simulation. To accomplish this, resource hazards are eliminated and instruction and memory latencies are reduced to one cycle. The machine model simulated will be described in Section IV. Our analysis tracks flow, output and anti-dependencies by examining the definition and use of each physical register and each memory location. Due to the enormous number of memory locations (2^{64} addresses in IA64 architecture), a four-level indirect hash table was implemented to reduce the memory space consumed. Each level contains 64k entries (16-bit) and each of them points to a 64k-entry table of the next level. We dynamically allocate space for each new memory accessed, in the granularity of 64KB. The fundamental data structures shown as follows are used for tracking dependencies.

- DEF[REG] = Reg_Def_TimeStamp[physical register ID]
- USE[REG] = Reg_Use_TimeStamp[physical register ID]
- DEF[MEM] = Mem_Def_TimeStamp[address]
- USE[MEM] = Mem_Use_TimeStamp[address]

To determine the earliest scheduling/issueable clock cycle of an instruction, the dependencies from all the source and destination operands of that instruction must be determined. Each operand of an instruction is checked against its most recently defined and used timestamps. The final scheduling timestamp is designated according to the dependence types and lengths. Flow-dependency (RAW), output-dependency (WAW) and anti-dependency (WAR) are all detected by checking the registers and memory addresses accessed by each instruction in order to maintain a legitimate instruction execution sequence. These dependencies are calculated as follows.

- RAW = max{DEF[REG] | REG $\in$ src operands }
- WAW = max{DEF[REG] | REG $\in$ dest operands }
- WAR = max{USE[REG] | REG $\in$ dest operands }
- MRAW = max{DEF[MEM] | MEM $\in$ src operands }
- MWAW = max{DEF[MEM] | MEM $\in$ dest operands }
- MWAR = max{USE[MEM] | MEM $\in$ dest operands }

After all the dependency types of each operand are resolved, the earliest scheduled issue cycle of the target instruction can be calculated by taking the maximum dependency as shown below.

- Issue_Cycle = max(RAW, WAW, WAR, MRAW, MWAW, MWAR)

Even though the technique can optimally place and issue each instruction at the earliest possible cycle, it does not represent the upper bound that a given program can achieve. In theory, more parallelism will be available if false dependencies are eliminated. Due to the finite number of architectural registers manipulated by a compiler, the compiler must reuse registers during register allocation phase and as a result create false register dependencies. Perfect dynamic register renaming or an infinite architectural register file can increase ILP by removing the components of register false dependencies, i.e. setting WAW and WAR to zeros from our last equation. Hence, after registers are renamed, the last equation becomes

- Issue_Cycle = max(RAW, MRAW, MWAW, MWAR)

Similarly, memory false dependencies can theoretically be avoided. Tyson and Austin proposed a hardware-based memory renaming technique in [9] in which a store/load cache and value file determines the producer and consumer relationship between memory operations, translating those references to register-like accesses in the value file. Standard register renaming techniques can then be applied to the value file items to remove false memory dependencies. Assuming memory false dependencies can also be completely eliminated, i.e. setting MWAW and MWAR to zeros, the earliest scheduled issue cycle of an instruction is now only constrained by flow dependencies. The new scheduling cycle is computed by

- Issue_Cycle = max(RAW, MRAW)

Even this equation does not calculate a lower bound on execution time if more aggressive microarchitecture enhancements such as value prediction [6] and/or dynamic instruction reuse [8] are considered. However, performing perfect value prediction would result in a meaningless limit of zero cycles to execute any program.

B Function-confined (Intra-procedural) Scheduling

The equations presented in previous section assume compilers not only perfectly schedule instructions across basic blocks with perfect branch information, but can schedule instructions across function boundaries without limit. In order to reach a more realistic ILP, a function-confined instruction window is introduced to limit the scheduling of instructions from different functions. First, each function is considered as an atomic scheduling region. Instructions from different regions cannot be issued at the same time. We use *call* and its corresponding *return* instructions as the head and tail delimiters for each atomic scheduling region. All instructions other than head and tail delimiters should be scheduled no earlier than their head delimiter and no later than their tail delimiter of their corresponding atomic scheduling region. The following notations are defined to illustrate our technique.

