Run-time versus Compile-time Instruction Scheduling in Superscalar (RISC) Processors: Performance and Tradeoffs

Allen Leung[1], **Krishna V. Palem**[2], **Cristian Ungureanu**[3]

Abstract

The RISC revolution has spurred the development of processors with increasing levels of *instruction level parallelism (ILP)*. In order to realize the full potential of these processors, multiple instructions must be issued and executed in a single cycle. Consequently, *instruction scheduling* plays a crucial role as an optimization in this context. While early attempts at instruction scheduling were limited to compile-time approaches, the recent trend is to provide *dynamic* support in hardware. In this paper, we present the results of a detailed comparative study of the performance advantages to be derived by the spectrum of instruction scheduling approaches: from limited basic-block schedulers in the compiler, to novel and aggressive run-time schedulers in hardware. A significant portion of our experimental study via simulations, is devoted to understanding the performance advantages of run-time scheduling. Our results indicate it to be effective in extracting the ILP inherent to the program trace being scheduled, over a wide range of machine and program parameters. Furthermore, we also show that this effectiveness can be further *enhanced* by a simple basic-block scheduler in the compiler, *which optimizes for the presence of the run-time scheduler in the target*; current basic-block schedulers are not designed to take advantage of this feature. We demonstrate this fact by presenting a novel enhanced basic-block scheduler in this paper. Finally, we outline a simple analytical characterization of the performance advantage, that run-time schedulers have to offer.

Key words: *Compile-time Optimizations, Dynamic Schedulers, Instruction Scheduling, Program Traces, Scope, Superscalar Processors*

[1]Permanent address: Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012-1185; +1 (212) 998–3518; leunga@cs.nyu.edu

[2]Permanent address: Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012-1185; +1 (212) 998–3512; palem@cs.nyu.edu

[3]Permanent address: Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012-1185; +1 (212) 998–3525; ungurean@cs.nyu.edu

1 Introduction

The trend in the design of modern RISC processors [10] is to have an increasing amount of ILP provided within a single processor with *multiple pipelined* functional units. The expected performance gains of these machines are fully realized by a variety of optimizations, with *instruction scheduling* (or scheduling for short) playing a significant role. Consequently, a lot of effort has been invested in exploring and improving the opportunities of these schedulers to take advantage of RISC processors with ILP. In the early RISC processors, scheduling was typically done at compile-time and statically produced a sequence of instructions with very good run-time behavior. As the degree of parallelism increased, these approaches to instruction scheduling have ranged from the early *local scheduling* approaches [29, 40] where the code-motion is limited to within a *basic-block* boundary [2], to more ambitious (and expensive) *global* approaches [4, 9, 26, 7] which move instructions across basic-blocks[4].

Typically, the data- and control-dependence relationships in the program limit the number of instructions that can be scheduled concurrently on the target processor. Consequently, compile-time approaches such as those identified above — limited by the amount of information available statically — make safe but limiting decisions and cannot be as aggressive as approaches that have access to information available only at run-time. For example, the outcome of conditional branches will affect the choice of instructions to be executed next, and hence those that ought to be scheduled with higher priority.

Consequently, as superscalar processor designs evolved, a significant trend has been to support instruction scheduling directly in hardware [15, 14, 8]; we will refer to this as *dynamic scheduling*. Informally, the run-time hardware has a "window" of instructions that it looks into, *ahead* of the current instruction to be executed. Using hardware support for data- and control-dependence analysis, "ready" instructions are moved up out-of-order in the instruction stream, to keep the functional unit cycles fully utilized. (Potential under-utilization of available functional units could be due to a lack of instruction availability or due to pipeline interlocks in an individual functional unit.)

In this paper, we undertake a systematic investigation of the performance gains to be expected by adopting static-compile time as well as hardware supported run-time scheduling. The measures that we will use to evaluate the merits of the above approaches to instruction scheduling are the *absolute* and *relative speedups* as well as the *normalized throughput*; we will return to discussing these measures below, in greater detail. In making these comparisons, we are motivated by two concerns.

First, our goal is to understand the benefit that (hardware supported) run-time scheduling yields, as a function of various significant architectural and program parameters. Our results establish the benefits of run-time scheduling in the affirmative for ranges of variations of these parameters. Specifically, with enough ILP in the source program, run-time schedulers approach the peak performance benefits afforded by compile time *global scheduling* — TRACE scheduling [9] is our canonical global scheduling scheme used here — even when the compile-time approach is assumed to be completely clairvoyant in that it precisely knows the behavior of branches ahead of time. Unless otherwise stated, when we refer to a global scheduler in the sequel, we will imply one with such clairvoyance. The rationale behind bestowing global scheduling with this extra power is that we wish to consider the limits of its ability when comparing it against the merits of run-time schedulers[5]. In a sense, we are using global scheduling as stated above, to be the *limit* on available ILP in the source program.

Second, our goal is to understand the degree to which existing compile-time scheduling approaches can be enhanced to assist the run-time hardware and help improve its performance. Here, our approach is to only consider improving local scheduling approaches, at compile-time. We will now outline the rationale in limiting our improvements to this setting. Primarily, as we establish in this paper, the presence of run-time schedulers supported by compile-time local scheduling yields performances that are extremely competitive with global scheduling in the case of processors with ILP ≤ 10. Furthermore, re-engineering a compiler to include some form of global scheduling — in addition to being in the region of diminishing returns for speedup or

[4]Informally, basic-blocks are parts of the program which do *not* have any conditional branches, such as the body of the inner-most loop of a loop-nest.

[5]In keeping with this spirit, we will also overlook certain side-effects of global scheduling, to be discussed in Section 5.2.

throughput as described below — is prohibitive, and it remains a technically challenging research question as well [26, 18, 27, 33]. We will describe some of the hurdles to effective global scheduling in sections 2.1 and 5.2

The key architectural parameters of interest to us are the number and types of functional units in the target processor, their degree of pipelining, and most significantly, the *scope* [6] of the lookahead hardware used by the run-time scheduler. The scope is the number of instructions that the hardware can analyze and attempts to schedule dynamically. Our experiments ranged over a total of 1344 parametric variations defining the spectrum of superscalar processors. In addition to machine parameters, the program parameters that we will vary are the types of inter-instructional dependences imposed by data- and control-flow, and the sizes of the basic-blocks.

In evaluating and projecting the relative performances of the different scheduling methods, we have a choice of: (*i*). analytical modeling (*ii*). simulation and (*iii*). hardware prototyping. The above sequence of approaches incurs increasing cost and effort in determining the trends, with a corresponding improvement in the accuracy of the predictions. Given the large range of configurations that we wish to explore, and the complicated nature of its constituent parameters, we have primarily chosen approach (*ii*); we simulate the run-time behavior of the hardware. We also have some analytical modeling that demonstrates the relative trends in the speedups of the scheduling approaches over the entire range of configurations. These analytical trends are sketched in Section 8. Clearly, the insight gained through the above approaches is an important prelude to embarking on an eventual prototyping effort, which — due to economic constraints — is typically limited to a narrow range of configurations.

1.1 Significant Results

We will now summarize the significant contributions of our paper. All our trends are derived from averaging over 30 program traces generated by us synthetically to exercise the key program parameters better, for each configuration; details will be provided in Section 3.

1. We show that even with a modest scope of about 15, a hardware scheduler can get almost 90% of the performance gain achieved by global scheduling. This trend improves with increasing scope, and basic-block size of the original program, approaching 100% very quickly,

2. We introduce an *enhanced basic-block scheduling* algorithm to which the run-time scheduler is visible and optimizes to take advantage of its behavior. (Current basic-block schedulers work with an idealization of a target processor without any run-time support.) To the best of our knowledge, this is the first instance of such an algorithm.

   Given that compilers for modern superscalar RISC processors invariably involve a basic-block scheduler, this algorithm can be thought of as a "no-cost upgrade" to an existing compiler, to squeeze extra performance out of the run-time hardware.

3. When used to optimize code for a machine with run-time support, the above scheduling scheme produced between $30 - 50\%$ of the performance improvement that global scheduling could achieve.

   We caution the reader that the room for improvement for compile-time schedulers for traces executed on machines with dynamic schedulers was quite small, and was in the range of $1 - 6\%$ for the program traces that we experimented with.

4. The trends in the normalized throughput are comparable to those in the speedup except for quantitative differences to be discussed in Section 3.

5. All of our experiments were performed in the context of processors that compact the instruction stream as well as in the context of those that do not. In the former case, instructions that have been scheduled dynamically out-of-order are no longer visible to the processor, whereas in the latter case, they stay in the instruction stream but are "tagged" so that they do not get re-executed. The former phenomenon

is referred to as *compaction* and occurs increasingly often in the newer generation of processors with instruction caches.

The overall trends are consistently the same across both architectures, except that compaction allows dynamic schedulers to approach the performance of global scheduling much faster. There are interesting tradeoffs between program parameters, the size of the scope and the availability of compaction, which we will discuss in Section 3.3.2.

Finally, our experiments confirm that increasing the size of the basic blocks shows us the increasing effectiveness of local scheduling methods applied at compile-time.

1.2 Related Work

A lot of work has gone into evaluating the effectiveness of instruction scheduling performed at compile-time, in isolation [4], [3], [9] [11], [23], [26], [29] [30], [33]. Similarly, a lot of interest has gone into the study of available ILP in superscalar RISC processors in isolation, *independent* of its interaction with instruction scheduling done at compile-time; please see [20], [6], [37], [36], [38] for further details. There are also some studies that study the combined effect of having optimizations at both levels by lumping the two together; [7, 27] are typical examples. To the best of our knowledge, we are the first to explicitly study the relative merits of instruction scheduling at each of these levels, with emphasis on the interactions and tradeoffs involved. For example, our study helps clarify the value of adding various types of scheduling techniques at compile-time given that run-time support already exists in current hardware, and helps delineate the regions of diminishing returns in performance, among these interacting domains.

2 Experiment overview

In order to present an overview of the main trends of our experimental results, we will briefly sketch the various scheduling frameworks that we intend to consider below and present more detailed descriptions in Section 4.

2.1 Instruction Scheduling

2.1.1 Compile-time Global Scheduling

In the context of scheduling at compile-time, the most ambitious schemes that we consider are global approaches that attempt to move code *across* basic-block boundaries. In this context, we choose trace scheduling and its derivatives as representatives of global scheduling methodologies [9, 11]. Global scheduling and related issues are discussed in Section 5 in greater detail.

Traces are loop-free[6] linear sections of code, which can span several basic blocks. They are derived from programs with known branch frequencies; branch-frequencies are derived via profiling the program's execution [32, 39]. Informally, given these frequencies, a trace is derived by "walking" along the directions of control flow so that whenever a branch instruction is encountered, we take the branch with higher frequency. An example is shown in Figure 1 which denotes a *control-flow graph* [2] of a sample program. The boxes denote basic-blocks (to be discussed in detail below in Section 4) with branch frequency information shown along the edges. For example, block $B1$ can go to $B2$ with a probability of 0.55 whereas it can go to block $B9$ with a probability of 0.45. The trace that will be fed to the global scheduler is $\{B1, B2, B4, B6, B7, B8\}$ in this case.

At a high-level, the scheduling algorithms that we will consider follow the canonical approach to instruction scheduling and can be described using the following framework:

[6] Compile-time instructions scheduling deals with program segments with loops in a very different way [23, 33], than regions of the program which are loop-free.

