Exploring the Benefits of Randomized Instruction Scheduling

Georgia Kouveli, Kornilios Kourtis, Georgios Goumas and Nectarios Koziris

Computing Systems Laboratory
National Technical University of Athens
{gkouv,kkourt,goumas,nkoziris}@cslab.ece.ntua.gr

Abstract. With the advent of the manycore era, processors are expected to comprise of simple cores, lacking sophisticated optimization capabilities such as out-of-order execution. Therefore, compiler optimizations such as instruction scheduling become relevant once more. At the same time, the increased diversity of processor architectures imposes the requirement of architecture-independence on compiler optimization techniques. The goal of this paper is to improve instruction scheduling by injecting randomness into the list scheduling algorithm used for basic block scheduling. In order for our approach to be practical and architecture-independent, we build upon the existing compiler infrastructure of GCC. We evaluate different methods of exploring the resulting search space of legal instruction schedules. Our approach consists of choosing the best schedule over a number of randomly generated schedules, using profiling on real hardware. We experiment with various processor architectures (XScale, Sun Niagara 2, Cell SPU, Intel Harpertown) and conclude that randomness in instruction scheduling is able to offer significant performance improvement in a noticeable number of cases.

1 Introduction

As processor architecture trends shift to multicore and manycore designs, a large class of future processors is expected to comprise a large number of simple cores, lacking sophisticated optimization techniques such as out-of-order execution. Consequently, compiler optimizations that had, up until recently, a small impact on performance, are becoming (once again) relevant. Among these, efficient instruction scheduling from the compiler is expected to play a more important role in achieving good performance for numerous applications, since processors tend to comprise in-order processing elements as in the case of the Cell B/E SPUs.

On the other hand, as the diversity of different processor architectures increases, the task of implementing a generic instruction scheduling engine within a compiler becomes an interesting challenge. Ideally, a compiler would be able to produce efficient instruction schedules for a wide range of platforms and applications. However, this task is challenging even when considering a single architecture. This is not surprising as even the relatively simple problem of optimal basic block scheduling is NP-complete [3]. Therefore, modern compilers usually rely on heuristics in order to produce satisfactory instruction schedules in the average case.

While instruction scheduling has been an object of extensive research in the past [4, 5, 11, 16], the problem remains open, as there exist cases where widely used compilers fail to produce efficient instruction schedules. For example, when developing for the Cell B/E platform, programmers often resort to manual assembly instruction placement, to fully exploit the processor potential [1]. Thus, we argue for a practical and automatic technique, that will help programmers achieve high performance for their applications. Moreover, this technique should be independent of specific architecture characteristics, so that it can be easily applied to emerging architectures.

The goal of this paper is to improve instruction scheduling, by building upon an existing compiler infrastructure, while remaining practical and architecture-independent. Instead of proposing a radical algorithmic change, we inject randomness into the list scheduling algorithm used for basic block scheduling. We use profiling on real hardware to choose the best schedule over a number of randomly generated schedules. This approach introduces considerable, yet fully parallelizable, overhead to the compilation process. Specifically, the random production of schedules can be offloaded to different cores, thus suiting very well the emerging manycore paradigm.

We evaluate our technique on the GCC compiler. GCC is the most widely-used open-source compiler offering support for a variety of instruction sets, which makes the experimentation with different processors easy. Moreover, GCC ensures the practicality of our approach, since it is a mature real-world compiler, where a great effort has been put into the production of efficient instruction schedules.

The platforms used for the evaluation of our approach are a Sun Niagara 2 multithreaded processor, an in-order Cell SPU, an XScale based evaluation board and a general-purpose Intel Xeon processor (Harpertown). The Harpertown processor is included in the experiments only for completeness, as it implements sophisticated techniques such as out-of-order execution, therefore we do not expect to achieve significant speedups for this platform. Our preliminary results show that inserting randomness in GCC's instruction scheduling can offer significant room for performance improvement for some of the architectures we experimented with.

The rest of the paper is organized as follows: Section 2 describes the preliminaries regarding basic block scheduling, while Section 3 explains our approach and discusses implementation details. Section 4 presents the results of our experimental evaluation. Section 5 discusses related work. The paper is concluded in Section 6.

2 Background

2.1 Instruction scheduling

Instruction scheduling is a compiler optimization that aims at improving the instruction-level parallelism of a program. It reorders program instructions to better utilize processor resources (e.g., pipelined units and multiple issue capabilities), and is thus subject to constraints related to platform characteristics as well as program data and control dependences.

