Improving selective scheduler approach with predication and explicit data dependence support

Dmitry Melnik, Alexander Monakov, Andrey Belevantsev,

Tigran Topchyan, and Mamikon Vardanyan *

Abstract

The global instruction scheduler and software pipeliner implemented by ISP RAS for the GCC compiler based on the selective scheduler has some deficiencies. Among some of the important ones we can name the absence of predication support for IA-64 and the usage of implicit data dependencies (i.e., no data dependence graph (DDG) is constructed and supported in the process of scheduling, but rather the elementary operation of moving an instruction up through another one is supported). This paper describes how we deal with both deficiencies, first, by adding predication support to the scheduler as yet another variant of an instruction transformation, which is a novel contribution of the paper, and second, by suggesting the explicit DDG construction and maintenance algorithm within the created scheduling framework in the GCC compiler,

*Institute for System Programming of Russian Academy of Sciences, {dm, amonakov, abel, tigran, mamikon}@ispras.ru

which is also our contribution and the novel improvement of the original selective scheduling approach.

1 Selective scheduler in GCC

The selective scheduler [7] is a top-down scheduler operating on arbitrary acyclic regions of code and supporting several scheduling boundaries. For each boundary, it computes a set of available instructions by walking a region in reverse topological order, heuristically chooses the best instruction among them, and then moves it from its original location (or multiple original locations, if those are found while traversing code motion paths) to the current scheduling boundary, generating bookkeeping copies on the fly if needed. The scheduling boundary is then moved down (and split into two boundaries on branches) and the process is repeated until all instructions from the region are scheduled.

To overcome dependencies, selective scheduling supports several instruction transformations. It can perform substitution through register copies to overcome true dependencies and register renaming to overcome anti- and output dependencies. On IA-64, implementation for GCC also generates control and data speculative loads to overcome control dependencies and some of the memory true dependencies [2]. Substitution and speculation can be named local transformations, that is, they are applied for breaking a dependency immediately after it is found during available instructions' computation, while register renaming and bookkeeping are global ones (require information about possible code motion

paths to be correct).

The selective scheduler can also perform software pipelining on cyclic regions by allowing code motion from already scheduled code. This is achieved by moving instructions across a loop back-edge and generating bookkeeping code in a loop preheader, thus forming the prologue of the newly pipelined loop. The number of loop iterations is not preserved, so control speculation support is needed to pipeline a memory load. The original algorithm also generates the epilogue by moving up conditional jumps that exit a loop and effectively pushing some instructions (that a jump was moved up through) down to both targets of the jump including a loop exit, but this transformation is not implemented in GCC. More details about GCC implementation specifics can be found in [2], [5], [3], and [4].

2 Predication support in the selective scheduler

2.1 Conditional execution on IA-64

Itanium architecture features full predication support, which means that most instructions may be annotated with a 1-bit predicate register, and execution of the predicated instruction will affect processor state iff its predicate register evaluates to 1 at runtime. Some other processor architectures provide partial support for conditional execution, for example by providing conditional moves.

Support for conditional execution allows to eliminate branches, which results in smaller code and reduces pipeline flushes from mispredicted branches. It also helps to increase us-

age of available execution units, and allows to start long-latency instructions (for example, loads from memory) earlier. However, careful compiler implementation is essential, because converting too much instructions into predicated form may result in lower quality code because of resource over-subscription.

2.2 GCC implementation

We present an approach to transform instructions into predicated form during instruction scheduling pass. This approach benefits from knowledge of unused processor units, to which predicated instructions may be assigned. The implementation is done in the selective scheduler, where it fits neatly as an additional form of instruction transformation.

Instructions in predicated form are added into availability sets when merging the sets on conditional branches, contrary to the other local transformations happening when a set is moved up through an instruction. The reason for this is that we need to know the branch target that originated the instruction so that we can predicate the instruction with the correct condition. For the same reason, predication support does not use the local transformation cache that is implemented to speed up the computation phase. Instead, a separate cache is implemented that can be queried using an instruction-condition pair, so that instruction forms with both the initial predicate and its inversion can be conveniently stored. Also, to propagate the newly predicated instructions further up, the dependency analysis of the scheduler is modified to allow moving predicated instructions through conditional jumps with the same or inverted predicate (even for possibly trapping instructions

like loads), unless the predicated instruction modifies the predicate register that guards the jump.

The selective scheduler performs code motion stage to generate necessary bookkeeping copies and to update the available instructions and other dataflow information (like register liveness). We have made three modifications to support predication on this stage. First, we are traversing code motion paths while looking for instructions that could generate the chosen one. When we search for a predicated instruction and we process a conditional jump guarded by the same predicate register, we need to proceed with the search only down the branch target that satisfies the condition of the predicated instruction. This follows from the fact that the instruction will never be executed down on the branch target with the inverted predicate.

Second, during code motion the same local transformations as when computing availability sets should be performed, as bookkeeping copies should be created out of the correct instruction form. As predication does not happen together with other local transformations, we cannot use the regular propagation engine for this purpose as we did previously. Thus, we have implemented a separate routine for reapplying local transformations. It reuses the same transformation history as another function for undoing transformations, which was implemented earlier to speed up moving the chosen instruction down from scheduling boundary while searching for original instructions. Predication support for history management functions was also implemented.