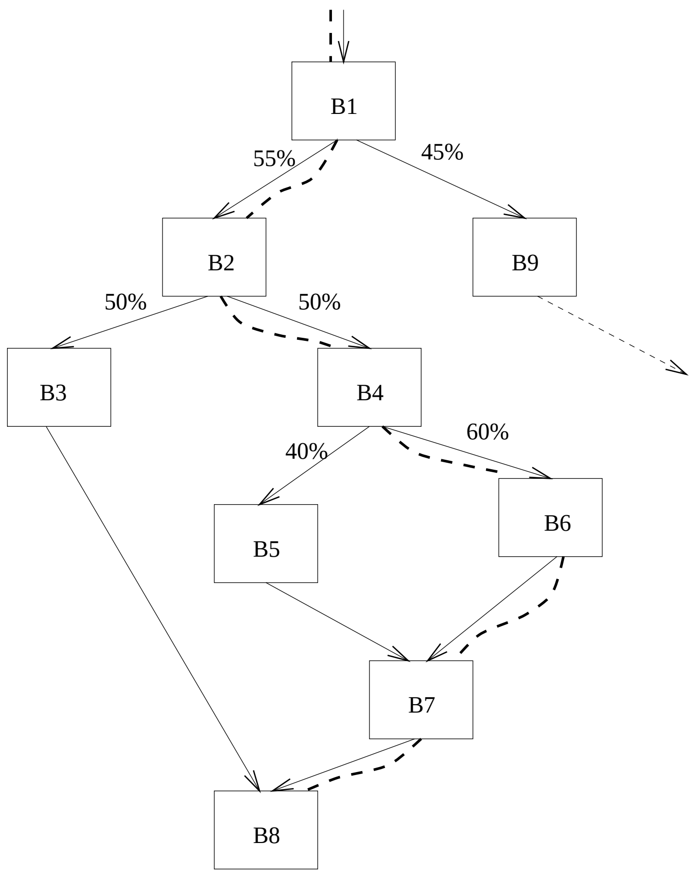

Figure 1: A trace in the program

1. Prioritize the instructions on the trace by assigning *ranks* to them.

   A variety of priority or rank functions have been defined in literature [3, 9, 29, 30] all of which aim to help improve the functional unit utilization and minimize the execution times of the programs, as they run on the target machine. Throughout our experiments, we will use the rank algorithm from [29].

2. Build a sorted list L with nodes of higher priorities occurring earlier on in this list.

3. *List schedule* the instructions so that:

   - subject to availability of functional units, and to existing data- (and control-)dependences, as many instructions as possible are greedily chosen and scheduled to start execution on the next cycle, in the order specified by L.

4. Take care of the side-effects caused by "speculative" and "replicative" moves as instructions are moved beyond basic-block boundaries; details can be found in Section 5.2.

In all of our experiments, we consider only steps $1-3$ above and ignore the side-effects mentioned in Step 4, caused by the global scheduler. In essence, our global schedulers are side-effect free for purposes of comparison in this paper. This is in keeping with the spirit of our stated goal of allowing the greatest possible flexibility to compile-time optimization, and trying to compare its performance with its run-time counterpart. As our results indicate, run-time scheduling proves to be very competitive *despite* the compiler having the above-mentioned flexibility. For convenience, we will refer to our global scheduling methods as G.

2.1.2 Compile-time Local Scheduling of Basic-blocks

In this case, all instruction movement can only be done within the basic-blocks and hence code motion is limited. However, this form of instruction scheduling is less expensive to perform at compile-time. Consequently, most early and current compilers incorporated this phase in some fashion [29, 40]. As stated in the introduction, we will consider these "local" scheduling schemes, henceforth denoted by L, that are oblivious of the fact that hardware run-time scheduling exists. However, if local scheduling schemes were to be sensitive to the presence of lookahead hardware, they could potentially optimize further for its presence. We will use E to denote such enhanced schemes that are local *but optimize for the presence of run-time scheduling support.* Unless otherwise stated, both these schemes will be used in conjunction with run-time scheduling in the target processor. As our results indicate, the performance of highly efficient local scheduling L can be improved upon by using its enhanced variant E to produce better schedules on the average. For completeness, in our experiments, we will also consider the effectiveness of compile-time local scheduling without hardware support for run-time scheduling; we will refer to this setting as V.

2.1.3 Run-time Scheduling in Hardware

Our canonical superscalar RISC processor has multiple pipelined functional units including possible support for branch prediction. The number of functional units as well as the depth of pipelining are parameterized. Furthermore, each functional unit can be of a fixed type such as a fixed point unit or a floating point unit. We will return to a detailed discussion of these variations and models in Section 4.2.

Modern processors explore the sequence of instructions dynamically at run-time, to determine if they can be moved ahead to fill "idle-cycles" in the pipelines. As stated before, the amount by which they look ahead is referred to as the scope. As the run-time hardware scheduler moves instructions out-of-order up the instruction stream, those instructions that have been scheduled ahead of their position can either remain in the stream and not get rescheduled, or be removed from the current snapshot of the stream as seen by the processor. We will consider superscalar RISC processors with and without compaction in our experiments. For convenience, each combination of the above parameters with the exception of the scope will be referred to as a *processor configuration* (or configuration for short) — an example configuration is one in which we have two floating point units, two fixed point units[7] all of which are 3-stage pipelines — and our experiments will involve varying the scope for each configuration.

2.2 Program Structure

The experiments will also involve varying the program parameters such as the size of the basic blocks and the mix of instruction in addition to the machine configurations mentioned above. We consider the following settings for the program structures. Recall that a significant part of our goal is to understand the impact of varying configurations on the *relative performances* of static and dynamic scheduling approaches. Therefore, the first part of our experiments are based on generating a substantially large — over $50,000$ test cases — synthetically generated program structures, each with about 200 instructions. A detailed description of the process used to generate these graphs is described in Section 7. Also, we will return to a discussion of the specific parametric variations involved in generating these program graphs, Section 3.3.

3 Overview of Experimental Results

In order to compare the effectiveness of the various instruction scheduling approaches of interest, we will consider the performance improvements to the running times of the various test programs, referred to as speedups, as well as improvements of the normalized throughput of the functional units.

[7] The branch and load/store units are always included implicitly.

3.1 Measures of Comparison

3.2 Speedup

Given a program trace and a fixed processor configuration let T_A be the running time of the trace, scheduled using method M_A. The running time is measured in number of cycles, that an execution takes on the machine configuration being considered. We are interested in determining the following *relative* and *absolute* speedups through our experiments:

1. The first measure or relative speedup of interest is $R_{L,G}$ relative to V which is

$$R_{L,G} = \frac{T_L - T_V}{T_G - T_V}$$

   This ratio compares the relative advantage that run-time scheduling in hardware has to offer, when compared to global scheduling — recall that global scheduling is designed to represent the limit to performance improvements that can be achieved. In this measure, the run-time scheduler is assisted by a conventional basic-block scheduler L, which is typically a part of modern optimizing compilers for superscalar processors [28]. Both of these are compared relative to basic-block scheduling in the context of a processor with no hardware support for scheduling, i.e., the method V.

2. The next measure is $R_{E,G}$ relative to V which is

$$R_{E,G} = \frac{T_E - T_V}{T_G - T_V}$$

   This measure compares enhanced local scheduling in the presence of hardware lookahead E, to global scheduling; again, the comparison is relative to basic-block scheduling on a processor with no hardware support V. Specifically, we will consider a basic-block scheduler that uses the algorithm that we introduce in Section 6. This comparison gives us an idea of the relative power of hardware with assistance from the basic-block scheduler, over global scheduling.

3. The third and final measure of comparison is $R_{E,G}$ relative to L,

$$R_{E,G} = \frac{T_E - T_L}{T_G - T_L}$$

   Here, we compare the relative advantage of enhancing the basic block scheduler. Specifically, we compare the improvement in performance that an enhanced basic-block scheduler gives us over global scheduling. Both these measures are compared with basic-block schedulers L that are currently used in compilers — those that do not account for the presence of hardware lookahead.

The above ratios were referred to as relative speedups since one method was compared to another, relative to a third approach acting as a frame of reference. For example, in defining $R_{L,G}$, with respect to (method) V, T_G is compared to T_L, with T_V as the reference. We will also consider *absolute speedups*, by which two methods A and B are compared directly.

1.

$$S_{E,G} = \frac{T_G}{T_E}$$

   This is particularly interesting since by looking at the data, we can easily deduce the relative performance of dynamic scheduling in hardware, enhanced by basic-block scheduling adapted to take advantage of this feature, when compared to completely global static scheduling methods.

2.

$$S_{E,L} = \frac{T_L}{T_E}$$

This ratio gives the percentage of speedup that the enhanced local scheduling method E yields, relative to conventional basic-block scheduling L.

3.2.1 Throughput

We also compare the effect of the various scheduling schemes on how well the machine resources are being utilized. The measure that we use to characterize this factor is the *normalized throughput* (or *throughput* for short).

For ease of comparison across configurations, the throughput is normalized by the number of functional units. The formula for computing the throughput for some scheduling method A is

$$NT_A = \frac{N}{mT_A},$$

where N is the number of instructions in the trace, m the number of functional units, and T_A the completion time of the instruction stream and is expressed in the following units:

$$\frac{instructions}{functional \quad unit - cycle}$$

Normally, we would expect improvements in throughput to be positively correlated with those in speedup, and our results confirm this trend.

3.3 Significant Experimental Trends

We will now present our experimental results derived from running the above-mentioned scheduling approaches E, G, L and V, on a variety of typical program structures. In deriving these programs, our primary motivation is to be able to isolate the effect of increasing scope on performance, as the processor configurations varied. Therefore, we generated a large family of program traces randomly, while simulating the typical effects of locality of usage, basic-block size variations as well as mixes of instruction types including loads and store and branches in each trace. More precisely, each program trace that we generated has about 200 instructions with about $10 - 20\%$ load and store instructions. The generated program traces had average basic-block sizes ranging from 5 instructions per-basic-block at the low end, to about 40 at the high end.

All our simulations were done for a wide range of idealized machine configurations, where the processor always included one dedicated functional unit for loads and stores, and one for branch prediction. Additionally, it had a range of between 2 and 10 functional units for floating/fixed point operations. We considered configurations where these two types of functional units were interchangeable, as well as those where they had to be dedicated. Each of these functional units for computing floating and fixed point operations can be pipelined, with depths ranging between 2 to 8 stages; all the units have an equal number of stages. This yields an experimental base of about $5,000$ machine configurations overall.

For these machine configurations, we varied the scope between 5 and 40 instructions and determined the speedup and throughput measures as follows. We generated and tested the performance of 30 distinct program traces, and averaged the resulting running times to derive the final measures of interest.

3.3.1 Speedup Ratios

We considered the relative and absolute speedups in the order in which we had described them above in Section 3.2. We remind the reader that our primary goal is to try and determine the impact of increasing scope as the machine configurations vary. Therefore, in order to extract the significant trends, rather than plotting the actual data points, we take the linear regression of the speedup figures in question, and plot the results for increasing values of scope. (Note that we have plotted all the data points[8] in the Figure 3 below to highlight the difficulty in displaying all the information; as can be seen there, the result is quite hard to evaluate. We intend to increase the sizes of the figures and display all the data, in an expanded full-version that is planned for journal submission.) In all of the figures shown below, we plot the trends parameterized by the basic block sizes and denoted by the letter b. For convenience, all the ratios are scaled by a multiplicative factor of 100 in these figures, resulting in the percentage of relative speedup.