We illustrate the effect of instruction scheduling using a simple example, where we consider a single-issue pipelined processor whose instructions take two cycles to complete. In this case, if two consecutive instructions have a data dependency, the second instruction needs to stall. Listing 1.1 shows a simple code sequence consisting of five instructions. The corresponding direct acyclic graph (DAG) that represents the data dependences between the instructions is shown in Figure 1. This initial code sequence requires nine cycles to execute. I2 needs to stall waiting for I1 to load data from memory into register \$1. Similarly, I4 needs to wait for I3 to complete. However, if I3 is moved upwards in the code sequence, the two stalls are eliminated. The resulting code sequence, shown in Listing 1.2, takes seven cycles to execute.

Listing 1.1: Code sequence example

```
I1: ld  $1,0($5)
I2: mul $1,$2,$1
I3: ld  $3,4($5)
I4: add $4,$3,$1
I5: st  $4,8($5)
```

Listing 1.2: Code sequence after instruction scheduling

```
I1: ld  $1,0($5)
I3: ld  $3,4($5)
I2: mul $1,$2,$1
I4: add $4,$3,$1
I5: st  $4,8($5)
```

There are several types of instruction scheduling, depending on what parts of the program we focus on. For instance, global code scheduling refers to code motion across basic block boundaries. Global scheduling algorithms take into consideration both data and control dependences. Code motion can be upward or downward, and speculative or non-speculative. However, speculative instructions must not have any unwanted side effects [3].

On the other hand, instruction scheduling can also be applied locally, i.e., within a basic block boundary. A basic block is a sequence of consecutive instructions that has a single entry point and contains no jump instructions, except possibly the last instruction in the block. In this work we are concerned with basic block scheduling.

2.2 List scheduling

Optimal scheduling of a basic block is an NP-complete problem. *List scheduling* [3] is a simple approximation algorithm that works well in practice. The algorithm's input is a sequence of instructions in a basic block, a data dependence graph and platform-related information (e.g., instruction latencies and pipeline units). Its output is a, possibly altered, valid sequence of instructions.

The algorithm visits the nodes (instructions) in the data dependence graph based on a prioritized topological order. The topological order is computed using a heuristic priority function. Examples of such priority functions are: the height of the node in the graph (critical path), resource usage metrics, source program ordering, or a combination of the above. Subsequently, the algorithm computes the earliest time slot in which each node can be scheduled, considering its dependences with previously scheduled nodes. Finally, the node is scheduled in the earliest time slot with sufficient available platform resources.

2.3 Instruction scheduling in GCC

The default instruction scheduler of the GCC compiler is the Haifa Scheduler [9]. Instruction scheduling in the Haifa Scheduler is performed twice, once between flow analysis and register allocation and once after register allocation. For each function, a control flow graph is constructed and the function is split into regions, which can be either a reducible inner loop, a loop-free procedure, or a single basic block. For each region, control flow attributes and data dependences are computed. Each basic block of the region is scheduled after its flow predecessors by calling the `schedule_block()` function.

For basic block scheduling, GCC implements a variation of the list scheduling algorithm. Algorithm 1 shows a simplified version of GCC's implementation. During the scheduling task, an instruction is either *pending*, *queued*, *ready* or *scheduled*. A *pending* instruction is an instruction with unmet dependences, a *queued* instruction has met dependences and can be scheduled after a sufficient amount of time, and a *ready* instruction is ready to be committed.

Initially, all instructions are classified as either *pending* or *ready*, by the `init_ready_list()` function, which adds to the ready-list *ready* instructions from all basic blocks of the region. As (virtual) time passes, instructions from the ready-list are chosen to be scheduled by the `choose_ready()` function. As these instructions are moved to the *scheduled* list, dependent instructions are moved to the *ready* or *queued* sets.

GCC uses a list of criteria in order to calculate priorities for the *ready* instructions. First, instructions are ranked by the highest criterion. If two instructions have equal rank, the next criterion is used until the tie is settled. The list of criteria used is the following:

1. the longest path from the instruction to the end of the basic block (critical path)
2. contribution to register pressure
3. in-block versus interblock motion
4. useful versus speculative motion
5. control flow probability
6. number of dependences upon the previously scheduled instruction
7. number of forward dependences
8. program order.

