Static Speculation as Post-link Optimization for the Grid Alu Processor

Ralf Jahr, Basher Shehan, Sascha Uhrig, and Theo Ungerer

Institute of Computer Science
University of Augsburg
86135 Augsburg
Germany
{jahr,shehan,uhrig,ungerer}@informatik.uni-augsburg.de

Abstract. In this paper we propose and evaluate a post-link-optimization to increase instruction level parallelism by moving instructions from one basic block to the preceding blocks. The Grid Alu Processor used for the evaluations comprises plenty of functional units that are not completely allocated by the original instruction stream. The proposed technique speculatively performs operations in advance by using unallocated functional units.

The algorithm moves instructions to multiple predecessors of a source block. If necessary, it adds compensation code to allow the shifted instructions to work on unused registers, whose values will be copied into the original target registers at the time the speculation is resolved.

Evaluations of the algorithm show a maximum speedup of factor 2.08 achieved on the Grid Alu Processor compared to the unoptimized version of the same program due to a better exploitation of the ILP and an optimized mapping of loops.

1 Introduction

The Grid Alu Processor (GAP, see Section 3) was proposed to speed up the execution of single threaded sequential instruction streams. Compared to other designs, GAP multiplies the number of Functional Units (FUs) instead of entire cores. To configure it, a superscalar-like processor frontend loads a standard sequential instruction stream that is dynamically mapped onto an array of FUs by a special configuration unit. Execution speed is gained very much from the high level of parallelism supplied by the FUs. The main influences on the mapping process are control and data flow dependencies as well as resource conflicts caused by limited resources, which restrict the level of instruction level parallelism (ILP) that can be exploited.

The algorithm presented in this paper tackles this by moving parts of a basic block (source block) to one or more preceding blocks (target blocks). By this, results that might be required in the near future, e. g. after upcoming branches, are calculated speculatively on otherwise unused resources. At the time the reason for the speculation is resolved, these results are made visible by compensation instructions (if required).

As the GAP shall be able to replace a superscalar RISC or CISC processor and, hence, be able to execute the same binaries, no recompilation would be needed to make

M.R. Guarracino et al. (Eds.): Euro-Par 2010 Workshops, LNCS 6586, pp. 145–152, 2011.

use of it. To preserve this advantage, we suggest using a post-link optimizer to apply platform-dependent code optimizations because the source code of the program to optimize is not needed in this case. Therefore, static speculation has been designed for use in a post-link-optimizer, hence after instruction selection, register assignment, and scheduling[1].

Static speculation is able to handle all types of control flow independent from domination- or post-domination-relations or the number of the source block's predecessors. The only exceptions are basic blocks that are targets of indirect jumps. A binary analysis together with profiling of the application delivers information about the execution frequency of basic blocks that can be selected as candidates for the modification.

In the remainder of the paper we give an overview of related approaches in Section 2 followed by a brief introduction of the GAP as target processor in Section 3. The algorithm is described in Section 4 followed by an evaluation of its effects on the execution of selected benchmarks in Section 5. The paper is concluded in Section 6.

2 Related Work

The GAP is a unique approach and no other code optimizations are yet suggested for it. However, similar challenges arise in compilation for superscalar or VLIW architectures as well as in hardware design. This section gives an overview.

For VLIW architectures, trace scheduling [5] is used to expose parallelism beyond basic block boundaries; it is implemented e. g. in the Multiflow Trace Scheduling Compiler [9]. This compiler also moves instructions above splits in the control flow graph but does this only if compensation instructions are not necessary. Other scheduling techniques for speculation working on the level of superblocks have been introduced and evaluated by e. g. Bergmann [2] and Mahlke [10]. These techniques require sophisticated knowledge of the program to optimize and, therefore, cannot be applied as post-link optimizations.

Without giving details, Bernstein et al. [1] suggest for scalar and superscalar architectures moving single instructions speculatively. Similar work is done by Tirumalai et al. [14]. The main difference to the work presented here is that we try to move as many instructions of a basic block as possible or reasonable at one time which decreases the overhead caused by repeatedly executing analyses. Beyond this, we also cope with the duplication of instructions to execute them speculatively.

Similarities also exist with *tail duplication* (see e. g. [6]).We also try to expose parallelism by duplicating instructions but handle only the important parts of basic blocks. Hence, the program is not as heavily rewritten but the modification effort is even smaller.