• T(inst, n): scheduled or issueable cycle (i.e. timestamp) of inst at call depth n.
• RUSE(n): set of all the resources used at call depth n.
• RDEF(n): set of all the resources defined at call depth n.

An integer n representing call depth is assigned for each function starting from main() with n = 1. For each function call instance during the execution, n is incremented and associated with the new function. It is decremented when a function is returned. Note that n is not unique and can be associated with different functions at different time. The issue cycle of an instruction, *inst*, at call depth n is represented by T(inst, n). We also keep the list of resources used and defined at call depth n represented by RUSE(n) and RDEF(n), respectively. Besides register and memory dependencies described in Section III, another component called function control dependency, FCTRL, was incorporated into the calculation of the issue cycle. FCTRL associated with each resource is used to track the dependency resulted from the limitation of a function boundary. Similar to DEF[r] and USE[r], FCTRL of a resource r are denoted by DEF[r].FCTRL and USE[r].FCTRL. Upon the return instruction being evaluated of a function, the critical path of the function, i.e. the latest issued instruction, is used as the issue cycle of the return instruction and later used to update the FCTRL cycles of all the resources used and/or defined inside this function. Subsequent instructions outside this function check both the resource dependencies and its corresponding FCTRL to determine their final issue cycles. These constraints are formulated as follows:

• T(ret,n) = max(T(i, n)) where i ∈ instructions executed at call depth n.
• USE[r].FCTRL = T(ret, n) if r ∈ RUSE(n), otherwise 0.
• DEF[r].FCTRL = T(ret, n) if r ∈ RDEF(n), otherwise 0.
• RAW_FCTRL = max{DEF[REG].FCTRL | REG ∈ src operands }
• WAW_FCTRL = max{DEF[REG].FCTRL | REG ∈ dest operands }
• WAR_FCTRL = max{USE[REG].FCTRL | REG ∈ dest operands }
• MRAW_FCTRL = max{DEF[MEM].FCTRL | MEM ∈ src operands }
• MWAW_FCTRL = max{DEF[MEM].FCTRL | MEM ∈ dest operands }
• MWAR_FCTRL = max{USE[MEM].FCTRL | MEM ∈ dest operands }

In essence, once a program is compiled, it is impractical for a static machine to lift and schedule instructions in a function before the function is called and executed. Based on this rationale, all instructions internal to a particular function must be scheduled and issued no earlier than their corresponding function call instruction. To satisfy this constraint, LAST_CALL is introduced into the calculation of issue cycle. This variable precludes future instructions inside the callee

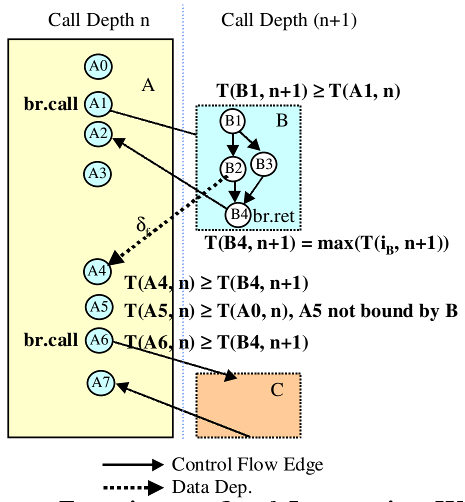

Figure 1: Function-confined Instruction Window

function from being scheduled ahead of the call.

It is also impossible to overlap two atomic scheduling regions in compiler scheduling for a static machine, although it is possible if dynamic instruction scheduling by hardware is allowed. Therefore, for two consecutive function calls, these two atomic scheduling regions are serialized as seen in the original program order. This is satisfied by the definition of LAST_RET(i), which impedes a subsequent call instruction to be scheduled ahead of the most recent return instruction. Under such circumstances, function calls are always scheduled in the original program order.

• LAST_CALL = T(call, n-1) for a call depth n.

• LAST_RET(i) = T(ret,n+1) for i ∈ call-type instructions at call depth n; otherwise, LAST_RET(i) = 0.

Finally, the definition of max.FCTRL assembles all the function-boundary-related control dependencies generated and the issue cycle is computed with one more function control dependency considered in the original Issue_cycle equation.