Relative Speedup $R_{L,G}$ with respect to V

The relative speedup $R_{L,G}$ is shown in Figures 2 and 3 below. In the interests of space, we restrict our presentation in this section to the extreme values of the numbers of functional units (m), as well as pipeline depths (k) in these figures. The goal is to be able to understand the effect of scope, as the machines become increasingly parallel along these two dimensions of ILP.

The first and most obvious trend is that run-time scheduling is able to achieve between $40 - 100\%$ of the performance of global scheduling and as would be expected, the gains in hardware improve with scope. Typically, for basic-blocks of size 40 or less, the hardware scheduler is very competitive with global scheduling; recall that a conventional basic-block scheduling method L is assisting the run-time scheduler in this comparison. Furthermore, the trends indicate that the gains achieved by the hardware scheduler are somewhat less pronounced with shallow pipelines ($k = 2$ as shown in Figure 2) than with deeper pipelines ($k = 8$ shown in Figure 3).

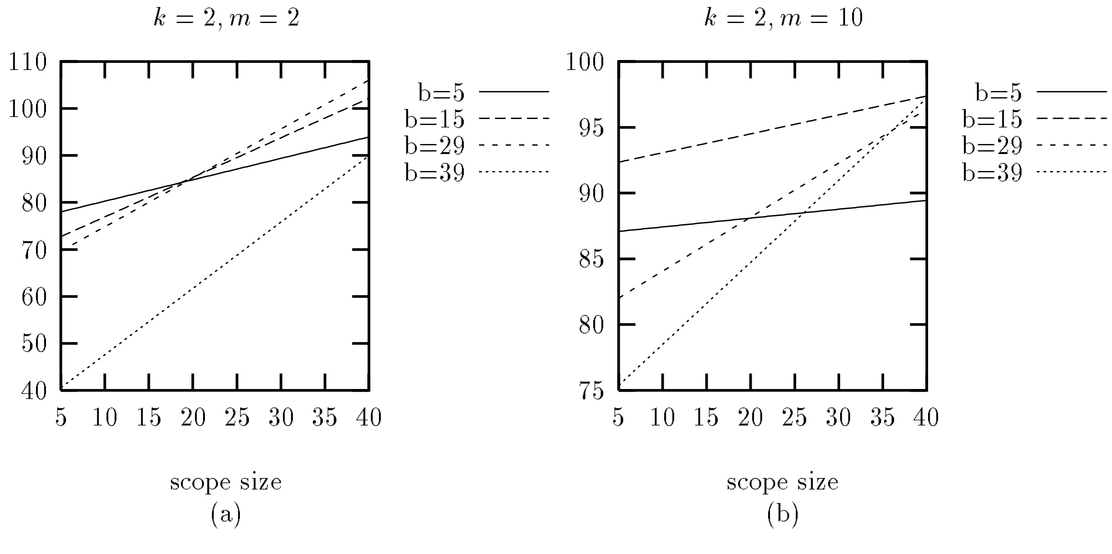

Figure 2: Percentage of relative speedup of run-time scheduling $R_{L,G}$ with respect a machine with no-hardware scheduling for pipelines of depth 2

Relative Speedup $R_{E,G}$ with respect to V

For completeness we also made the same comparisons as above with the basic-block scheduler being replaced by an enhanced basic-block scheduler. As shown in Figures 4 and 5 below, the trends are identical with the exception that the enhanced basic block scheduler helps the run-time hardware "catch-up" with global scheduling somewhat more rapidly. (We will consider this improvement more directly, when we discuss the absolute speedup ratios below.) To see this, we only need to make a comparison with the corresponding trends in Figures 2 and 3 respectively. For example, the data in Figure 2 (a) corresponds to that in Figure 4 (a) — both experiments are conducted using the same input data on identical processor configurations

[8]Each data point represents the average value from 30 samples.

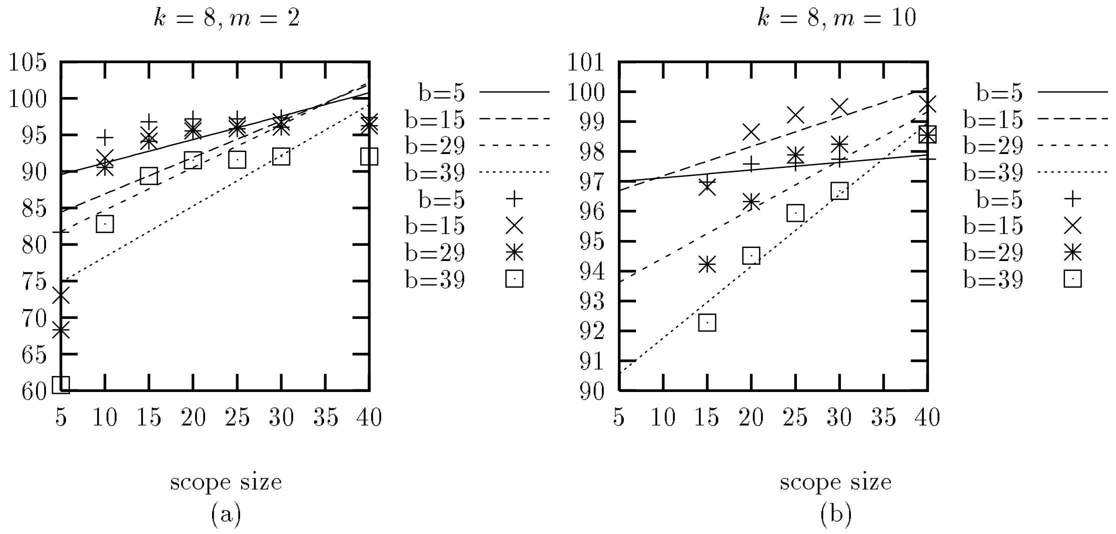

Figure 3: Percentage of relative speedup of hardware $R_{L,G}$ for pipelines of depth 8

— and each of the trends in the former figure can be directly compared to that in the latter. The relative speedups start at slightly higher values in the current case and reach 100% relative speedup more quickly than $R_{L,G}$ with respect to V.

Relative Speedup $R_{E,G}$ with respect to L

Here, our goal is to understand the value of enhancing the basic-block scheduler to take advantage of the dynamic scheduling at run-time. Once again, we use global scheduling as the envelope or limit to ILP, that can be extracted from the source program.

Note that the numerator in our ratio $R_{E,G}$ is the number of cycles that enhanced basic-block scheduling gains, over global scheduling. As the trends indicate, for small k and m (Figure 6 (a)), enhanced basic-block scheduling *in conjunction with the run-time scheduler* achieves between $25 - 90\%$ of the performance of method G; note that the denominator quantifies the performance advantage of global scheduling over basic-block scheduling. Also, note in the same figure that this performance improvement is much more pronounced for larger basic-block sizes since the enhanced scheduler has more room to optimize.

A particularly interesting trend is that with increasing ILP, the relative advantage of enhanced basic-block scheduling actually drops (Figure 6 (b)) as the scope goes up. The reason for this is that with increasing ILP, the run-time hardware can perform quite well independent of any assistance from the compile-time scheduler. To understand this better, let us reconsider the essential advantage of compile-time scheduling: to prioritize instructions so that *in the presence* of contention for (critical) functional-units, instructions that expose greater ILP are given priority. This advantage is less significant as the contention diminishes due to greater ILP in the current configuration. For example, let consider the case of a configuration with an unlimited number of functional units. In this case, any run-time scheduler *with unlimited scope* — note that our drop in performance is with increasing scope and hence is tending towards the "unlimited" case — will do equally well independent of the compile-time support, since all available instructions are immediately scheduled on each cycle.

Absolute Speedup $S_{E,G}$ with respect to V

This ratio is particularly interesting since it gives us a very direct measure of the rate at which the hardware scheduler, assisted by enhanced basic-block scheduling, can approach the limit case of global scheduling. The Figures 7 and 8 show this speedup for the four extreme points in the space of configurations of interest to us.

As can be seen there, enhanced basic-block scheduling can assist the hardware to where it achieves over 90%

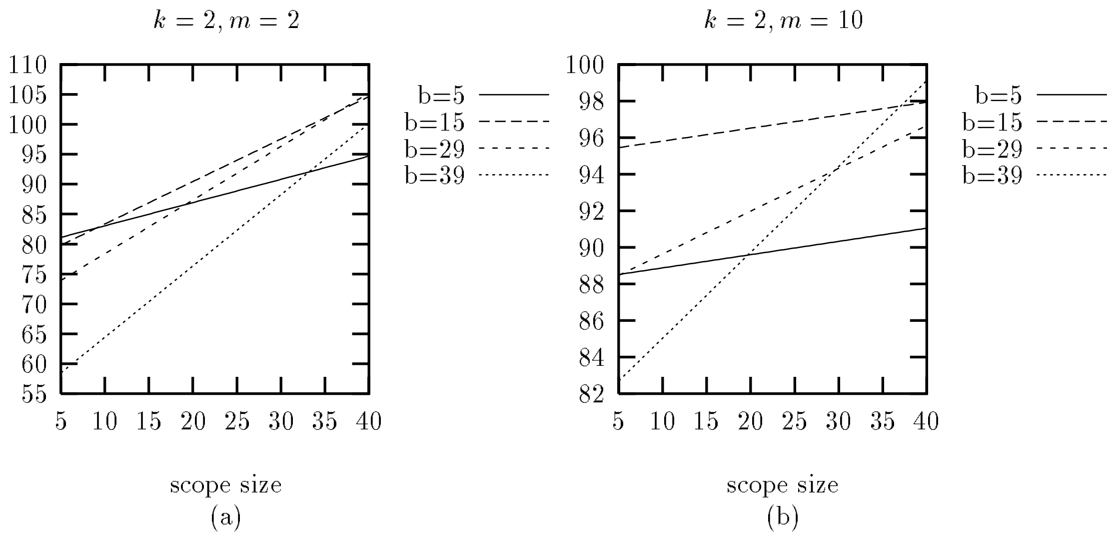

Figure 4: Percentage of relative speedup of *enhanced* run-time scheduling over global — $R_{E,G}$ — with respect a machine with no-hardware scheduling for pipelines of depth 2

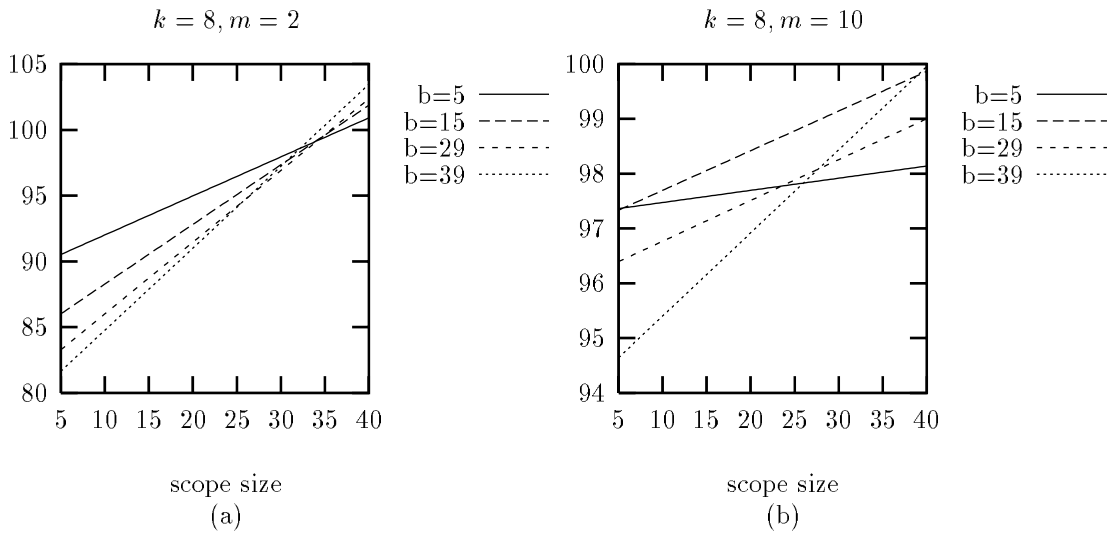

Figure 5: Percentage of relative speedup of hardware $R_{L,G}$ for pipelines of depth 8