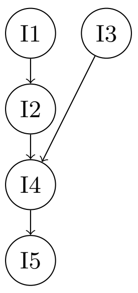

Fig. 1: Data dependences for sample code sequence

Algorithm 1 A simplified version of basic block scheduling in GCC

```
schedule_block()
{
    clock_var = -1;
    init_ready_list();

    while (!finished()) {
        clock_var++;
        /* Advance DFA state. */
        advance_one_cycle ();

        /* Add to the ready list all pending
         * instructions that can be issued now. */
        queue_to_ready (&ready);

        /* Sort the ready list. */
        ready_sort();

        /* Target-specific reorder. */
        target_reorder_ready_list();

        for (;;) {
            if (ready.n_ready == 0 &&
                                 can_issue_more) {
                /* Allow scheduling instructions
                 * directly from the queue. */
                early_queue_to_ready();
                ready_sort();
            }

            /* Choose instruction to schedule. */
            insn = choose_ready();

            /* Consult the DFA. */
            cost = tmp_state_transition(insn);
            if (cost >= 1) {
                queue_insn(insn, cost);
                continue;
            }

            /* Schedule instruction and move all
             * instructions with resolved
             * dependences to the ready list. */
            schedule_insn(insn);
            break;
        }
    }
}
```

The ordering of these criteria is merely a heuristic. For some specific application or architecture one of the criteria that are low on this list can be more important than a higher one, thus this heuristic will be suboptimal for this case.

Moreover, before an instruction is chosen from the ready list to be scheduled, the backend is allowed to perform a processor specific reordering of the “ready” list, by means of a target-specific hook.

3 Randomized list scheduling

3.1 Instruction scheduling as an optimization problem

Our approach can be viewed as an optimization problem, where our goal is to find a schedule with better performance than the default produced by GCC. The search space of the problem is the set of valid instruction schedules, and the cost function is the performance of each instruction schedule on real hardware. Instead of modifying the compiler heuristics for instruction scheduling, we explore the solution space using search methods based on randomness.

First, we modify GCC to produce schedules different from the default, while still respecting dependences between instructions, thus maintaining the validity of the resulting schedules. Ideally, this would only require a minor modification in the GCC’s implementation of the list scheduling algorithm. All instructions in the ready list have met dependences, and so it is obviously valid to schedule each one of them next. Therefore, we modify GCC’s block scheduling algorithm to choose arbitrarily any instruction from the sorted ready list, instead of always choosing the first.

The set of all possible scheduling choices can be represented by a tree, where each branch corresponds to the choice of an instruction, and each path from the root corresponds to the resulting instruction schedule. In the case of an internal node the schedule is partial, while for a leaf node the schedule is final.

We can use various tree traversal algorithms to produce different instruction schedules, as we are exploring its nodes. Each algorithm corresponds to a method of choosing the next instruction to schedule. By experimenting with different methods, we can produce schedules that are positioned close or far from GCC’s default.

In this work we explore the decision tree according to three different methods by explicitly instructing the compiler to make different scheduling choices. For each of the architectures we experiment with, we produce a constant number of schedules using each method. We choose the best schedule by profiling the schedules on real hardware. This way we are able to determine whether exploring the search space can actually yield performance improvements for each architecture, and which search algorithm leads to the production of better instruction schedules.

3.2 Search methods

GCC’s default instruction scheduler always chooses the first instruction in the (sorted by rank) ready list. When modifying the instruction scheduling algorithm in order to produce different schedules, there are various possibilities regarding the choice of the next instruction to be scheduled. We could, for example, instruct the compiler to always choose between the first two instructions based on given probabilities. Other possibilities include various discrete probability distributions, such as uniform distribution. Allowing more choices in the scheduler results in an augmented search space. This constitutes a trade-off: a large search space may contain better solutions, but it is more expensive to explore.

To evaluate the effects of inserting randomness into the instruction scheduling algorithm, we produce a relatively large, constant number of instruction schedules. There is, however, the possibility that two of the schedules produced might end up to be identical. In order to avoid wasting resources in this manner, we keep in-memory information about the past scheduling choices made for distinct compiler invocations. We use a tree to represent this information. As mentioned before, leaf nodes represent full schedules, internal nodes represent partial schedules and edges represent scheduling choices (specific instructions).

During each separate compilation of the program we move downwards, towards the leaves of the tree. Nodes are added as necessary, according to the length of the ready list. When subtrees originating from a node have been fully explored, we mark the parent node as fully explored to avoid making the same combination of choices again. In this paper, we experiment with three search methods: DFS (Depth-First Search), BFS (Breadth-First Search) and Uniform.