• max.FCTRL = max(LAST_CALL, LAST_RET(i), RAW_FCTRL, WAW_FCTRL, WAR_FCTRL, MRAW_FCTRL, MWAW_FCTRL, MWAR_FCTRL)

• Issue_Cycle = max(RAW, WAW, WAR, MRAW, MWAW, MWAR, max.FCTRL)

Figure 1 illustrates a simple example to show how this technique works. In this code sequence, function A calls function B and C. Each node inside each function represents one instruction, e.g. A1 is a call instruction that calls function B. The fundamental goal is to restrict instructions of each function into an atomic scheduling region. If the caller has a dependency from the callee, the dependent instructions from the caller can only be scheduled after the last issued instruction (return) of the callee. In Figure 2, instruction B1 cannot be scheduled earlier than its corresponding call instruction, A1. The return instruction, B4, is scheduled as the last instruction issued in B. Since A4 is dependent of B2, A4 is

Symbol	Purpose	Size
sip	Starting chunk IP	8 Bytes
n	A chunk size	4 Bytes
i	Complementary loop ID	4 Bytes

Table 1: Chunk Information for Identifying Loops

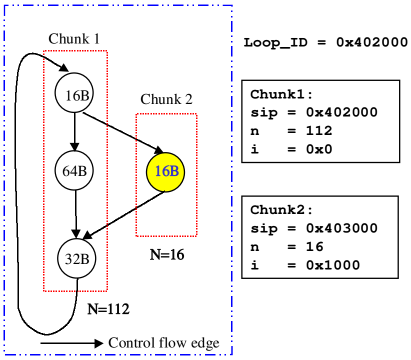

Figure 2: A Loop Composed of 2 Chunks

only scheduled after B4 is scheduled. Conversely, A5, independent of function B and A4, can be scheduled as early as the issue timestamp of A0, the first instruction of function A. The second call instruction A6 of function A and its entire code body function C are scheduled after function B.

C Loop-confined Window — Intra-loop Scheduling

As aforementioned, compilers are unlikely to schedule instructions from distinct loop bodies. In other words, instructions in a loop are considered to be within an atomic scheduling region and are not scheduled concurrently with instructions from other loops unless high-level code transformation techniques such as loop fusion are performed. In order to approximate this constraint and investigate its impact to our ILP study, we introduce a loop-confined scheduling window as an additional instruction scheduling limiter on top of the function-confined scheduling window.

It is difficult to identify all the complex loop structures inside a program through a single-pass execution driven simulation. Hence, we acquire this information during compilation-linking time. To provide this information, the IA64 research compiler was instrumented for generating an extra section of binaries where loop structures are encoded into a designated representation as shown in Table 1.

In this representation, a loop is broken down to one to several chunks, each represents a sequence of consecutive blocks after compiler's block ordering optimization. A chunk can consist of several basic blocks and basic blocks of different chunks do not overlap. A unique loop ID (a unique address for an instruction in the loop) is assigned by the compiler for those chunks which compose a loop. To reduce the size of the loop structure information, we store the complementary loop ID, i (4 bytes), instead of the unique loop ID (8 bytes). For each chunk of a loop, its complementary loop ID is computed by subtracting the unique loop ID from its starting chunk IP (instruction pointer) address, i.e. i = *(sip - unique loop_ID)*. In later ILP simulations, the unique loop ID of each chunk is then inverted from the prior formula on-

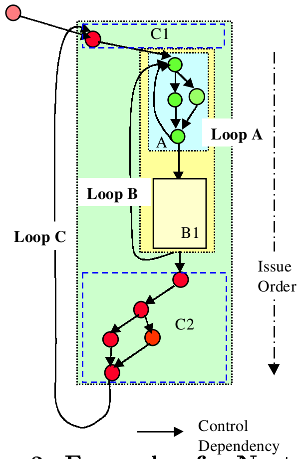

Figure 3: Example of a Nested Loop

the- fly to determine if instructions are in the same loop. Note that outer loop has a different loop ID from their inner loops for multi-level nested loops and unique loop IDs are also assigned for basic blocks outside loops for identifying instructions entering or exiting from loops. Exceptions of our study are those precompiled libraries which will be only be limited by function-confined scheduling window. Therefore, our simulated ILP numbers for loop-confined scheduling window will be somewhat optimistic if loops are present inside the precompiled library code.