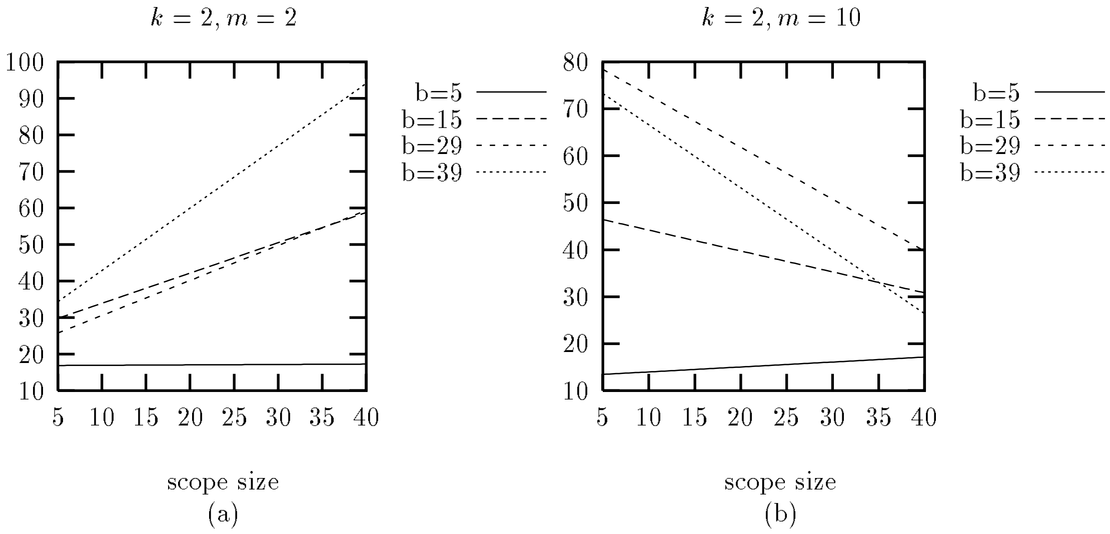

Figure 6: Percentage of relative speedup of hardware with enhanced basic block scheduling $R_{E,G}$ with respect to L

of the (maximum) possible speedup that global has to offer. Furthermore, with few exceptions, it reaches the limit of 100% with a scope size of 40; exceeding 100% implies that we can actually do better than the form of global scheduling used in our experiments, which is very unlikely.

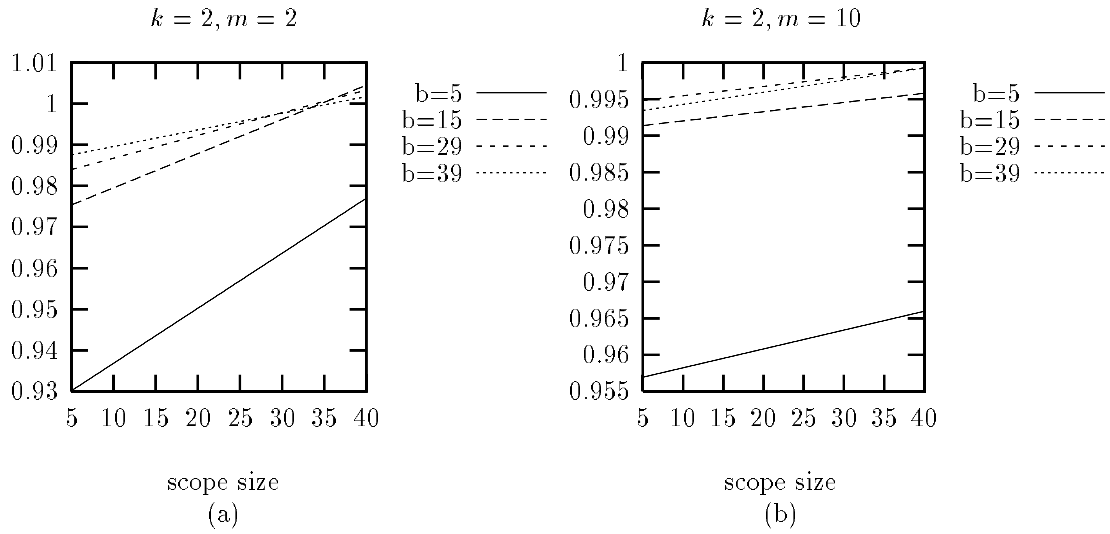

Figure 7: Absolute speedup of hardware with enhanced basic block scheduling $S_{E,G}$

A second interesting trend is that this speedup grows with the size of the basic block. We expect that this trend is driven by the acknowledged improvement of basic-block schedulers, with increasing basic-block sizes.

Absolute Speedup $S_{E,L}$ with respect to V

This ratio directly yields the benefit gained by enhancing the basic-block scheduler using the algorithm in Section 6. The broad trends are indicated in Figure 9. The most significant trend to observe is that the advantage that the enhanced scheduler of type E has over the more rudimentary kind of scheduling L, goes

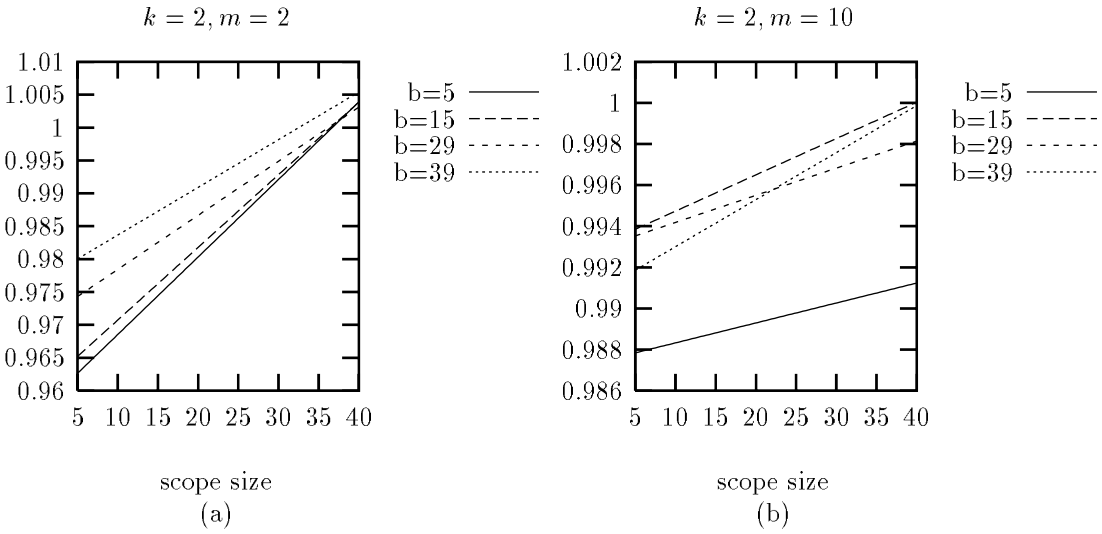

Figure 8: Absolute speedup $S_{E,G}$

down with increasing scope in the run-time scheduler. We postulate that this is due to the fact that hardware is able to discover most of the instruction level parallelism in the presence of large scope, and any extra compile time help becomes inconsequential. However, we note that the actual speedup values are very small and are typically in the vicinity of $1 - 3\%$[9].

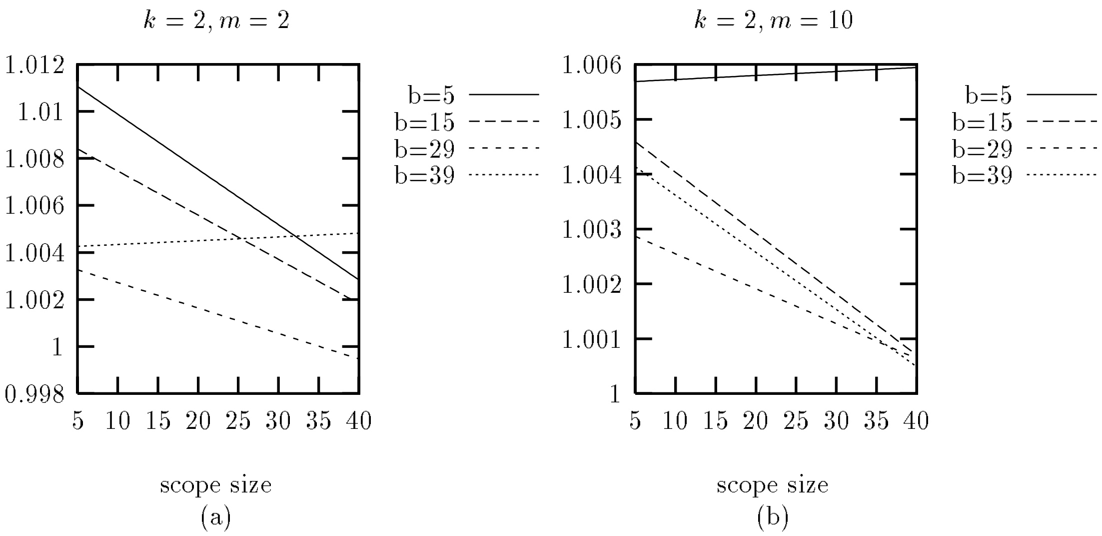

Figure 9: Absolute speedup $S_{E,L}$

Throughput:

As noted before, the trends in throughput are very similar to those demonstrated in the context of speedup. However, an interesting new trend that we observed was that the throughput dropped substantially with increasing ILP in the processor. This implies that despite the scheduling support, the expected performance improvements with increasing ILP are less than linear — the drop in throughput was about 40% when the

[9]This range is not immediately apparent without inspecting the full data set, which will be provided in the full-version.

number of functional units goes from 2 to 4 for a pipeline of depth 4. The remaining trends were very similar to those demonstrated by the speedup measures. Therefore, in the interests of not duplicating similar information, we will restrict our discussion to two of the trends as representative examples.

1. The enhanced global scheduling approach E tends towards the global method G with increasing scope and basic-block size.
2. Increasing basic-block size improves throughput in the case of approaches E and L since the basic-block schedulers perform better with larger basic-blocks.

3.3.2 The advantages of compaction

The overall trends that we have discussed thus far were present consistently in processor configurations that compacted the instruction stream, as well as in those that did not. However, a very conclusive and interesting relationship emerged between the trends in the two settings. This relationship has to do with the advantage that compaction has in influencing the performance of the machine. The traditional belief is that compaction benefits run-time schedulers by exposing more instructions in a given "look-ahead window" i.e., the instructions that can be moved out-of-order for a given scope. Our experiments systematically and quantitatively confirm this expectation and we will now highlight this result by comparing the corresponding values of the absolute speedup $S_{L,G}$. The reason for choosing this ratio is that L denotes the raw improvement of hardware-scheduled code with basic-block scheduling being very elementary, as one would expect in a typical superscalar setting today.