In the DFS algorithm (Algorithm 2), the next instruction to be scheduled is always the first instruction in the ready list that contains unvisited children. Therefore, this search method always produces the GCC's default schedule first, and generally favors schedules relatively close to the default, as it first explores branches that have many identical scheduling choices.

Algorithm 2 Pseudocode for DFS search algorithm

```
dfs_choose_next(Node current) {
    for i in current.children:
        if not i.fully_explored():
            return i
}
```

In the BFS algorithm (Algorithm 3), the next instruction to be scheduled is the first that has never been selected before. Even though this search method, similarly to DFS, always produces the default schedule first, it generally results in scheduling choices that are significantly different from the default. Hence, it favors a more aggressive exploration of the scheduling tree.

Algorithm 3 Pseudocode for BFS search algorithm

```
bfs_choose_next(Node current) {
    for i in current.children:
        if i.never_selected():
            return i
    for i in current.children:
        if not i.fully_explored():
            return i
}
```

In the Uniform search algorithm (Algorithm 4), the next instruction to be scheduled is selected by a uniform random distribution over all instructions in the ready list, excluding those that correspond to subtrees that have been fully explored. This search method's behavior is expected to resemble more the behavior of BFS rather than DFS, as it also favors exploring schedules that are positioned further away in the decision tree. It results, however, in a more aggressive exploration of the tree than BFS.

Algorithm 4 Pseudocode for Uniform search algorithm

```
uniform_choose_next(Node current) {
    unexplored_children=[]
    for i in current.children:
        if not i.fully_explored():
            unexplored_children.append(i)
    // pick with uniform distribution
    return pick_random_node(unexplored_children)
}
```

3.3 Production of unique schedules

One important aspect in the process of randomly creating schedules is ensuring that a significant number of unique schedules is produced. Even though we keep track of past scheduling choices, there is always a possibility that some of the resulting schedules are identical. There are several reasons for this.

First, GCC performs two separate instruction scheduling passes, which can be viewed as permutations of the instruction sequence (under several constraints). Similarly, the whole instruction scheduling procedure can be viewed as the composition of these two permutations. Therefore, distinct choices during the two passes can lead to the same final schedule, as the compositions of two distinct pairs of permutations can lead to the same sequence.

This observation motivated some initial experimentation regarding which of the two scheduling passes need to be modified. In this paper, we present results for the case with the best average performance: applying randomization to both passes. Note that we apply the same search algorithm (e.g., DFS) on both passes.

For the Uniform search method, we did not keep track of the random scheduling choices made during the second pass. Due to the nature of the Uniform search, keeping track of the choices made in the first pass was enough to ensure all produced schedules were unique. On the other hand, keeping track of choices made in the second pass would only introduce unnecessary overhead.

Another reason why distinct scheduling choices can lead to the same final schedule is the fact that GCC enquires a DFA pipeline hazard recognizer [13] before issuing an instruction on a given cycle. This pipeline hazard recognizer is constructed automatically from the backend machine description file, where the processor's pipeline properties are described (e.g., pipeline structure, structural hazards, instruction delays). In particular, after an instruction is selected, the DFA is consulted to decide whether the selected instruction should be actually scheduled. For the architectures we experimented with, which don't impose resource constraints on the schedules, deactivation of this feature retains the validity of the schedules, while it leads to an increase in the unique schedules produced. However, these schedules exhibit worse average performance. Therefore, we decided to retain the default behavior of GCC, effectively pruning the search space.

Identical schedules may also arise due to subsequent optimization passes that further reorder instructions. For example, in some architectures GCC performs scheduling of branch delay slots. Preliminary experiments also showed that although deactivation of this optimization lead to a larger number of unique schedules, the performance of these schedules was worse than the default case. Therefore, in our experiments we leave this optimization enabled.

As generation of duplicate schedules can not be completely eliminated, we need to avoid the overhead of executing identical schedules more than once. For this purpose, we create a checksum (SHA1) for each resulting schedule, and check against previously executed versions before execution.

3.4 Implementation

The version of GCC we work with is 4.5.2. This version of GCC supports plugins, which make the process of adding new features to the compiler easier. Plugin code can be compiled separately from the rest of the compiler code, speeding up the building and testing time. Plugins are loaded dynamically when the compiler is executed, using the `-fplugin=`*`name`* flag.