As shown in Section 4 our algorithm also has parallels with software pipelining (e. g., Llosa [8]) because it can split a loop formed by a single block into two parts and rearrange them (i. e., a prologue is formed). Nevertheless, it does not reach the complexity of most algorithms for software pipelining because we assume that instructions

[1] Nevertheless, additional implementation effort arises from this and it can happen that the optimization performs not as well as if implemented directly in the compiler. Somehow this is the price to pay for not having access to the source code.

in blocks have already been scheduled. Accordingly, we do not try to divide the source block into equal blocks in terms of approximated execution time and support only one stage.

Regarding processor design techniques, out-of-order execution as implemented by scoreboarding [12,13] or Tomasulo's scheme [15] – both in combination with branch prediction – execute instructions speculatively, too. The hardware-effort needed to allow out-of-order execution is very high and adds new limitations e. g. for the issue unit of a processor as shown by Cotofana et al. [4].

Hence, the outstanding features of the algorithm presented here are its large number of instructions which can be handled in one iteration, its ability to handle different constellations of blocks independent of the number of the source block's predecessors or the domination and/or post-domination relation between a source block and its predecessors. Beyond this, it is a post-link optimization that uses only information available from the analysis of the binary file and profiling. This causes also the struggle to modify only small parts of the program with the aim of achieving maximal effects.

3 Target Platform: Grid Alu Processor

The Grid Alu Processor (GAP) has been developed to speed up the execution of conventional single-threaded instruction streams. To achieve this goal, it combines the advantages of superscalar processor architectures, those of coarse grained reconfigurable systems, and asynchronous execution.

A superscalar-like processor front-end with a novel configuration unit is used to load instructions and map them dynamically onto an array of functional units (FUs) accompanied by a branch control unit and several load/store units to handle memory accesses.

The array of FUs is organized in columns and rows. Each column is dynamically and per configuration assigned to one architectural register. Configuration and execution in the array is always from the top to the bottom, data can flow only in this direction. The rows of the array are used to model dependencies between instructions. If an instruction *B* is dependent of an instruction *A* than it must be mapped to a row below the row of *A*.

To be able to save configurations for repeated execution all elements of the array are equipped with some memory cells which form configuration layers. The array is quasi three-dimensional and its size can be written as `columns x rows x layers`. So, before clearing the array it is first checked if the next instruction to execute is equal to any first instruction in one of the layers. Then, in all cases, the new values of registers calculated in columns are copied to the register file at the top of the columns. If a match is found, the corresponding layer is set to active and execution continues there. If no match is found, the least recently used configuration is cleared and used to map new instructions. With this technique, the execution of loops can be accelerated very much because instructions do not have to be re-issued.

For a detailed description of the GAP please refer to Uhrig et al. [16] and Shehan et al. [11].

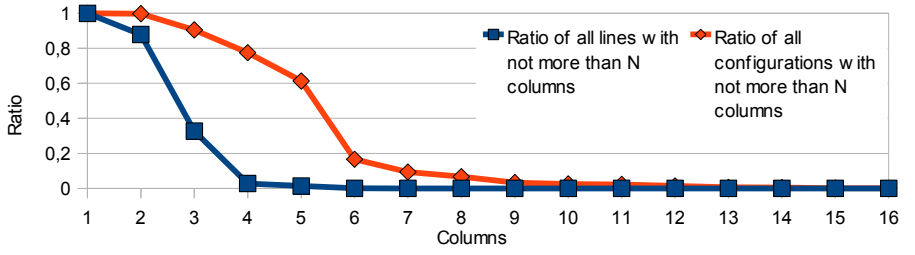

Fig. 1. Ratio of the configurations/rows which use the number of columns shown on the axis to the right for the benchmark `jpeg_encode` executed on GAP with 16x16x16 array

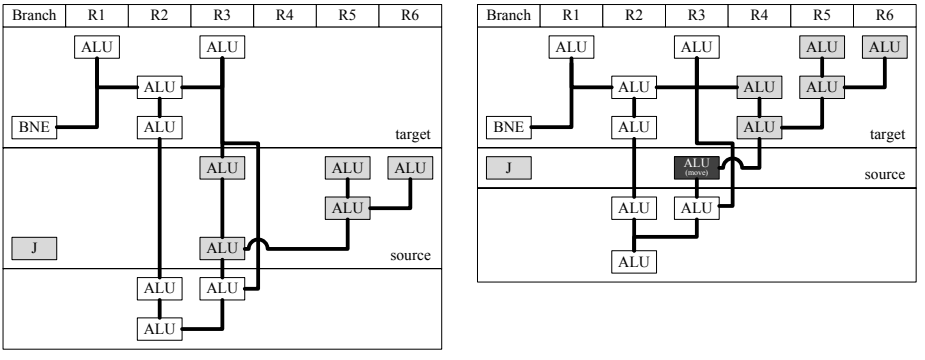

Fig. 2. Layout of three basic blocks in the GAP array without (left) and with (right) speculation