In Figure 10, we show the two "cones" for these configurations defined by the speedup ratios, with basic block size 11 and 35 defining the extreme boundaries. These ratios are derived for a machine with two functional units, each of which are pipelined to be of depth two. All the speedup values for each of two cases lie completely within the appropriate "cone". Note that the cone for the case of compaction dominates that for the case without any compaction. Furthermore, there is an overlap in the region where the latter starts achieving similar speedups as in the case of machines with compaction, however for larger basic-blocks.

Now, as we increase k to 8, the two cones completely separate out as shown in Figure 11 and the speedups achieved by the case with compaction completely dominate those without, as a dramatic demonstration of this trend. In this case, the compaction has a very pronounced advantage in yielding performance improvements.

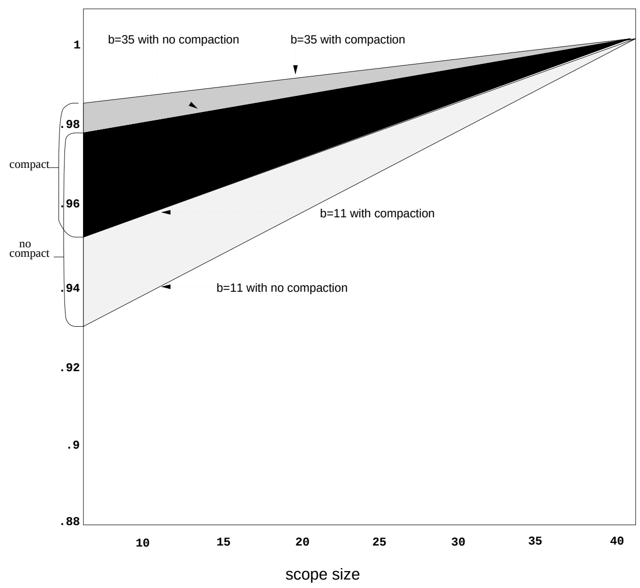

Figure 10: The shift in the absolute speedup $S_{L,G}$ with $k = 2$, $m = 2$

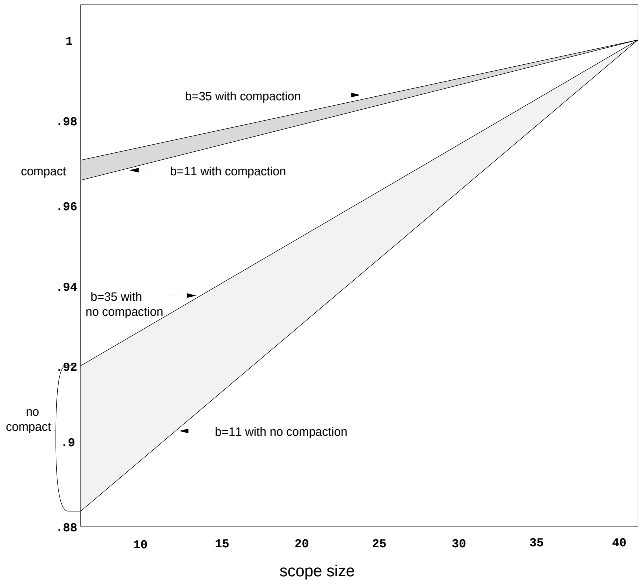

Figure 11: The shift in the absolute speedup $S_{L,G}$ with $k = 8$, $m = 2$

Finally, this advantage of compaction is diminished somewhat with increasing ILP as shown in Figure 12. For example, as m is increased to 10 the speedup gains of the configuration with compaction are comparable to those without.

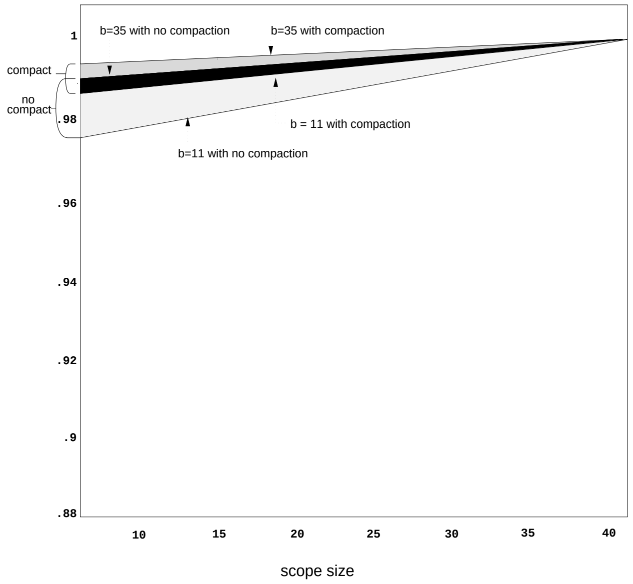

Figure 12: The shift in the absolute speedup $S_{L,G}$ with $k = 2$, $m = 10$

Collectively, we can conclude from these trends that the benefits of compaction are particularly advantageous and hence desirable to have in the machine: (i). if the scopes are not too large, (ii). when the machine has limited to modest ILP (≤ 4 functional units), (iii). or when the program has relatively small basic-blocks. If one or more of the above three attributes are not true, then the added cost to support compaction ought to be reconsidered if hardware resource is an issue.

4 Modeling the Domain

The domain of interest to us consists of programs executing in a modern superscalar RISC processor. For convenience, we will introduce the modeling details in two parts. First, in Section 4.3 below, we will present some preliminaries that formally define the notion of instruction scheduling in general; this will not involve any discussion of hardware for the support of dynamic (run-time) scheduling. Following this, in Section 4.4, we will formally define the model to account for the latter case as well.

4.1 Program Representation

As noted before, the scope of our optimizations are instruction traces. A trace is formally represented as a weighted *D*irected *A*cyclic *G*raph (or *DAG*) $G = (V, E; BB, \rho, w)$, where each instruction i on the trace is represented as a node in V. We use $BB(i)$ to denote the basic block in which i occurs in the original program. We also have a function ρ which maps basic blocks into the set $\{1,\ldots,L\}$, such that $\rho(x) < \rho(y)$ if basic block y is control dependent on basic block x. Informally, this function gives us the sequence in which basic-blocks occur were we to adhere to the control-flow of the original program. The edges E in the trace represent data dependences. There is a directed edge from node i to node j whenever instruction i has to be executed before instruction j.

These traces execute on a pipelined machine and hence, might involve additional delays during the execution due to latencies. Specifically, given an instruction i issued at time $\tau(i)$, it may not always be possible to issue another instruction j at a later time $\tau(i) + t, t > 0$, because the result(s) of i are not yet available and j uses them. This delay is referred to as the *inter-instruction latency* (or *latency* for short) of i relative to j, and is modeled by the integer weight $w(i, j)$ given to edge (i, j). An example of a graph representing the program is given in Figure 13, where a program consisting of two basic blocks is shown. Each instruction is represented by a node (which for ease of reference are labeled). The last instruction, c, in basic block 1 is the conditional branch. Control flows to basic block 2 from block 1, as determined by using profiling information to be the most likely path[10]. The basic block to which an instruction belongs to, is represented by the index of that instruction. The first basic block is composed of instructions a, b, c and d, and basic block 2 is composed of instructions t, u, w, x, y and z. The number next to an edge connecting two nodes represents the latencies between these instructions.

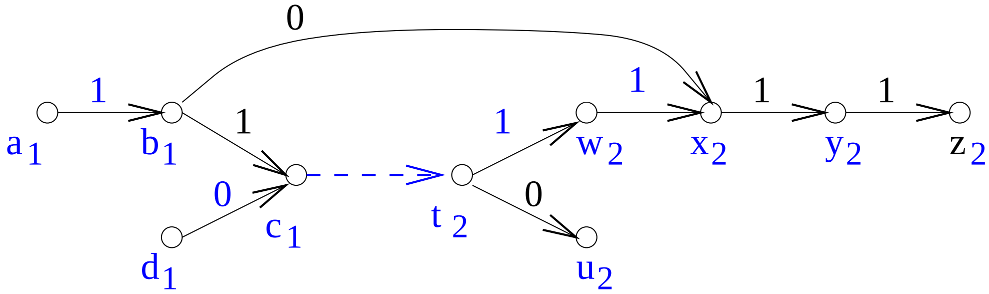

Figure 13: Graph representation of a program

4.2 The Superscalar Processor

These traces execute on a superscalar pipelined processor specified as follows. There are a total of T types of functional units in the processor: fixed point units, floating point units, branch units and so on. Each functional unit type TY is implemented in a superscalar processor as a pipeline with k_{TY} stages; the number of stages is also referred to as the depth of the pipeline. For example, let us consider a machine with fixed and floating point units denoted respectively as types 1 and 2. Also let the floating point unit have three

[10] Control could also flow out of block 1 to other basic-blocks not shown in the figure, with lower probabilities.

stages and the fixed point unit have two stages; $k_1 = 3$ and $k_2 = 2$. For convenience, we will represent the type of functional unit f in the target processor by the function $type(f)$.

Let m_t be the number of functional units of type t. Then there are a total of $m = \sum_{1 \leq t \leq T} m_t$ functional units. A particular instruction i in the trace might need to use a specific functional unit, such as a fixed point unit. This requirement is specified by $unit(i)$ for each instruction i in the trace.

4.3 In-order Execution

The CPU fetches instructions from memory and executes them. Instructions are arranged in a linear sequence generated at compile-time which will be referred to as the *instruction stream*. For convenience, let us consider these instructions as residing at increasing address locations in the program space. Based on these addresses, a natural order is imposed on them: $i < j$ if and only if instruction i resides at a lower address than instruction j; we will informally say that i occurs earlier than j in the stream.

To describe the run-time behavior of the processor, we will adopt a simple idealization which is valid from the viewpoint of the instruction scheduler. In particular, we will consider a model that reflects the behavior of early RISC processors [21, 16, 35] that did *not* embody any dynamic support for instruction scheduling; instructions were executed strictly in the order specified by the compiled instruction-stream.

In this setting, we have a *program counter* PC which always points to the next instruction i to be executed. We have a parameter S which denotes the current scope of the window from which instructions can be executed[11]. At any given cycle, instructions are either *ready* or are *waiting*; we will return to a formal definition of this issue later. Informally, ready instructions can be *issued* since all their inputs have been computed and ready. Note that we can issue at the rate of one instruction per-functional-unit on each cycle. In this section, we will assume that[12] $S = m$.

In this model, the processor chooses no more than m_t instructions of type t such that whenever any instruction j is chosen and issued on the current cycle, every instruction $1 \leq i' < j$ must also be issued. In other words, the processor issues as many instructions that can be issued, subject to availability of functional units, in contiguous order starting with the currently available one. It stops issuing as soon as a "non-ready" instruction j' is encountered at which point PC is updated to point to j'. An example is provided in figure 14.

An instruction stream is shown here with the instructions having the dependences shown in figure 13. Consider that the machine it is executed on has three identical functional units, capable of executing any of the instructions in the example. For each time step, the instructions in the current scope are shown in boldface, and those which are executed are underlined. Note that at time step 2, even though both b and t are ready and in the scope, only b can be issued because c is not ready.

4.4 Out-of-order Execution and Hardware Lookahead

As noted before, the trend in modern superscalar RISC processors is to have varying levels of run-time scheduling [14, 15]. In this case, the processors have special hardware support [27, 22, 7] to analyze the current program dependences "on-the-fly" and schedule as many independent instructions as possible on a given cycle. Consequently, in these modern processors and in the interest of finding multiple independent instructions that can be co-scheduled, invariably $S > m$.