Figure 2 shows a simple loop with a conditional branch. Each node of the loop represents a basic block. The number shown inside each node indicates the size in bytes of the instructions for the basic block. Two sequences of consecutive blocks (two chunks) are shown inside this loop. Total instruction size n for a chunk is obtained by summing the instruction sizes of the basic blocks of a chunk. In this example, the sizes (i.e. n's) for chunk1 and chunk2 are 112 bytes and 16 bytes respectively. The starting chunk IP, *sip*, is the beginning IP address of the first instruction of a chunk's entry basic block. Since chunk1 and chunk2 are in the same loop structure, their loop ID must be identical and are equal to their respective sip's minus their complementary loop IDs (i). Before any instruction is fetched and executed, the ILP simulator will read in the loop identification information generated by compiler and build up a loop structure table accordingly. The IP address of each instruction being executed is then tested against the loop structure table by the following condition, $IP \geq sip_x$ && $IP < (sip_x + n_x)$. If two instructions own the same loop ID, they are in the same loop. If a subsequent instruction has a different loop ID from its preceding instruction, then this instruction either has exited from a loop or entered into a new nested loop. We use Figure 3 as an example to elucidate our technique. There are three loops in this example. Loop A is an inner loop of Loop B which is encompassed by another outer loop, Loop C. To enable loop-confined scheduling window, the basic criterion is to schedule instructions in loop A as an atomic scheduling region, as well as Loop B and C. Therefore, instructions in block B1 cannot be scheduled until all the instructions in Loop A are scheduled. However, there exists some flexibility in scheduling instructions between two atomic instruction sequences. For example, instructions in block C2 can be scheduled into block C1 as long as there is

```
/* Instruction Loop at call depth n */
{
  if ( inst→loopID != last_loopID) {
      max_issue_clk[last_loopID] = max_issue;
      latest_loop_blk_issue_clk = max_issue;
      max_issue = 0;
  }
  .....
  /* After issue_cycle computed by
     function-confined scheduling window */
  if ( issue_cycle > max_issue_clk[inst→loopID]) {
      issue_cycle = latest_loop_blk_issue_clk + 1;
  }
  .....
  max_issue = MAX(issue_cycle, max_issue);
  last_loopID = inst→loopID;
}
```

Figure 4: Loop-confined Scheduling Algorithm

no data dependency between C2 and Loop A or between C2 and Loop B.

Figure 4 illustrates the algorithm of our intra-loop scheduling technique. In addition to loop ID used for identifying the transition when an instruction enters or leaves a loop, the following timestamps are also maintained and processed on each instruction's basis at a particular call depth.

• max_issue_clk(n, LID) : updated with the latest/maximum instruction issue cycle of a loop identified by LID, when exiting from this loop.

• latest_loop_blk_issue_clk(n) : keep track of the latestmaximum issue cycle of the latest loop region traversed.

The first variable, max_issue_clk(n, LID), keeps track of the maximum issue cycle among issued instructions of a loop at call depth n. This is used to check whether subsequent instructions of the same loop, e.g. instructions in region C2 of Figure 3, can be scheduled within the critical path of previously scheduled code of the same loop, e.g. C1. If they cannot, then the earliest cycle these instructions can be scheduled is right after latest_loop_blk_issue_clk(n) which accounts for the critical path of the latest loop code traversed.

D Semantic-only Dependency

In the IA64 instruction set architecture specification [2], architectural registers such as application registers and register stack engine (RSE) are handled by the machine itself and are not exposed to compilers. These registers can cause true dependencies that are implicit to IA64 instructions. For instance, when a function is called, the call-type branch instruction copies CFM (current frame marker) register to the PFM (previous frame marker), then restore the CFM from PFM as well as renamed registers back to the caller's configuration at return. Another example, a register spill (st8.spill) copies the NaT bit corresponding to the register to the common User NAT collection application register (AR36) which is read when a register fill (ld8.fill) restores the data. However, these dependencies can possibly be eliminated if a different hardware implementation is opted or more sophisticated compiler optimization techniques such as function inlining are invoked. We refer to a machine model without these implicit dependencies as a model with semantic-only dependencies, and a machine model with these dependencies as a baseline machine model.