The plugin API provides functions for interacting with the pass manager [9]. By means of this plugin, we instruct the pass manager to replace the default GCC instruction scheduling pass with the modified scheduling pass that implements randomized list scheduling. A Python script acts as a wrapper to the plugin, actually implementing the tree traversal algorithms and communicating with the plugin to produce the instruction schedules.

4 Experimental evaluation

4.1 Experimental setup

We perform experiments on the following platforms:

- PlayStation 3 (Cell SPU)
- Intel XScale IOP342 (ARM), evaluation board IQ81342SC
- Sun Niagara 2 (Ultrasparc T2)
- Intel Xeon Harpertown (x86_64).

Benchmark	**Description**
mult	Matrix multiplication
adv	Stencil solver for the advection PDE
fw	Floyd-Warshall algorithm
fw (int)	Floyd-Warshall algorithm (integer)
liv6	Livermore kernel number 6 – general linear recurrence equations
liv11	Livermore kernel number 11 – first sum
liv12	Livermore kernel number 12 – first difference
liv20	Livermore kernel number 20 – discrete ordinates transport
inner	Livermore kernel number 3 – inner product
trid	Livermore kernel number 5 – tri-diagonal elimination, below diagonal

Table 1: Benchmarks used in experiments

Benchmark	**XScale**			**SPU**			**Niagara 2**			**Harpertown**		
	bfs	**dfs**	**uni**	**bfs**	**dfs**	**uni**	**bfs**	**dfs**	**uni**	**bfs**	**dfs**	**uni**
mult	27	55	100	100	52	100	25	79	100	14	45	100
adv	19	49	100	100	97	100	100	60	100	12	55	100
fw	07	02	100	100	23	100	11	70	100	27	36	100
fw (int)	46	37	100	100	25	100	11	63	100	07	39	100
liv6	19	53	100	100	36	100	100	70	100	12	20	100
liv11	14	54	100	100	16	100	100	84	100	12	19	100
liv12	43	46	100	100	16	100	100	84	100	12	17	100
liv20	31	44	100	100	02	100	100	56	100	12	67	100
inner	52	58	100	100	38	100	100	82	100	12	18	100
trid	14	60	100	100	32	100	100	84	100	12	18	100
mult (vec)	-	-	-	100	83	100	-	-	-	-	-	-
adv (vec)	-	-	-	37	40	100	-	-	-	-	-	-
fw (vec)	-	-	-	100	58	100	-	-	-	-	-	-

Table 2: Percentage (%) of unique schedules out of 500 produced schedules

The XScale processor lacks hardware support for floating point operations. Floating point instructions are emulated in kernel via illegal instruction faults.

Table 1 contains a short description of the benchmarks we use for the evaluation of our approach. We experiment on relatively small loops, because it allows us to analyze the produced assembly code and determine the specifics of performance behavior. Most of these benchmarks are taken from the Livermore loops [2]. For the Cell SPUs, we also used vectorized versions for three of the benchmarks.

For each method we produce 500 schedules, compiling with the `-O2` and `-funroll-loops` GCC flags enabled. Table 2 shows the percentage of unique schedules generated for each case. We execute each unique schedule and calculate the performance improvement (speedup) compared to the GCC default schedule, which was compiled with the same optimization flags.

4.2 Performance evaluation

Figures 2, 3, 4 and 5 show minimum, maximum and average speedups for each benchmark for the XScale, Harpertown, Niagara 2 and Cell SPU architectures respectively. As expected, the performance improvement of our method in the Harpertown processor is marginal (Figure 3), because Harpertown implements sophisticated out-of-order execution. For the XScale, on the other hand, our approach significantly improves performance in most cases (Figure 2): it results in a 33% speedup for the *liv12* benchmark, and a 13.3% and 10.8% average speedup over all benchmarks for Uniform and BFS, respectively. The big differences in the average speedup for the architectures we experimented with might also imply that GCC's modeling of the architecture could be improved for the architectures exhibiting the greater speedups.