Once again, let the current instruction to be issued, and hence the one which PC points to, be i. In our present setting, the processor issues greedily as many ready instructions as possible from the range $i, (i+1) \ldots (i+S-1)$ of the instruction trace. For convenience, we will refer to $w = S - m$ to be the *width* of the lookahead window.

Having stated this, it is now interesting to consider the possibility that should an instruction be executed

[11] In the very early machines, $S = 1$ since the processors had a single pipelined functional unit.

[12] If $S < m$ then the machine is not able to start instructions on all its functional units in the same time step, and resources are under-utilized.

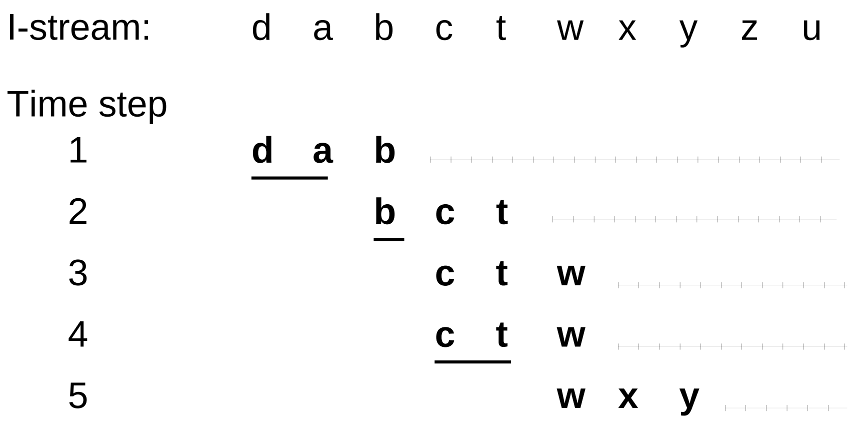

Figure 14: Inorder execution

dynamically out-of-order, it should not be *re*-executed as the program counter advances as shown in the example. In Figure 15 we run the same instruction stream as in Figure 14, on a machine with three functional units, but capable of out-of-order execution this time. At time step 2, both instructions b and t are issued, but not c; hence, t remains (inhibited) in the scope until c is issued, as indicated by the shaded boxes in the figure.

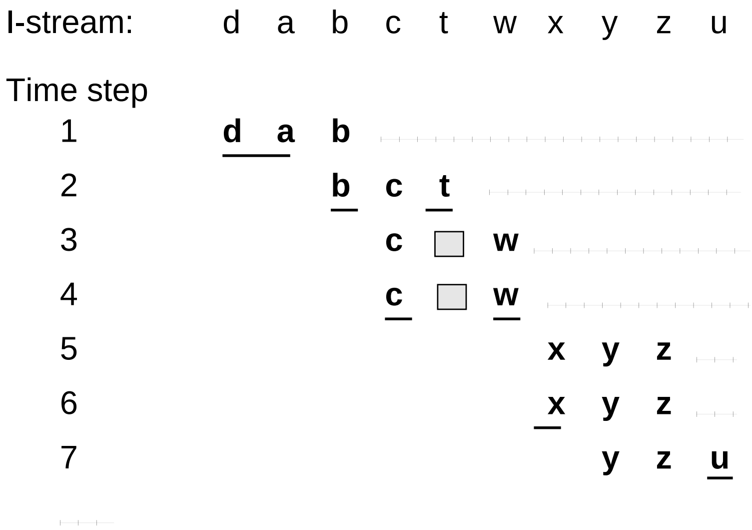

Figure 15: Out-of-order execution without compaction

Typically, hardware implementations attempt to solve this problem in one of two different ways and we will briefly review each in turn for completeness.

4.4.1 Architectures that Don't Compact

In the earlier settings for machines with lookahead, the problem of avoiding incorrect executions — such as reexecuting instruction t in the previous example — were dealt with by "tagging" them. As the PC advances, the hardware does not execute tagged instructions but rather skips over them.

For the purpose of present discussion, we will abstract this notion of tagging and represent it conceptually as a single *inhibition bit* that is associated with an instruction in scope; note that this is meant only as a mechanism for explaining the notion and need not reflect the actual implementation. In this setting, the instruction always remains in the scope whether it has been executed or not. However, it is not reexecuted if its inhibition bit is set.

Upon issuing a superscalar instructions, the *PC* is readjusted to point to the *next* instruction in the stream that has not been issued yet, as shown in Figure 15.

A disadvantage of the above implementation is that "dead" instructions remain in the instruction stream and use up a part of the current scope. Consequently, only a part of the instructions in the current scope — those whose inhibition bits are not set — are useful to the run-time scheduler. However, the current trend in processor design with instruction caches is helping overcome this drawback; we will briefly review this approach now.

4.4.2 Compacting the Instruction Stream

In the presence of the instruction cache, the processor moves instructions from program memory to the cache, from where they are executed. Consequently, the current snapshot of the instruction stream that the processor executes is the part that is resident in cache. In this setting, dispatching a subset of instructions to cache and executing them (logically) eliminates them from the current scope. Only those instructions that have not been executed yet remain in cache. Consequently, in this approach, *all* the instructions in the current scope are eligible (i.e. not inhibited) and hence potential candidates for execution. As noted in our previous results (Section 3), compaction offers performance advantages since no part of the scope is wasted on dead instructions. An illustrative example is shown in Figure 16, where the program as before was run on a machine with three functional units, and out-of-order execution with compaction. Note that with compaction, in cycle 3, instruction t is no longer in the scope.

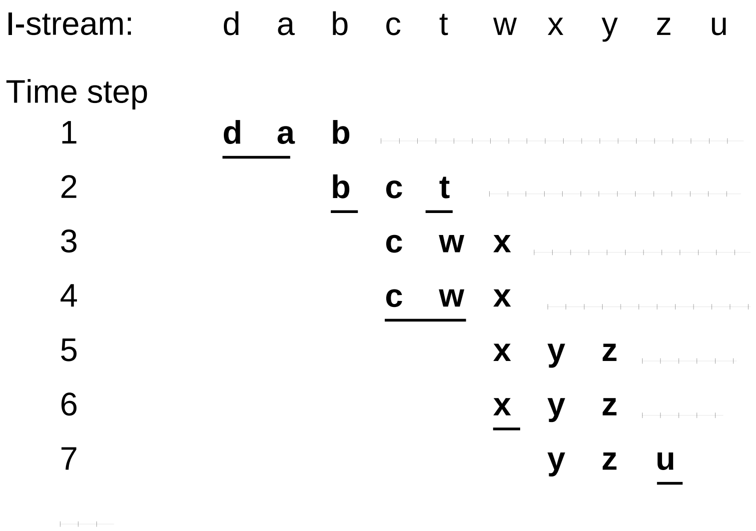

Figure 16: Out-of-order execution with scope compaction

5 Instruction Scheduling at Compile-time

Instruction scheduling is recognized to be an important step in producing optimized code for RISC processors with ILP; this was especially true of the early processor designs that did not include any hardware scheduling support. As a result, optimizing compilers incorporated an instruction scheduling phase as shown in Figure 17.

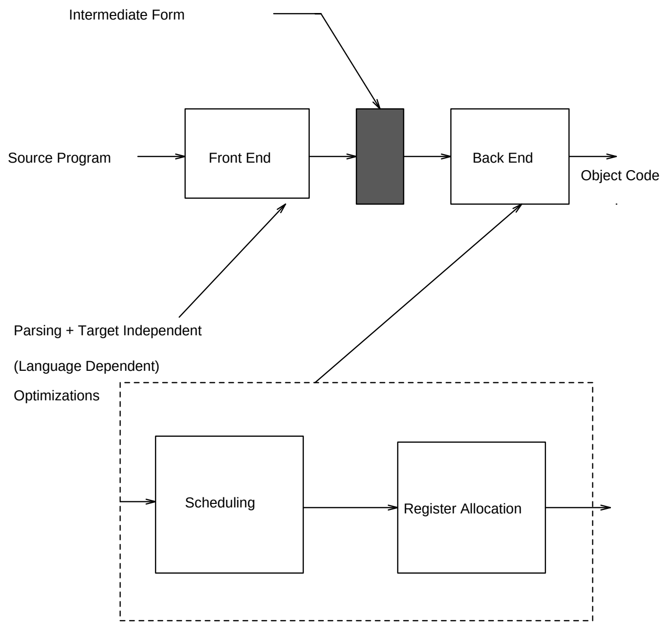

Figure 17: Instruction scheduling phase of the compiler

Informally, this phase typically involved arranging the instructions statically at compile-time so that if the instructions are executed in this sequence on the target machine, the overall completion time is minimized. An example of a schedule for the program with two basic-blocks in Figure 13 for a machine with two functional units is shown in Figure 18. In the rest of this section we will discuss the notion of scheduling a trace, as if it were done in isolation, that is, as if only the trace is executed on all runs of the program[13].

Formally, given a trace G, a *schedule* S is a pair of functions (σ, M), where $\sigma(i)$ denotes the *scheduled start time* of instruction i ($\sigma : V \to N$), and $M(i)$ denotes the functional unit assigned to instruction i ($M : V \longrightarrow \{1...m\}$).

When scheduling a trace on the target machine, we need to ensure that the data-dependences (inter-instructional latencies) and the resource availability in terms of the number of functional units, are all obeyed. Such a schedule is a *valid schedule* if it satisfies both the data dependency constraints and the resource constraints, as defined below:

1. For all $(i,j) \in E$, $\sigma(j) \geq \sigma(i) + w(i,j) + 1$.
   i.e., The precedence constraints and inter-instructional latencies are satisfied for each instruction.
2. For all $i, j \in V$ and $i \neq j$, $\sigma(i) = \sigma(j)$ implies that $M(i) \neq M(j)$.
   No two instructions can be scheduled to start on the same functional unit in the same time step.
3. For all $i \in V$, $unit(i) = type(M(i))$.
   An instruction of type t must be scheduled on a functional unit of the correct type.

A schedule is represented as a table, where column i represents the set of instruction scheduled to be executed at time step i. The goal of instruction scheduling is to produce a schedule with minimum length. That length is referred to as the *completion time*. The instruction stream is obtained from the schedule by linearizing the table column-wise. For example, the instruction stream in Figure 18 is produced by linearizing the schedule S as shown[14].

[13] In reality, it is likely that off-trace branches are taken, but our experiments and comparisons are defined completely in the context of a trace.

[14] In the early RISC processors, the linearized instruction stream had to include the idle-cycles since hardware did not support interlocks; we assume that all current machines support hardware interlocking and hence idle-cycles can be skipped.

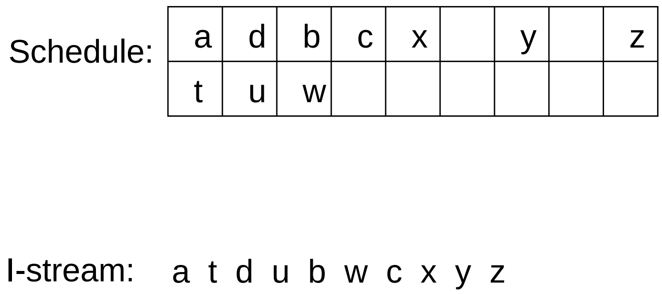

Figure 18: A schedule and its linearization

5.1 Local versus Global Scheduling

The above description of instruction scheduling was in the context of a given program trace, and hence can span multiple basic-blocks. We will call a schedule to be *local* when any two nodes i and j in different basic blocks appear in the order of their basic blocks: $BB(i) < BB(j) \Rightarrow \sigma(i) < \sigma(j)$. A schedule which is not local is called *global*. In Figure 19, three schedules are presented for the program in Figure 13 for a machine with only one functional unit. Schedules S_1 and S_2 are local schedules, and S_3 is a global schedule. It can be seen that they have different completion times, with the global schedule having the smallest one of these three. Also, S_1 has a smaller completion time that S_2 and indeed it is the best local schedule possible for a machine with inorder execution.