4 Static Speculation

In this section, we describe the algorithm to move instructions to preceding basic blocks together with the aim of the proposed program transformation.

When running code generated by the default compiler (GCC 2.7.2 with `-O3`, the latest version available for SimpleScalar/PISA), in most rows of the array only a small number of FUs is configured with instructions (see Figure 1). So there are enough FUs that could be used to calculate additional results, even if they might not be needed, because they would consume only little or no additional time. To use these spare resources we try to speculatively execute instructions from following blocks.

An example is shown in Figure 2. It shows three basic blocks which could stand for an *if-then*-structure and have been placed into the array of the GAP. The influence of data dependencies on the instruction placement can be observed. Also, after each control flow instruction synchronization, which is a special pecularity of GAP and shown by a horizontal line in Figure 2, is required. In this example, all instructions of the second block can be moved to the first block and executed speculatively.

If the second block shall not be executed, i. e., the branch from the first block to the third block is taken, its effects must not be visible. In the example, `R3` is the only

register which would have been modified by the speculatively executed instructions and overwrite a value which is used by subsequent instructions. Therefore, the moved instructions work on `R4`, which was initially unused, instead of the original `R3`. The overhead for executing the speculative instructions is zero in this example, because they are executed in parallel to the first block.

If the second block shall be executed, the content of `R4` must be copied to `R3`, the original target register. In the figure, this additional compensation instruction is marked by a dark box. Even if multiple compensation instructions would be required, they can be placed in the same row of the array and can be executed in parallel because they do not have any interdependencies. In our example, the overhead for the compensation instruction is zero because it is executed in parallel with the branch instruction. In the worst case, all compensation instructions can be executed in parallel because they do not have any dependencies on each other and, hence, consume the same time as a single additional *move* instruction.

Depending on the critical path and the resource utilization of the source and the target block, the number of rows of the array that are required to map the combined blocks should be lower. If a lower number of rows is needed for the modified blocks the average number of instructions per row increases. Hence, the number of instructions that can be executed in parallel increases resulting in a higher instructions per cycle (IPC) value.

To avoid executing too many unnecessary instructions, we move instructions only if the probability of the usage of the calculated results is above a fixed value, e. g. 30%. This value has been found to be a good tradeoff between performance and additionally executed instructions. Also we want to modify blocks only if they contribute significantly to the total program performance. Hence a block must be executed more often than a fixed boundary, e. g. more than 10 times.

Nevertheless, we cannot be sure that the total configuration length will be shorter after the modification of the blocks. This is because of the eventually required additional row for the compensation instructions, the potential inconvenient layout of the critical paths of the blocks, and resource restrictions. For example, if memory operations are moved into a block which already uses many memory access units, then additional rows will be needed to map the moved memory operations. Hence, resource restrictions can also restrict the degree of parallelism of instructions. This problem is solved by introducing an objective function to estimate the height of the modified configuration. The additional height could also be limited by a parameter, but currently it is set to zero.

This objective function is mainly taken into account when selecting the number of instructions to move. It is maximized in respect to the objective function and the availability of enough registers to use them as temporary registers.

A special case is to move instructions across a loop branch. As example, imagine a block with a conditional branch to its first instruction, so it forms a very simple loop. If we shift a part of the source block to all the preceding blocks, we shift instructions from its beginning to one or more blocks with edges to the loop and also to the end of the loop. these re-ordered parts of the loop can be executed with a higher degree of parallelism. Loop carried dependencies are also handled because the speculatively

