Limits and Graph Structure of Available Instruction-Level Parallelism

Darko Stefanović and Margaret Martonosi

Princeton University, Princeton NJ 08544, USA

Abstract. We reexamine the limits of parallelism available in programs, using run-time reconstruction of program data-flow graphs. While limits of parallelism have been examined in the context of superscalar and VLIW machines, we also wish to study the causes of observed parallelism by examining the structure of the reconstructed data-flow graph. One aspect of structure analysis that we focus on is the isolation of instructions involved only in address calculations. We examine how address calculations present in RISC instruction streams generated by optimizing compilers affect the shape of the data-flow graph and often significantly reduce available parallelism.

1 Background and Related Work

Most studies of the limits of available instruction-level parallelism have focused on the timing of an optimal schedule of the instruction sequence for an idealized processor model. We propose to examine directly the data flow graph of the instruction sequence. Thus we will be able to gain insight into the structural properties of the available parallelism, so that we may understand which elements of the instruction sequence, or which compiler idioms, affect available parallelism. In particular, here we show that the presence of address calculations for memory operations greatly affects parallelism; in some programs, it is precisely the address calculations that limit the asymptotically achievable parallelism. As in earlier studies, we assume no hardware limitations: the degree of parallelism available is the degree exploitable. Examining very long dynamic code sequences means that control flow is entirely revealed and does not constrain parallelism. Imperfect alias analysis in compilers [1, 2] sequentializes code by enforcing the order of the store-load pair, together with all potentially aliased memory operations, for any value that cannot be held in registers and is temporarily stored in memory (register spill, call save, or otherwise); by precise memory disambiguation at run-time we remove all such constraints as well.

A number of studies over the past three decades have looked at the limits of parallelism [1–10], using instruction-scheduling simulators. The simulator reports the number of cycles needed to execute the program, and the number of instructions executed. The ratio of the two gives the IPC as the standard measure of instruction-level parallelism [11]. The simulator effectively constructs the moving "front line" of the data-flow graph [3]; thus, constructing an entire data-flow graph is not necessary to obtain a single number, the cycle count. However, having an *explicitly* constructed graph permits us to study its structure: we can inspect the computation nodes repeatedly, and evaluate the graph using *multi-pass* and *backward-flow* algorithms. We will illustrate this new possibility on one example: we will recognize instructions involved in address calculations using a backward-flow algorithm.

While in the past reconstructing large graphs was dismissed as impractical [3], that is no longer the case. Currently available memory space permits building graphs sufficiently large to capture interesting application behavior—parallelism analysis using a conceptual dependence graph of a moving window of program execution was demonstrated by Austin and Sohi [1]. Recently, Ebcioğlu *et al.* described a system for dynamic code translation and optimization [12], aimed at transparent porting of applications to a VLIW execution engine. Among other results, they evaluate achieved parallelism without resource constraints, and with store-load bypassing. We obtain comparable parallelism numbers, except for their results with the "combining" optimization, which in some cases show much higher parallelism. This optimization breaks dependence chains of immediate-operand instructions with the dependence on a common register, by adjusting the immediate values (a form of constant folding at the machine level); code modifications are outside the scope of our study.

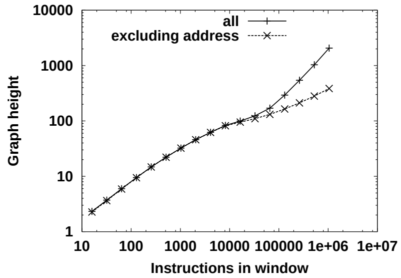
2 Run-time Analysis of Programs

Our analysis uses the core of the SimpleScalar architectural simulation toolset [13] for the Alpha instruction set, and dynamically constructs a program's data-flow graph. Conceptually, graph nodes correspond to executed instructions, while graph edges correspond to computed operand values. The values are tracked through memory, including multi-byte values through partial and unaligned accesses. This allows us to recognize when entire stored values are reloaded. Nodes are not created for instructions identifiable as data transport: register moves and memory loads; instead, the values are appropriately bypassed from the producing node to the using node. Thus the data flow of the computation is reconstructed independent of the storage layout.

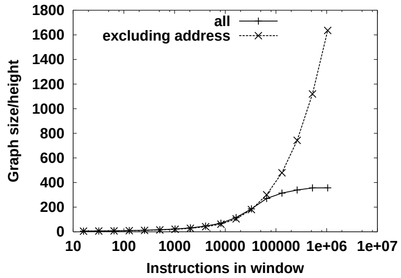
We simulated a number of SPEC95 and Mediabench programs, with up to 1800 million instructions executed. Benchmarks were compiled on a Digital Alpha 21164 EV56 using native C and Fortran compilers, and highly optimized as specified by SPEC. For each benchmark, we varied the size of the instruction window as powers of 2, between 16 and 1M (limited by the memory capacity of the simulator host).