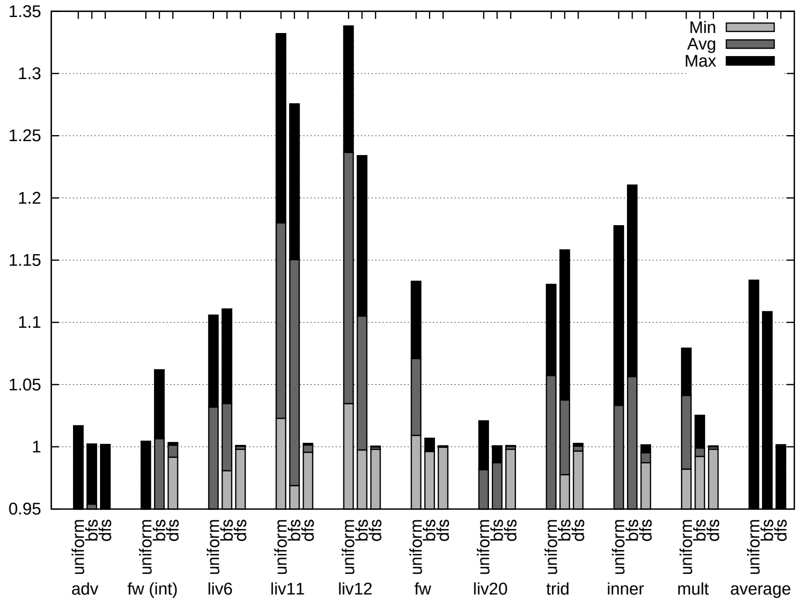

Fig. 2: Speedups for XScale

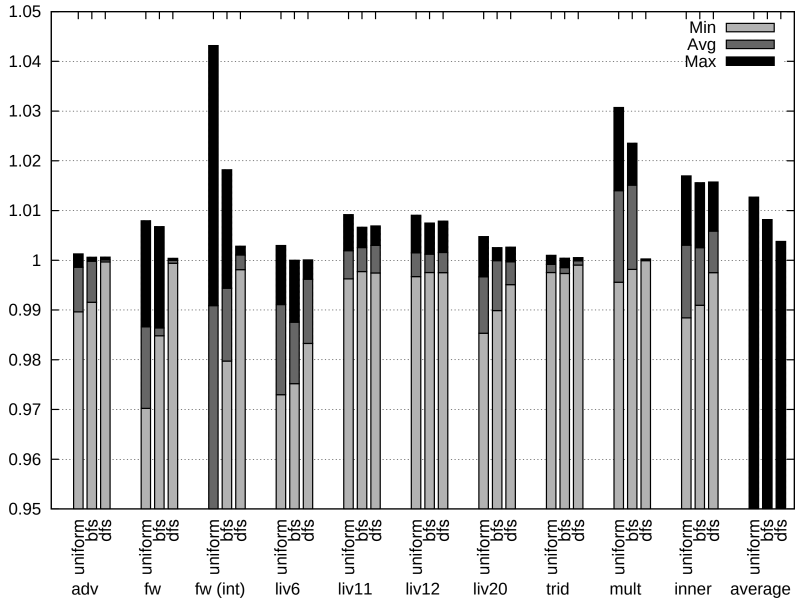

Fig. 3: Speedups for Harpertown

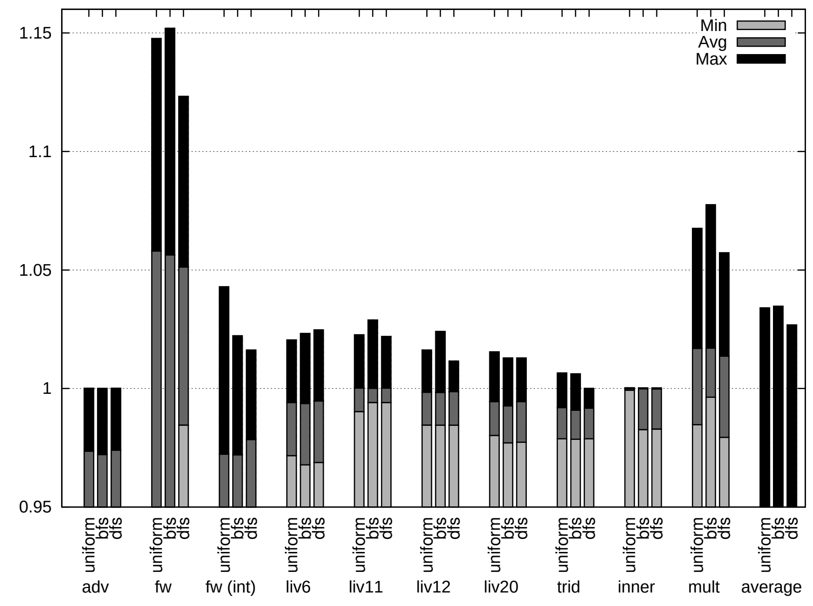

Fig. 4: Speedups for Niagara 2

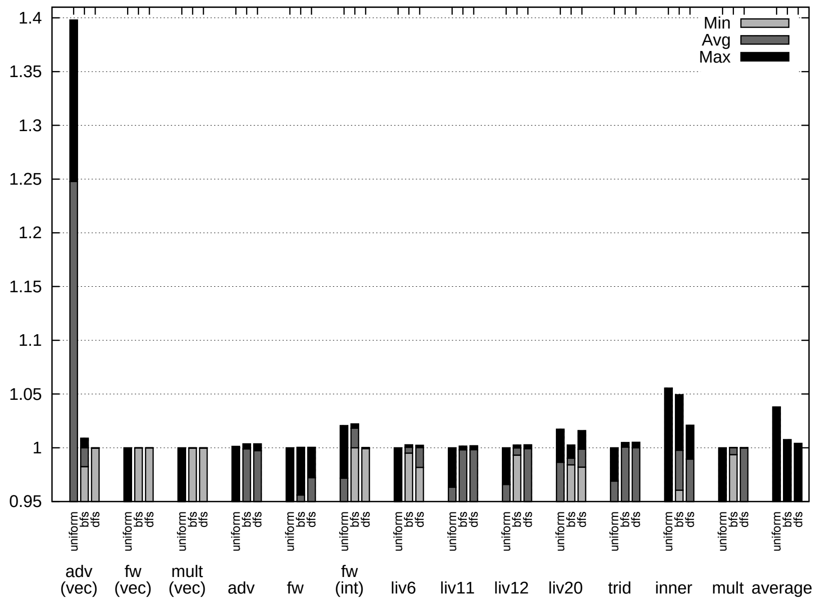

Fig. 5: Speedups for Cell SPU

In general, the Uniform method achieves the best results. Contrarily, the DFS method achieves the lowest speedups, but maintains an average speedup close to 1.0. The BFS method is generally better than the DFS method, and in some cases (e.g., Niagara 2) supersedes the Uniform method.

A general conclusion that can be drawn from the results is that schedules which improve performance, when they exist, are located "far" from the default schedule of GCC. Therefore, the DFS method, which explores scheduling choices close to the default, fails to achieve noticeable performance improvement.

It is also important to note that whereas some benchmarks seem to have significant room for improvement, others don't. In most cases where significant maximum speedups are achieved by this approach, we can also note that the average speedup among all 500 produced schedules is also quite high. This implies that we can probably estimate whether it is meaningful to further explore the search space by producing a smaller number of schedules at first, and only continuing if the preliminary results seem positive.

5 Related work

Metaheuristics [12] have been extensively used in optimizing compilers. A recent example is the MILEPOST GCC [10], a machine-learning based compiler that aims to predict the best optimization flags and sequence of optimization passes to automatically improve the execution time, code size, or compilation time of specific programs on different architectures.