IV. SIMULATION MODEL

A Machine Model

This research is intended to be independent of the microarchitectural impacts from a specific machine implementation. Therefore in our simulation environment, similar to most of the limit studies, we assume an ideal machine with perfect caches, infinite fetch width, and infinite number of issue ports and functional units, and instructions in each single issue can be arbitrarily bundled. All the instruction latencies are assumed to be one cycle, namely, a unit-latency machine model. In addition, the compiler has perfect knowledge of control flow during execution. The IA64 binaries were therefore compiled with predications and speculative loads turned off. Function inlining was enabled whenever appropriate to enlarge function sizes and number of instructions available to the compiler. Moreover, all nop instructions are excluded from ILP computation.

B ILP simulator

Our ILP simulator, ILPsim based on the algorithm presented in Section 3, is built on top of an execution- driven IA64 simulator which can fetch, decode, execute and retire IA64 instruction bundles in program sequence and provide simulation options in both functional and micro-architectural modes. The simulator translates each architectural register ID in the decode stage into a physical register ID for both explicit and implicit registers in order to establish precise resource dependencies during function calls and software pipelining loops.

V. SIMULATION RESULTS

Several studies [5] [7] [10] had shown that both theoretical and sustainable ILP in floating point applications are typically higher than those in integer applications, primarily due to the nature of more computational parallelism found in floating-point algorithms. Hence, researchers typically do not worry too much about the ILP in floating-point applications. For this study, we only concentrate on the available ILP in integer applications using SPECint95 as our benchmark suite.

A Metrics for Parallelism

Instruction per cycle (IPC) is used as the metric for measuring the parallelism of each compiled benchmark. When unit-latency is assumed for all the instructions, IPC is equivalent to ILP, instruction level parallelism. The final IPC (ILP) of an entire application is calculated by dividing the issue cycle of the critical path of the entire application critical path by the total number of instructions issued. Since our control speculation is 100% accurate, so the number of instructions issued is equivalent to the number of instructions executed.

B Result Analysis

The SPECint95 benchmarks were compiled using the Intel research compiler. All the simulations run up to 300 million instructions using the training input sets except for *perl*. *perl* was executed up to 80 million instructions because the