We first look at the parallelism reported for the graphs consisting of all instructions in the examined window; the results are presented in plots (a) and (b) in Figures 1 and 2. The solid lines, labelled *all* in the graph height plots (a), show the growth of average graph height (length of critical path) with increasing instruction window size. The axes in graphs (a) are both logarithmic; the slopes of the curves (below 1) show that the dependence is sublinear. The solid lines, labelled *all* in the graph parallelism plots (b), show the ratio of graph size (number of instruction nodes) to height. This ratio is a measure of average available parallelism, because it reflects the potential speedup of a machine with unbounded hardware resources (limited only by data dependences) over a sequential machine that executes exactly one instruction per cycle in program order. As the instruction window size increases, so does the parallelism. However, we note some distinct behaviors. In *145.fpppp*, the parallelism saturates quickly: with an instruction window size of 128K, it is 314, with 1M, it is 357. Not so in *110.applu*: parallelism grows smoothly (but sublinearly) even as very large window sizes are reached. The *absolute* values of parallelism are vastly different: whereas *145.fpppp* achieves over 300, and *110.applu* over 1000, we have only 45 for

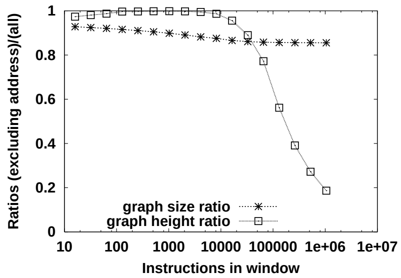
129.compress (not shown). This agrees with observations [3] that some numerical programs have very high intrinsic parallelism, proportional to problem size and exposed by unrolling loops (which we in effect do).



(a) Graph height

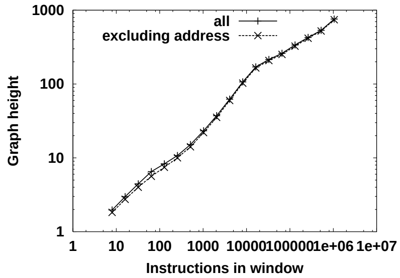


(b) Graph parallelism measure

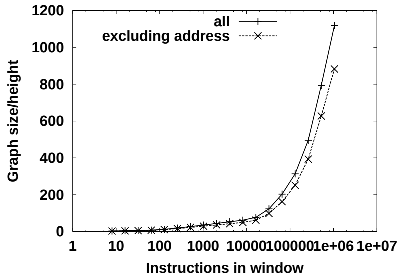


(c) Ratios excluding address calculation

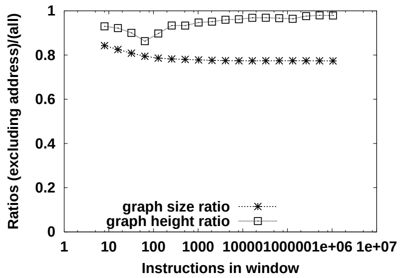
Fig. 1. Benchmark *145.fpppp*



(a) Graph height



(b) Graph parallelism measure



(c) Ratios excluding address calculation

Fig. 2. Benchmark *110.applu*

Excluding Address Calculations. Will there be differences with respect to available parallelism between the data-flow graph as built, and its subgraph that excludes purely address calculations? This is an interesting question, because the latter graph seems closer to the algorithmic intent of the program, address calculations being partly an artifact of the particular compiler/RISC architecture realization of the program. Recall that while we are reconstructing the data-flow graph at run-time, we are able to recognize when a load instruction L retrieves a value written to memory by a previous store instruction S and produced by a previous computational instruction C. We bypass such a load—an instruction that uses the loaded value sees it instead as coming from C, similar to the *load-store telescoping* optimization [12]. Note that L is no longer needed to represent the computation, and in some cases S also is no longer needed (if L is the only load of the value). Loads and stores are preceded by instructions to calculate an address. (These instructions may in turn include other loads.) If certain loads and stores are no longer needed to represent the computation, then the corresponding address calculations are not needed either. However, while we are building the graph we cannot know which computations will end up being used *only* to calculate addresses. This we determine in a separate, backward-propagating pass over the data-flow graph. (Address calculation recognition subsumes the *stack pointer register analysis* of [10].) The dashed lines, labelled *excluding address* in plots (a) and (b), give the graph height and graph parallelism measure for the data-flow subgraph without address calculations. Plots (c) show the relative size and height of the subgraph with respect to the full graph. We show both in the same plot area to make it easier to compare with the graph parallelism measure plot. (Consider the intersections of (c) curves and the intersections of (b) curves: their abscissæ coincide.) Let us first look at the relative subgraph size, labelled "graph size ratio" in plots (c). This ratio changes very little with instruction window size, and the small observed change is in the direction of somewhat smaller ratios as the window size is increased. Indeed, in the backward-propagating algorithm we must conservatively assume that values present at the end of the instruction window *may* be used as non-addresses in the continuation of the program after the window; as the window grows, the inaccuracy of that assumption diminishes and with it the number of instructions inaccurately assumed to be involved in non-address computation. The ratio varies greatly across benchmarks: 0.9 for *145.fpppp*, 0.8 for *110.applu* and *124.m88ksim*, but just 0.2 for *129.compress*.