Metaheuristics have also been used in the past to improve instruction scheduling in particular: Beaty [4, 5] proposes the application of Genetic Algorithms in order to explore the solution space of valid instruction schedules. The author follows two approaches, one of which is based on list scheduling. This work does not include an experimental evaluation, but it is mainly concerned with complexity analysis of the algorithms. In [11], Genetic Algorithms are applied to the combined problems of instruction scheduling and register allocation, with profits on performance that vary between -2% and 26%. In [16], another approach based on Simulated Annealing is described, but it requires excessive overhead. [14] presents an application of machine learning techniques in order to automatically obtain a heuristic scheduling algorithm for a given architecture. Once the heuristics are produced, they are applied to each application for this architecture. [6] applies supervised learning to induce heuristics for predicting whether each basic block will benefit from instruction scheduling, thus applying scheduling only to a subset of the basic blocks.

In [15] a different approach to randomizing list scheduling is followed. Instead of randomizing all scheduling choices, only the breaking of ties in the list scheduling algorithm is randomized, creating a search space of schedules, which is however limited and relies on the assumption that the heuristic used for sorting the ready list performs well. This is applied to both forward and backward list scheduling. In the same thesis, another randomized scheduling algorithm, based on iterative repair scheduling, is proposed. In our work, we tried instead to expand the search space of schedules further, without proposing a major algorithmic change.