Third modification is required because of the bookkeeping code generation process. The

original selective scheduling approach finds all original instructions of the one chosen for scheduling and removes them, also including bookkeeping copies possibly generated during the same code motion process. This is required for correctness, because the scheduler doesn't know the places where bookkeeping copies would be generated, but rather it creates them on the fly at any control flow join point it traverses, relying on the unnecessary copies' removal to avoid executing the chosen instruction more than once on some control flow paths. When the chosen instruction is predicated, its original instructions found during code motion process should not be removed, but each of them should rather be predicated with the inverted condition of the chosen instruction. In addition, we must ensure that the predicate register is not clobbered along the code motion paths between the scheduling point and the original instruction(s). Currently, tracking availability of predicate registers is not implemented, so we disallow moving predicated instructions through jumps predicated with different registers to retain correctness.

As predication does not affect live ranges of registers, the merging routine is also modified so that the availability of the target register in predicated assignments is inherited from the original non-predicated instruction. Correctly storing this information allows scheduling the assignment originated from the certain branch target before scheduling the branch itself without renaming the target register even in the situations when the register is live on the other branch target.

The implementation allows arbitrary interaction with other transformations. First, predicated instructions may be subject to substitution when moving up through a register

copy that is executed either unconditionally or guarded with the same condition. Second, the target register of a predicated instruction can be renamed. This is achieved by making a predicated instruction *separable*, i.e. having a left-hand side and a right-hand side, which are assigned equal to the corresponding sides of the original instruction, of course only in case when the original unpredicated form is also separable. Third, predicated memory loads may be speculated in case they are being moved through more than one conditional branch.

Predication support also improves pipelining quality of the selective scheduler in several ways. First, predication allows avoiding unnecessary speculation when a memory load is pipelined (i.e. moved up along a loop back edge), thus removing the need of creating speculative check instructions. Second, unnecessary renaming is also avoided as when the target register of an instruction is live on a loop exit, the instruction predicated with the inverted condition of this exit does not destroy the contents of this register. Third, pipelining instructions via predication will not result in unnecessary code execution, because on the last iteration of the loop the pipelined instruction predicate will be false, and this will account for the extra execution of its bookkeeping copy in the loop prologue.

Preliminary testing was done on the SPEC CPU 200 benchmarks. The baseline optimization level was `-O3 -ffast-math`[1]. We have made two peak level runs, both of which enabled predication support, and one of the runs also had the *doloop* pass disabled. The doloop pass makes use of the `br.cloop` instruction, which does not take into account any

[1] `-O3` enables the selective scheduler on Itanium by default since GCC 4.4.

predicate registers, so predication cannot be used together with pipelining in this case. Disabling the doloop pass increases performance for the integer benchmarks and decreases performance for the floating point benchmarks.

The average performance increase is very slight in both cases (~0.5%), with notable speedups of `twolf` (1.3%), `swim` (2.6%), `galgel` (1.5%), and `sixtrack` (2.5%) with the enabled doloop pass, and of `eon` (2.5%), `twolf` (1%), `swim` (2.6%), `applu` (1.5%), `mesa` (1.6%), `facerec` (1.8%), and `sixtrack` (1.2%) with the disabled doloop pass. There is no significant degradations with the enabled doloop pass, however, there are degradations of `wupwise` (-3.2%) and `ammp` (-1.3%) with the disabled doloop pass. The main source of speedup is likely the ability to pipeline loops without transforming the loads into the speculative form. Disabling the doloop pass or making it play well with the predication support may be an option in case the performance degradations will be fixed.

3 Supporting explicit data dependence graph

The original selective scheduling approach [7] doesn't require an explicit data dependence graph, but rather relies on the process of incremental construction of availability sets, where on each step a decision is made locally on whether the given instruction can be scheduled before another in a certain form. However, there is a number of enhancements to the core selective scheduling algorithm, which effective implementation requires an explicit data dependence graph (DDG).

First, the heuristics used at each step of the algorithm to choose the best instruction can be improved. Currently, code transformations like register renaming or control speculation may increase the critical path length of the loop region when doing pipelining. This may result in a performance degradation when the processor stalls execution on a register move or on a speculative check instruction, waiting for the memory load of one of the arguments to complete. To prevent this, the effects of generating new instructions during aggressive code transformations should be estimated, which involves walking def-use chains to estimate how much the transformation would delay scheduling consumers of the original instruction. Also, scheduling priority of instructions can be calculated by using enhanced G^* [6] or speculative yield [8] heuristics, designed especially for interblock scheduling in presence of speculative transformations. To implement those, we also need traversing DDG to estimate the dependence tree height. Second, by design the selective scheduling algorithm can leave excessive register copy operations, which should be removed by a simple copy propagation pass at scheduling time. And third, the algorithm is pretty slow, since on each step it requires the local data dependence analysis between the current instruction on a code motion path and the previously computed availability set.