Relative subgraph height, labelled "graph height ratio" in plots (c), shows significant variation with window size. In *145.fpppp* it remains close to 1 up to a window size of 16K, but drops sharply thereafter, so that by 1M it is just 0.2. In other words, for smaller windows, the subgraph height is about the same as the full graph height, but for larger windows, the subgraph height collapses. The critical path is determined by a dependence chain of address calculations carried in a loop. If address calculations are eliminated, a much larger amount of parallelism is exposed. We observed the same pattern in *141.apsi*, *099.go*, *134.perl*, *126.gcc*, *130.li*, and *mpeg2decode*. On the other hand, in *110.applu* the ratio of graph heights is close to 1: the critical path is for the most part determined by the "data" calculations, i.e., instructions other than address calculations. We observed a similar pattern in *146.wave5*, *124.m88ksim*, and *adpcm*.

We may summarize the findings as follows: When address calculations form long dependence chains, they can dominate "data" computations, and their removal is beneficial for parallelism. When address calculations are localized, their removal does not affect graph height, yet it reduces graph size; therefore, parallelism is reduced.

3 Future Directions

With data-flow graphs explicitly constructed we are not restricted to critical paths through the entire graph, but can zoom in on particular nodes. For instance, we can examine the critical path of the computation that produces the address for a load (with a view to prefetching), or the critical path that produces a conditional value (with a view to scheduling beyond the corresponding branch). We should consider what can be done in language implementation to reform the way memory data are accessed: a compiler optimization such as array index "strength reduction" can introduce a chain of address calculations where none is apparent at the source level. On the other hand, to appreciate the *practical* repercussions of available parallelism, we should consider code mappings to realistic processors, where memory bandwidth and control flow uncertainty are taken into account. We intend to combine the analysis of instruction-level parallelism with analysis of bit usage [14], which will lead to a finer-granularity description of parallelism as the basis for code mapping decisions for hybrid fixed-configurable processors.

References

1. T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *19th ISCA*, pages 342–351, May 1992.
2. J. W. Davidson and S. Jinturkar. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. In *MICRO-28*, Dec. 1995.
3. A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Trans. Comput.*, C-33(11):968–976, Nov. 1984.
4. D. W. Wall. Limits of instruction-level parallelism. WRL Research Report 93/6, Digital Equipment Corporation, Western Research Laboratory, Palo Alto, CA, Nov. 1993.
5. C. C. Foster and E. M. Riseman. Percolation of code to enhance parallel dispatching and execution. *IEEE Trans. Comput.*, C-21(12):1411–1415, Dec. 1972.
6. N. P. Jouppi. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Trans. Comput.*, 38(12):1645–1658, Dec. 1989.
7. M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple instruction issue. In *ASPLOS III*, pages 290–302, Boston, Massachusetts, 1989.
8. M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *18th ISCA*, pages 276–286, May 1991.
9. M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *19th ISCA*, pages 46–57, May 1992.
10. M. A. Postiff, D. A. Greene, G. S. Tyson, and T. N. Mudge. The limits of instructions level parallelism in SPEC95 applications. In *3rd Workshop on Interaction Between Compilers and Computer Architecture*, Oct. 1998.
11. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, Inc., San Mateo, California, 1996. Second Edition.
12. K. Ebcioğlu, E. R. Altman, S. Sathaye, and M. Gschwind. Optimizations and oracle parallelism with dynamic translation. In *MICRO-32*, Nov. 1999.
13. D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, pages 13–25, June 1997.
14. D. Stefanović and M. Martonosi. On availability of bit-narrow operations in general-purpose applications. In *10th FPL*, Villach, Austria, 2000.