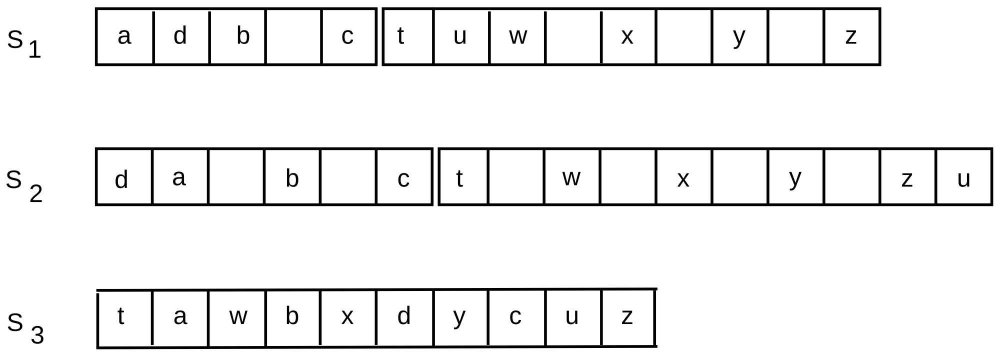

Figure 19: Different possible schedules for a previous graph example

5.2 Global Scheduling and Side-effects

A significant issue that arises in the context of moving instructions globally — either statically at compile time, or dynamically at run-time — is that instructions move across branch points creating side-effects referred to typically as *speculation* and *replication*. In either case, and independent of whether the compiler or the hardware performs the code-motion, these side-effects require expensive repair, leading to *overheads*, as detailed in Figure 20; for a complete discussion of these issues, please see [9].

In this figure, the top row details code motion where the instruction goes from being executed *sometimes* since it is on one of the two conditional branches, to being executed *always* after the code-motion. This type of an execution is referred to as speculative, since the instruction I in question is now being executed independent of which branch is taken, whereas in the original program it would have been executed only when the left branch is taken.

The complementary situation is shown in the two figures in the bottom row. In this case, after code motion, instruction I is executed only when the left branch is taken, whereas previously, it is executed independent

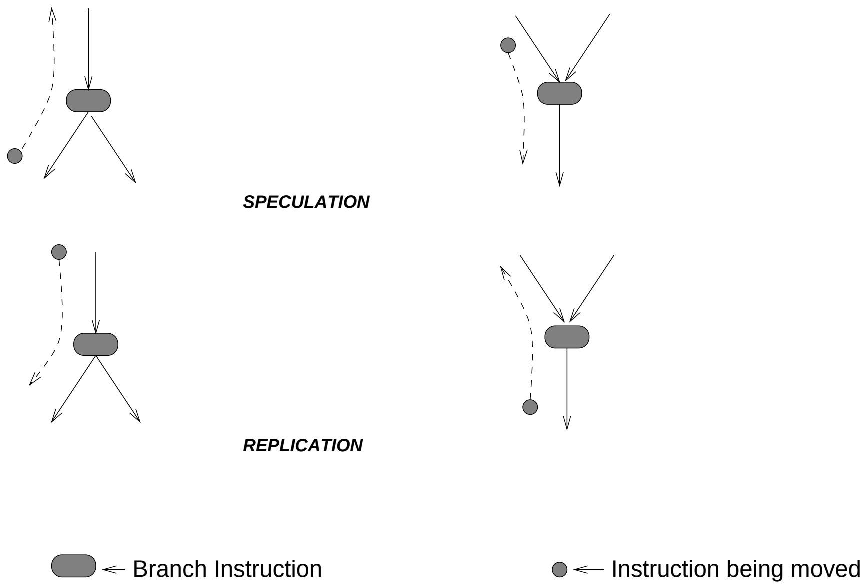

Figure 20: The Compilation Path

of the branch. The solution in this case is to make a copy of I on the right branch as well — hence, the term replication.

Both of these side-effects can lead to prohibitive costs and consequently — depending on whether scheduling is performed in hardware or by the compiler — the designers limit the degree to which instructions can cross basic-block boundaries.

In our experiments, we will permit instructions that are not branches or merges to be moved without any restrictions. We note that in the case of run-time scheduling by hardware, this indeed is the current trend since the idea is that the scope extends only as far as complete code motion is possible. We will also permit similar freedom to global compile-time instruction scheduling on the trace.

In contrast, branches (and merges) are treated differently since the corresponding overheads can be prohibitive [11]. Consequently, current approaches to scheduling in both the compiler as well as the run-time hardware, is to limit the relative reordering of these instructions completely. We will retain this restriction in all of our experiments.

6 Enhanced Basic Block Scheduling Algorithm

Given that global scheduling is expensive to do, we would like to have a local schedule, so that it is simpler, but which takes advantage of what the machine can offer in terms of run-time scheduling. By not taking advantage of this knowledge, a better schedule might be missed. For example S_1, even though optimal for a machine with inorder execution, has a greater running time (13 time steps) than the sub-optimal S_2 (12 time steps) on a machine with out-of-order execution and scope $s = 3$. The algorithm presented below has these desirable features we wanted, and it is this algorithm that we have referred to as method E in previous sections.

The input to algorithm E is the graph $G(V, E; BB, \rho, w)$, as presented in the introduction. The algorithm consists of two phases: *assigning priorities* to nodes and *greedily scheduling* them. However, both phases differ from those existing in traditional local schedulers.

6.0.1 Phase 1: computing priorities

The priority of a node has two components: a *local priority* and a *global priority*. For both components we will use the rank prioritizing function as presented in [29].

The way to compute the two components is described in the following. Consider each basic block A in the trace (except the last one), and its successor B. Compute ranks to all instructions in basic blocks A and B considered together. The rank obtained for an instruction i will be its global rank component if i is in basic block A, or its local component if i is in basic block B. For the first basic block compute local ranks by considering it separately. Instructions in the last basic block in the trace will have their global priorities equal.

These two components can be combined in different ways for producing the priority of a node, resulting in different algorithms. We have experimented with two such possibilities: (i). we have given the two components equal weight and (ii). the priority of the node is established lexicographically (i has greater priority that j if i's local priority is greater, or if they have equal local priorities and i's global priority is greater). Because these schemes produced very similar results, henceforth we shall not distinguish between them.

6.0.2 Phase 2: scheduling

As in the traditional greedy scheduling, we maintain for each node i the earliest time step at which node i can be scheduled if it is to observe the data dependences it has on its already scheduled predecessors. This information is stored in *start*[i], and is initialized at 0. We also maintain *indegree* as the indegree of node i in the subgraph of G obtained by removing the outgoing edges of all scheduled nodes. Initially, *indegree*[i] is set to the indegree of i in G.

We maintain a *waiting* list of instructions whose *indegree* became 0. The list is prioritized by their *start* value, and initially it is empty.

We also have for each basic block a list called *ready* which keeps instructions that can be scheduled at any time (subject to resource constraints). The *ready* lists are prioritized by the node's priorities, and are initialized to contain all nodes with *indegree*=0 in G.

Scheduling of node i consists of adjusting *start*[j] and *indegree*[j] for all direct descendants j of i: $start[j] \leftarrow max\{start[j], time + w(i,j) + 1\}$, and $indegree[j] \leftarrow indegree[j] - 1$. Of course, i gets added to the scheduled instructions for basic block $BB(i)$. Every time an instruction is scheduled, the number of instructions already scheduled (and the number of instructions remaining to be scheduled) in that basic block is incremented (respectively decremented).

A node i that has not been scheduled, is eligible for scheduling if it is ready (i.e. is in $BB(i)$'s ready list) and either (i). all instructions in basic blocks preceding $BB(i)$ have been scheduled or (ii). all ready instructions in preceding blocks have been scheduled, and the sum of the number of instructions from $BB(i)$ already scheduled together with the number of instructions remaining to be scheduled in all the preceding blocks is less that s.

After scheduling nodes on all functional units has been tried, *time* is modified as follows: if at least one *ready* lists is not empty, *time* is incremented; otherwise it is advanced to the value of the smallest *start* value from the *waiting* list. All nodes i from *waiting* whose *start*[i]=*time* are moved to the *ready* list of the corresponding basic block. The above process is repeated until all nodes have been scheduled.

6.0.3 Comparing the local and lookahead algorithms

The main difference between the lookahead and local algorithms is that the former "looks" at the next basic block(s) to gain more information. It improves on the local scheduling in both its components: priorities and scheduling. The ranks of the lookahead algorithm are a "refined" version of the local ranks because they take into account inter basic block dependencies. Also, the modified scheduling phase has the effect of discovering that instructions with high priority in the second basic block will become ready at some early time step only because of out-of-order execution of its predecessors. Consequently it is unnecessary to fill all the idle slots among these instructions, because they would be executed speculatively.

7 Simulation and Trace Generation

Using software simulation, we have gathered data on the behavior of programs executing on machines with various degree of scope. The experiment is conducted in the following way. First, synthetic traces representing programs for machines of varying configurations are generated, with a total of 30 samples for each configuration. We then run the traces through our scheduler on three different algorithms — local, enhanced local and global. Each generated schedule is then fed into a simulator mimicking processors with and without compaction. As is typical in evaluating instruction scheduling approaches in isolation to understand its raw benefits, our simulator models an idealized RISC machine with an unbounded number of registers.

Our trace generator produces sample data in two phases: instructions generation and dependencies generation. In the first phase the boundaries of basic blocks are determined. We accomplish this by randomly selecting the locations of the branch instructions, i.e., the boundaries of each basic block. The location of the branch instruction, given a trace with average of basic block size of b, is chosen using a gaussian random source so that it is in the vicinity of the i_bth instruction of the trace. Then, load, store and arithmetic instructions in the trace are generated in specified frequencies — these are input parameters — and distributed uniformly across the trace. Once generated, each instruction is tagged by its type which determines the number of inputs and outputs that it has.

Branch instructions are given one input edge to simulate its dependency on a a control variable. Load instructions are either given zero or one edge; the one-edge case simulates their dependence on address calculation. Stores are given one edge to simulate dependence on the value to be stored. Finally, arithmetic instructions are assumed to have up to 2 input edges representing the dependences on input arguments.

Next, we simulate the manner in which a compiler generates dependency edges during our "edge" generation phase by selecting the edges for each instruction one-at-a-time as follows: First, we maintain a set of **definitions**, or simply def nodes which denote the locations where variables are formally defined. This set represents the current collection of useful values which will be **used** further down the instruction stream. The instructions on the trace is then processed in sequence and dependences are added as follows.

Depending on the instruction's type, the number of edges that feed it and hence the number of predecessors in the graph are well defined, as stated above. Given this fact its predecessors are chosen from the set of current defs randomly. After the dependency edges of an instruction have been selected, the instruction is added to the defs set, provided that it generates a new value as in the case of a load or an arithmetic instruction. We then repeat this process until all the instructions on the trace are processed.