All described problems can be addressed by adding support for explicit data dependence graph (DDG). Given the large number of code transformations and new instructions generated during the selective scheduling, DDG should be updated incrementally with each such transformation. The problem can be formulated as follows. For the given acyclic program region the data dependence graph is constructed, which is a full bi-directional

dependency graph with nodes corresponding to program region instructions. Two nodes are connected with an edge iff corresponding instructions have a data dependence between them. Each edge is attributed a dependence type, parts of instructions that have a dependence and the register (or memory location) that causes a dependence. For each of valid code transformations the selective scheduling applies (instruction move, unification of expressions, generation of bookkeeping, forward substitution, register renaming, data and control speculation, predication) a corresponding transformation for the data dependence graph is applied so DDG is consistent at any scheduling point. Also, this dynamically updated DDG should include not only every scheduled instruction, but also those instructions not yet scheduled and residing in availability sets.

DDG transformations fall in two categories: those which are performed when computing availability sets, and those performed when moving instructions across basic blocks or emitting new instructions. Transformations from the first group are easier to implement, since most transformations performed at the stage of moving expression up to the scheduling point are aimed at removing dependencies (except for *unification*, when expressions coming from different control paths are merged into one), and the new dependencies introduced at this stage can be copied from the instruction the expression is moved through. E.g. when performing forward substitution of expression $a : r_2 * r_3$ through instruction $b : r_3 = r_5$, in the resulting expression $a' : r_2 * r_5$ all dependencies with instruction a by register r_3 (including the true dependence from b) are eliminated, and all dependencies with instruction b by r_5 are copied to a'.

The difficulty with transformations from the second group is that emitting new instructions with destination registers different from original ones (those are chosen arbitrarily from the set of free registers across code motion paths) may result in new anti or output dependencies with instructions not involved in the current transformation. E.g. generating bookkeeping code on one of incoming edges with the new destination register will introduce an anti dependence with instructions that were reading from this register. The same problem applies to moving instructions that write in memory. So when emitting instruction with the renamed destination register or moving instruction with a side effect to a basic block where this effect becomes exposed to different execution paths, we generate the appropriate dependencies with instructions on those paths that read/write the same register (or memory location). To generate such dependencies, we use a reverse lookup table indexed by a register number (or a memory location) containing region instructions that read (write) that location. Also, the memory alias graph derived from previous GCC optimizations is used.

While some scheduler transformations belong only to the first category (e.g. forward substitution), most require DDG transformations on both stages of scheduling. E.g. register renaming and data speculation involve removing data dependencies on the stage of availability set construction and generation of new dependencies when emitting a speculation instruction with the renamed destination register, recovery code, and changing register in the original instruction. The most complex part of the DDG update (creating and splitting nodes with partial transfers of dependencies, as well as searching for dependent nodes not

directly involved in transformation) happens only when actually scheduling the instruction, while supporting DDG nodes for expressions in availability sets comes pretty simple. Due to the paper size constraints we don't strictly describe all the DDG transformations in detail, but most DDG transformations are naturally derived from transformations done to expressions during computation of availability sets.

4 Conclusions

We have presented novel improvements to the selective scheduling approach, including predication transformation support and explicit data dependence graph support. The former improvement has been implemented in the GCC compiler, yielding moderate speedups on some of SPEC CPU 2000 tests. The DDG construction and update support has been developed and is in the process of implementing in GCC. We plan to finish the implementation soon and then to proceed with improving the scheduling heuristics as sketched in the previous section.

References

[1] Alfred Aburto's system benchmarks. Could be found at

`ftp://gd.tuwien.ac.at/perf/benchmark/aburto`

[2] Andrey Belevantsev, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik, and Dmitry Zhurikhin. An interblock VLIW-targeted instruction scheduler for GCC. In

Proceedings of GCC Developers' Summit, Ottawa, Canada, June 2006.

[3] Andrey Belevantsev, Maxim Kuvyrkov, Alexander Monakov, Dmitry Melnik, and Dmitry Zhurikhin. Implementing an instruction scheduler for GCC: progress, caveats, and evaluation. In Proceedings of GCC Summit 2007, Ottawa, Canada, July 2007, pp. 7-21.

[4] Arutyun Avetisyan, Andrey Belevantsev, and Dmitry Melnik. GCC instruction scheduler and software pipelining on the Itanium platform. 7th Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-7). Boston, MA, USA, April 2008.
`http://rogue.colorado.edu/EPIC7/avetisyan.pdf`

[5] Andrey Belevantsev, Dmitry Melnik, and Arutyun Avetisyan. Improving a selective scheduling approach for GCC. GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems, Brasov, Romania, September 2007.
`http://sysrun.haifa.il.ibm.com/hrl/greps2007/`

[6] C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B. R. Rau, and M. Schlansker. Profile-driven instruction level parallel scheduling with application to superblocks. In Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-29), Paris, 1996, pp. 58-67.

[7] Soo-Mook Moon and Kemal Ebcioğlu. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining. ACM TOPLAS, Vol 19, No. 6, pages 853–898, November 1997.

[8] R. A. Bringmann. Enhancing instruction level parallelism through compiler-controlled speculation. Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1995.