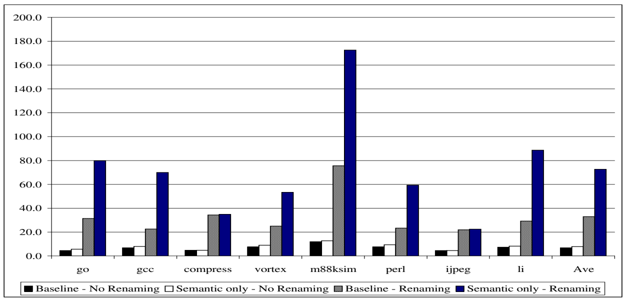

Figure 5: ILP with No Limit Scheduling

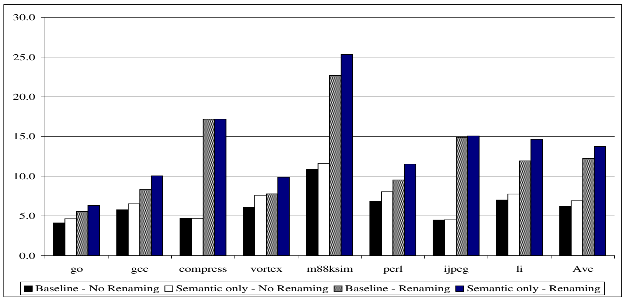

Figure 6: ILP with Function-confined Scheduling

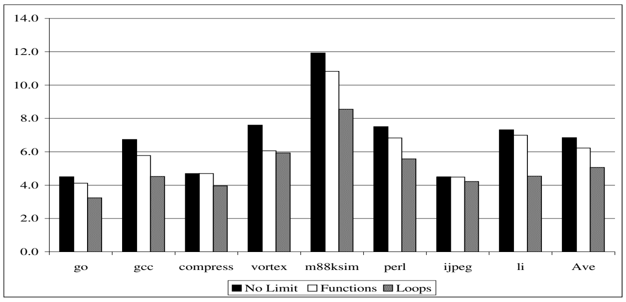

Figure 7: Comparison of Scheduling Windows

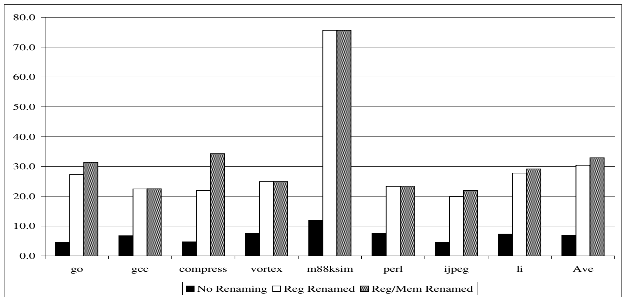

Figure 8: Performance of Different Resource Renamed, No Limit

function-confined algorithm could not manipulate longjmp at this time. Compared to the number of instructions of prior studies in [1] [7], we believe 80 million instructions can well address the goal of our study.

B.1 *ILP from Top-down*

Figure 5 shows the ILP results with and without resource renaming for baseline and semantic-only dependency models. Without any renaming, the average ILP of baseline model is 6.8 even when a large register file used in the IA64. However, if full resource renaming techniques including register and memory are applied, the ILP increases to an encouraging number of 32.9 for baseline model and 72.6 for semantic-only dependency model. The ILP results with function-confined scheduling window are considered in Figure 6. The high averaged ILP numbers with full renaming from Figure 5 are dramatically reduced by 5.3 times down to 13.7 for the semantic-only model and by 2.7 times to 12.2 for the baseline model. This implies that benefits from renaming are significantly diminished when we limit register allocation scope to call depths. Even so, intra-procedural resource renaming is still capable of delivering almost 2 times improvement (6.2 to 12.2 for baseline model and 6.9 to 13.7 for semantic-only model) over the ILP of the original compiled code. More resource renaming effects will be discussed in next section. Furthermore, the observation of ILP deviations from no limit on instruction scheduling window to function-confined scheduling window suggests that an opportunity to improve ILP does exist inter-procedurally. This suggests that a multi-threaded architecture [3] that supports multiple thread units, each with their own register files can exploit a greater amount of parallelism. A speculative independent thread, identified either by compiler or hardware, can be spawned as a separate light-weight execution thread to execute on one of the thread units.

B.2 *Performance effect with different scheduling windows*

Following a baseline machine model with no resource renaming, Figure 7 shows the ILP with different instruction scheduling windows perceived by a compiler. By using the architecturally available resources allocated, the average ILP is 6.8.This ILP drops 10% to 6.2 when the scheduling window is limited to function boundaries and drops 33% to 5.1 when the scheduling scope is further limited to loop boundaries. The ILP values for *compress* and *ijpeg* show little change from no limit on scheduling window to function-confined. This is due to the fact that intra-functional dependencies exist and these dependencies dominate critical paths of execution.

B.3 *Performance with resource renaming*

Register renaming can significantly improve instruction level parallelism by removing all the false dependencies created by register reuse. Memory renaming requiring extra hardware support as proposed in [9] can further improve ILP by removing false memory dependencies. The effect of eliminating false dependencies on ILP is illustrated in Figure 8 to Figure 10. With resource renaming, the ILP while scheduling across the entire program is increased close to five times.

When the scheduling windows constrained by functions and loops, the ILP are increased by 97% and 63% respectively. Once register renaming is applied, adding memory renaming support only renders marginal extra performance. For instruction scheduling limited by function boundaries, the ILP