To make the generated trace realistic, we also simulate the locality of reference of typical programs — the value used as input for an instruction tends to refer to result computed recently. Each definition in the defs set has a chance of getting deleted from this set after an instruction is processed. Additionally, we skew the probability of it being deleted from the current set of defs so that recent additions to the set have a better chance of surviving.

8 Analytic results

We have analytical characterizations of the limits to speedup that the run-time scheduler with lookahead hardware can achieve over processors that do not have this feature. Unlike the experimental phase, we will not be considering the impact of any compile-time optimization. We will use T_V to denote the running time of a given program on a machine with no hardware scheduling, but with an instruction scheme generated by greedily optimizing basic-blocks locally; only step 3 from our overall framework for instruction scheduling is applied. The same stream is also fed to a superscalar processor with hardware scheduling and takes T_H

cycles. All the machine and program parameters are otherwise the same, and the processors do not compact the instruction stream in what follows.

As before, we say that an instruction i is executed speculatively provided it is executed out of order across basic-block boundaries; it is executed ahead of at least one branch[15] instruction ahead of it in the original instruction stream. Let us characterize the *speculative degree* d of the scheduling scheme to be the number of basic block boundaries that an instruction can cross in the above sense. When the speculative degree $d > 1$, the instructions can be pulled from two or more different basic blocks and interleaved together in a single cycle. This value is of interest to us since the advantage that the hardware scheduler has, and hence improvements to T_H follow from this movement. More precisely, maximal amount of interleaving M is limited by the maximal speculative degree of the hardware scheduler d, and the number of available idle slots, i.e.,

$$M \leq \min(d+1, m(k+1)) \tag{1}$$

An interesting analytical characterization of the speedup $\frac{T_V}{T_H}$ is:

Theorem 1 *For all k, m, d, and s,*

$$\begin{aligned} \frac{T_V}{T_H} &\leq 1 + M\frac{mc(k+1)}{mc(k+1) + M(b-c)} \\ \textit{with} &\quad M = \min(d+1, m(k+1)) \\ \textit{and} &\quad c = \min(s-1, b(s-1)/s) \end{aligned}$$

Here, c denotes the average number of instructions that are executed speculatively per basic-block and equals $\frac{S}{L}$ where S is the total number of speculatively executed instructions and the given program trace has L basic blocks in it.

Due to limitations of space, we will not be able to present the details of how this speedup is derived but will summarize some of the main trends below.

- If $b \rightarrow \infty$, then $c \rightarrow (s-1)$, and $T_V/T_L \rightarrow 1$

  This is quite intuitive since, as the average length of the basic block increases for a given size of the trace, the number of basic-blocks in it go down. Consequently, the compile-time basic-block scheduler performs increasingly better in relation to the run-time scheduler.

- If $s \rightarrow \infty$, then $T_V/T_L \rightarrow 1 + M$

  If the scope s increases while the other parameters stay fixed, the performance of H becomes increasingly better and approaches that of a global method G whereas V's performance does not improve. It is easy to see that $1 + M$ is the maximum speedup possible since it is the limit of the number of instructions that can be moved out of order by the run-time scheduler to fill idle cycles.

- If $m \rightarrow \infty$, then $M = d$, and $T_V/T_L \rightarrow 1 + d$

  This tells us that by increasing the number of functional units, we can increase the speedup that run-time scheduling has to offer, but the increase is eventually limited by the number of instructions that can be moved across basic-block boundaries and hence by permitted speculative degree d.

9 Conclusions and Remarks

The thrust of our paper has been to understand the relative value of incorporating instruction scheduling in hardware via run-time schedulers, versus compile-time support for it. These tradeoffs between architecting

[15] In general, these could be merge instructions also.

the compiler versus the processor are ubiquitous to the RISC wave, and we view the results of our study as a step in understanding this boundary in the context of instruction scheduling.

A significant conclusion is that in the presence of processors with run-time hardware, the need for aggressive compile-time global scheduling is not very pronounced. However, we do show that dynamic schedulers improve with some assistance from local (basic-block) scheduling, done at compile time. We found it surprising that the performance of the run-time scheduler was substantially good even when the competing compile-time global scheduler was endowed with the ability to predict branches perfectly — an expectation that is not true in practice.

All of the above experiments were performed with synthetically generated program traces, as described in Section 7. Our intention in doing so was motivated by the control that we have in selectively generating families of these program structures with emphasis on particular parameters in isolation; for example, we could structure the programs to emphasize particular pipelined latencies and mixes of instruction types. We are currently in the process of extending our experiments to include "realistic" workloads derived from the SPEC family of benchmarks. Here, we are working closely with Christopher Gleaser of Nullstone Corporation to generate the representative test programs and necessary profiling information.

10 Acknowledgments

The authors wish to thank Pradeep Dubey for helpful discussions about the trend in commercial superscalar processors and the uses of compaction. They also wish to thank Zvi Kedem for discussions that helped consolidate the interpretation of absolute speedup, and Salem Reyen for providing the initial code for our experimental scheduler.

References

[1] Ramón D. Acosta and Jacob Kjelstrup and H. C. Torng. An instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors *IEEE Transactions on Computers,* C-35(9):815-828, 1986

[2] F. Allen, B. Rosen, and K. Zadeck (editors). *Optimization in Compilers,* ACM Press and Addison-Wesley (to appear).

[3] D. Bernstein and I. Gertner. Scheduling Expressions on a Pipelined Processor with a Maximal Delay of One Cycle In *ACM TOPLAS* 11(1), pp. 57-66, 1989

[4] D. Bernstein and M. Rodeh. Global Instruction Scheduling for Superscalar Machines. In *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation,* pp. 241–255, 1991.

[5] E. G. Coffman Jr. and R. Graham. Optimal Scheduling for Two Processor Systems *ActaInform. 1* 1971, pp. 200-213

[6] P. Dubey and G. B. Adams, III and M. Flynn. Instruction Window Size Trade-Offs and Characterization of program Parallelism In *IEEE Transactions on Computers* 43(4) 1994

[7] R. Cohn and T. Gross and M. Lam and P. S. Tseng. *Architecture and Compiler Trade-offs for a Long Instruction Word Microprocessor* In Proceedings of ASPLOS III, Apr. 1989, pp. 2-14

[8] Keith Dieterdorf and Rich Oehler and Ron Hochsprung. Evolution of the PowerPC Architecture *IEEE Micro* pp. 34-49 1994

[9] J. Fisher. Trace Scheduling: A General Technique for Global Microcode Compaction. *IEEE Transactions on Computers,* C-30(7):478–490, 1981.

[10] W. M. Johnson. Superscalar Microprocessor Design Prentice Hall, Engelwood Cliffs, NY, 1991

[11] Joseph Fisher. Global Code Generation for Instruction-level Parallelism: Trace Scheduling-2 1991

[12] J. Fisher. J. Ellis, J. Ruttenberg, and A. Nicolau. Parallel Processing: A Smart Compiler and a Dumb Machine. In *Proceedings of SIGPLAN'84 Symposium on Compiler Construction,* pp. 37–47, 1984.

[13] J. Goodman and W. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Proceedings of ACM Conference on Supercomputing,* pp. 442–452, 1988.

[14] Linley Gwennap. 620 Fills Out PowerPC Product Line Microprocessor Report, October 1994

[15] Linley Gwennap. MIPS R10000 Uses Decoupled Architecture Microprocessor Report, October 1994

[16] J. Hennessy and J. Jouppi and J. Gill and F. Baskett and T. Gross and C. Rowen and J. Leonard. The MIPS machine In *Proc. IEEE Coupcon* San Francisco, CA, Feb. 1982

[17] J. Hennessy and T. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM TOPLAS,* 5(3), 1983.

[18] W.-M. W. Hwu et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. The Journal of Supercomputing, Vol. 7 (1993), 229-248.

[19] W. Hwu and P. Chang. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *Proceedings of 16th Annual Symposium on Computer Architecture,* pp. 242–250, 1989.

[20] Ray A. Kamin III and George B. Adams III and Pradeep K. Dubey. Dynamic Trace Analysis for Analytic Modeling of Superscalar Performance In *Performance Evaluation 19 (1994)* pp. 259-276, 1994.

[21] M. Katavenis. Reduced Instruction Set Computer Architecture for VLSI MIT Press, Cambridge, MA, 1984

[22] D. Kuck and Y. Muraoka and S. Chen. On the Number of Simultaneously Executable in Fortran-like Programs and Their Resulting Speedup *IEEE Transactions on Computers,* C-21, pp. 1293-1310, Dec. 1972

[23] M. Lam. Software Pipelining: An Effective Method for VLIW Machines In *Proc. ACM SIGLPAN'88 Conference on Programming Language Design and Implementation,* pp. 318-327, 1988.

[24] S. McFarling and J. Hennessy. Reducing the Cost of Branches. In *Proceedings of 13th Annual Symposium on Computer Architecture,* pp. 396–403, 1986.

[25] S. McFarling. Procedure Merging with Instruction Caches. In *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation,* pp. 26–28, 1991.

[26] S. Moon and K. Ebcioglu. An Efficient Resource-constrained Global Scheduling Technique for Superscalar and VLIW Processors In *Proc. MICRO-25* 1992

[27] Toshio Nakatani and Kemal Ebcioglu. Making Compaction-Based Parallelization Affordable *IEEE Transactions on Parallel and Distributed Systems,* 4(9) 1993 pp. :1014-1029

[28] K. Palem and V. Sarkar. Code Optimization in Modern Compilers Lecture Notes, *Western Institute of Computer Science, Stanford University,* 1995.

[29] K. Palem and B. Simons. Scheduling Time-critical Instructions on RISC Machines. *ACM TOPLAS,* 5(3), 1993.

[30] K. Palem and B. Simons. Instruction Scheduling. In *Optimization in Compilers,* (eds: F. Allen, B. Rosen and K. Zadeck). ACM Press and Addison-Wesley (to appear).

[31] D. Paterson. Reduced Instruction Set Computers. *Communications of the ACM,* 28(1):8–21, 1985.

[32] K. Pettis and R. Hansen. Profile Guided Code Positioning. In *Proceedings of SIGPLAN'90 Conference on Programming Language Design and Implementation,* pp. 16–27, 1990.

[33] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops In *Proceedings of the 27-th Annual Symposium on Microarchitecture* (MICRO-27), Nov. 1994

[34] S. Pinter. Register Allocation with Instruction Scheduling. In *Proceedings of SIGPLAN'93 Conference on Programming Language Design and Implementation,* pp. 248–257, 1993.

[35] G. Radin. The 801 Minicomputer. *IBM Journal of Research and Development,* 27(3):237–246, 1983.

[36] G. Sohi and S. Vajapeyam. Instruction Issue Logic in High-performance Interruptible Pipelined Processors In *Proceedings 14th Annual ACM Symposium on Computer Architecture,* pp. 27–34, 1987.

[37] M. Smith, M. Johnson and M. Horowitz. Limits to Multiple Instruction Issue In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 27–34, 1987.

[38] D. Wall. Limits to Instruction Level Parallelism In *Proceedings of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 290–302, 1989.

[39] D. Wall. Predicting Program Behavior Using Real or Estimated Profile. In *Proceedings of SIGPLAN'91 Conference on Programming Language Design and Implementation,* pp. 59–70, 1991.

[40] H. Warren. Instruction Scheduling for the IBM RISC System/6K Processors. *IBM Journal of Research and Development,* 85–92, 1990.