In [7] an improved instruction scheduling algorithm is proposed that targets the Explicit Data Graph Execution (EDGE) processor architecture. Simulated Annealing is used to estimate the optimal schedule performance and the results are used to develop more elaborate heuristics for the proposed scheduling algorithm. [8] extends the aforementioned work by introducing a hierarchical approach to machine learning for instruction scheduling. Features extraction is used to classify segments of code with similar characteristics and reinforcement learning is employed to learn heuristics for these classes. Both of these works target the EDGE architecture, while the focus of our work is on improving instruction scheduling for commodity architectures in an existing compiler infrastructure.

6 Conclusions – Future work

In this paper we propose the use of randomness to improve instruction scheduling. Inserting randomness in GCC instruction scheduling yields significant speedups for most benchmarks on XScale, and for some benchmarks on Cell SPU and Niagara 2. For Harpertown, instruction scheduling seems to be much less important, which is expected as the processor used has sophisticated out-of-order execution capabilities. The best results seem to be reached by the Uniform search method. The BFS method follows relatively closely the results of the Uniform method, also achieving significant speedups. The DFS method achieves smaller speedups, due to making scheduling choices closer to the default.

Although our approach can achieve significant improvements, it also introduces significant overhead. This extra work is, however, fully parallelizable, therefore splitting the production and/or profiling of instruction schedules among many cores is a straightforward way to reduce the overhead. Moreover, our approach is meaningful only for the cases where performance really matters and the cost of the extra work will be alleviated by a great decrease in program execution time, for example for programs that have long running times.

As future work, we plan to further research into reducing this overhead, as well as into implementing more efficient search algorithms. Our long-term goal is to develop a full system that will be able to automatically improve performance by exploiting this approach. Ideally, such a system would support two alternatives regarding the application of randomization: statically, at compile time, or adaptively, at runtime.

References

1. Ibm Assembly Visualizer for Cell Broadband Engine, http://www.alphaworks.ibm.com/tech/asmvis.
2. Livermore loops, C version, http://www.netlib.org/benchmark/livermorec.
3. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*, chapter 10. Pearson/Addison Wesley, second edition, 2007.
4. Steven J. Beaty. Genetic algorithms and instruction scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO-24)*, 1991.
5. Steven J. Beaty. Genetic algorithms for instruction sequencing and scheduling. In *Workshop on Computer Architecture Technology and Formalism for Computer Science Research and Applications*, 1992.
6. John Cavazos and J. Eliot B. Moss. Inducing heuristics to decide whether to schedule. In William Pugh and Craig Chambers, editors, *PLDI*, pages 183–194. ACM, 2004.
7. Katherine E. Coons, Xia Chen, Doug Burger, Kathryn S. McKinley, and Sundeep K. Kushwaha. A spatial path scheduling algorithm for EDGE architectures. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (12th ASPLOS'06)*, San Jose, CA, 2006.
8. Katherine E. Coons, Behnam Robatmili, Matthew E. Taylor, Bertrand A. Maher, Doug Burger, and Kathryn S. McKinley. Feature selection and policy optimization for distributed instruction placement using reinforcement learning. In *PACT*, 2008.
9. Free Software Foundation. GNU Compiler Collection (GCC) Internals. Technical report, Free Software Foundation, 2008.
10. Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, Chris Williams, and Michael O. Boyle. MILEPOST GCC: machine learning based research compiler. In *GCC Summit '08*, pages 1–13, 2008.
11. Fernanda Kri and Marc Feeley. Genetic instruction scheduling and register allocation. In *SCCC*, pages 76–83. IEEE Computer Society, 2004.
12. Sean Luke. *Essentials of Metaheuristics.* 2009. available at http://cs.gmu.edu/simsean/book/metaheuristics/.
13. Vladimir N. Makarov. The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC. In *Proceedings of the GCC Developers Summit May 25–27, 2003, Ottawa, Ontario Canada*, 2003.
14. J. Eliot B. Moss, Paul E. Utgoff, John Cavazos, Doina Precup, Darko Stefanovic, Carla E. Brodley, and David Scheeff. Learning to schedule straight-line code. In *NIPS*. The MIT Press, 1997.
15. Philip Schielke. *Stochastic Instruction Scheduling.* PhD thesis, November 13 2000.
16. Philip H. Sweany and Steven J. Beaty. Instruction scheduling using simulated annealing. In *Proceedings of the 3rd International Conference on Massively Parallel Computing Systems (MPCS '98)*, 1998.