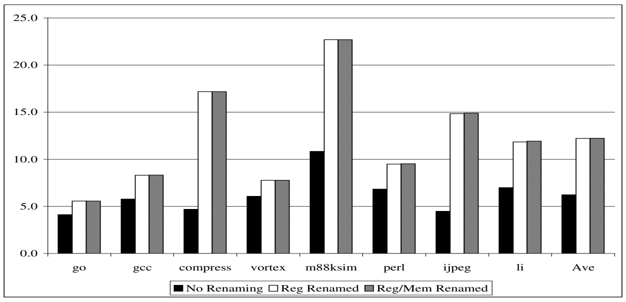

Figure 9: Performance of Different Resource Renamed, Function-confined window

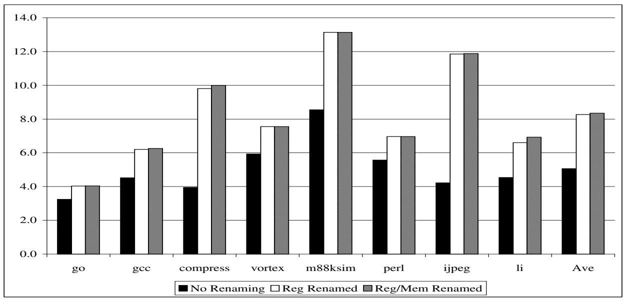

Figure 10: Performance of Different Resource Renamed, Loop-confined window

does not benefit from memory renaming at all. This is because much of the benefit of memory renaming is avoiding anti- dependencies between stack frame references which are deallocated, then immediately reallocated for the next function call; since our simulation does not permit overlapping the execution of functions in the loop constrained or function constrained experiments, much of the benefit of memory renaming is lost. Also note that the ILP constrained by loops with all false dependencies removed is 8.3 on average. It is generally agreed that effective ILP continues to shrink with the higher latencies on real machine implementations, exacerbating the difficulty of the compiler in filling instructions slots as latencies increase.

VI. CONCLUSIONS

In this paper, we propose a new performance analysis technique to be used to more closely predict and evaluate instruction-level parallelism for static machines such as EPIC. This technique is simple, useful, and effective in evaluating future microarchitecture designs. From the observations of using this technique, we reach the following conclusions on the compilers ability to extract parallelism on an EPIC architecture:

• A significant amount of parallelism exists in the benchmarks studied when instruction scheduling is not constrained and all false dependencies are removed. An average of 72.6 IPC is achieved for SPECint95 for a unit-latency machine model. When the implicit machine dependencies of current hardware implementation are taken into account, the IPC drops down to 32.9.

• The elimination of dynamic resource renaming reduces available ILP 6.8, 6.2 and 5.1, respectively, for an instruction scheduling with no limit, function-confined and loop-confined windows. These results point to the need for future research in compiler analysis and transformation techniques, and in microarchitectural enhancements to achieve higher degrees of ILP and break loop and function level constraints on compilers.

• When local register renaming within loops is performed ILP improves by 63% (IPC is increased from 5.1 to 8.3). This is primarily because the binaries used in the experiment are compiled for a particular IA-64 processor. The register allocation algorithm tries to minimize the number of registers used as long as the performance for that processor is not impacted.

• Finally, ILP can escalate from 6.8 to 32.9 when no instruction window restrictions are imposed and all false dependencies are lifted. This indicates performance opportunity for exploiting speculative thread-level parallelism (TLP) to improve single integer program performance.

VII. ACKNOWLEDGEMENT

The authors would like to thank Jesse Fang for his support on this research, Hong Wang for his technical assistance of the IA64 simulator infrastructure, and Yong-Fong Lee for his review of this paper. This research has been sponsored by the National Science Foundation (CAREER award #C036835) and Intel Corporation.

VIII. REFERENCES

[1] M. Butler, T-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *ISCA-18*, 1991.

[2] Intel Corporation. Ia-64 application developer's architecture guide. Intel Literature Centers, 1999.

[3] P. Dubey, K. O'Brien, K. O'Brien, and C. Barton. Single-program speculative multithreading (spsm) architecture: Compipler assisted fine-grained multithreading. In *PACT-95*, 1995.

[4] N. Jouppi and D. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *ASPLOS-III*, 1989.

[5] M. Lam and R. Wilson. Limits of control-flow on parallelism. In *ISCA-19*, 1992.

[6] M. Lipasti, C. Wilkerson, and J. Shen. Value locality and local value prediction. In *ASPLOS-VII*, 1996.

[7] M. Postiff, D. Greene, G. Tyson, and T. Mudge. The limits of instruction level parallelism in spec95 applications. In *INTERACT-3 at ASPLOS-VIII*, 1998.

[8] A. Sodani and G. Sohi. Dynamic instruction reuse. In *ISCA-24*, 1997.

[9] G. Tyson and T. Austin. Memory renaming: Fast, early and accurate processing of memory communication. *IJPP*, 1999.

[10] David Wall. Limits of instruction-level parallelism. Technical Report DEC WRL 93.6, Compaq Corp., 1993.