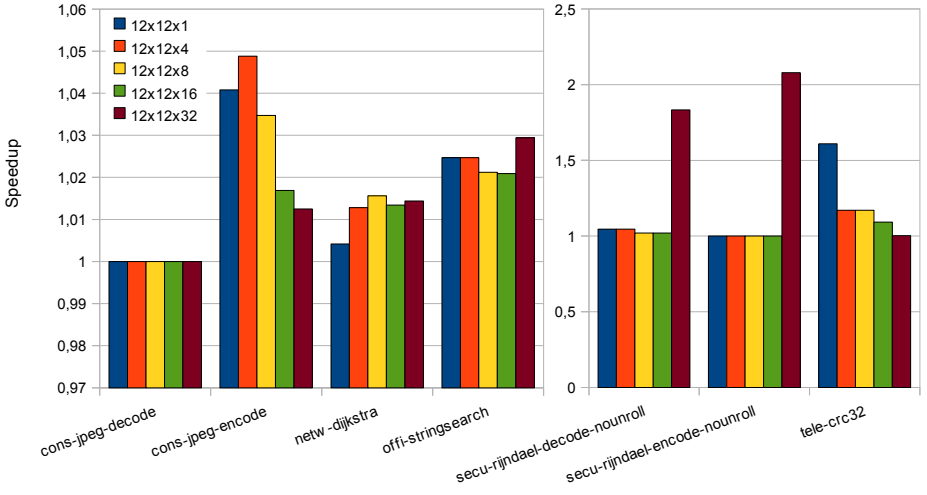

Fig. 3. Maximum speedup for selected benchmarks on GAP array of 12x12xN FUs

calculated results are not copied to the target registers until it is clear that the loop will be executed at least one more time.

To sum up, we expect better performance in terms of execution time. In an optimal case a better use of the FUs of the array is achieved because more columns and less rows are used. This leads to less reconfigurations of the array and a higher degree of parallelism inside the array.

5 Evaluation

We evaluated the static speculation algorithm using seven selected benchmarks of the MiBench Benchmark Suite [7]. We first compiled them using a standard compiler (GCC with optimizations turned on, $-O3$). Second, we performed on these binaries a static analysis and applied the proposed post-link-optimization with our tool GAPtimize.

The GAP is simulated by a cycle accurate simulator which can execute the same binary files as the SimpleScalar simulator [3] using the PISA instruction set architecture. For all benchmarks, we used a *bimod* branch predictor and an identical cache configuration.

Figure 3 shows the speedup that can be gained for the seven selected benchmarks and several configurations of the GAP by the optimization technique over unmodified code. They have been distributed on the two charts according to the main reason for the speedup. The maximum speedup of 2.08 is achieved for benchmark `secu-rijndael-encode` (AES) with an array of 12 rows, 12 columns and 32 layers (i. e. 12x12x32). The speedup is calculated as the number of total clock cycles for the unmodified program divided by the number of clock cycles needed for the modified program executed on GAP with identical configuration.

The speedup for the benchmarks `secu-rijndael-decode-nounroll`, `secu-rijndael-encode-nounroll`, and `tele-crc32` is caused by effects beyond those described in Section 4. The main reason for the speedup is here GAP's ability to accelerate the execution of loops if the loop body fits completely into the array. This is more often the case if the program has been modified with reduction of the length of configurations as objective. As example, GAP with the 12x12x1 configuration executing `tele-crc32` accelerates 108310 loop iterations after applying the algorithm instead of 428 without modifications. This is because the configuration of the loop is short enough after applying the static speculation optimization to map it onto the single available configuration layer. In other words, the static speculation and the hardware architecture are working hand in hand.

The more configuration layers are available the less is the impact of the optimization for `tele-crc32` and `cons-jpeg-encode`. This is because larger loops can be mapped to multiple layers anyway and, hence, the advantage of the static speculation is not as high as with a small number of layers. The acceleration of the two `secu-rijndael-*-nounroll` benchmarks is caused by the same effects. Hereby, a very long loop is mapped to multiple layers and by static speculation the number of layers required for the loop is reduced. Consequently, more layers are available to configure other code fragments.

Nevertheless, speed up can also be gained for benchmarks without dominant loops like `cons-jpeg-encode`. This is due to a higher level of ILP (more FUs er line of the array are used) and less instruction cache misses. Again, these effects are reduced if the number of configuration layers is increased.

6 Conclusion

In this paper, we present an algorithm for a post-link-optimizer to increase the degree of ILP in some parts of a program. Therefore, instructions are moved from one basic block to the preceding blocks. This modification allows in-order architectures with high fetch and execute bandwidth to execute these instructions speculatively. The speculative instructions are statically modified to use registers not required by the original program flow at that time. If the following branch is resolved the results are copied into the original target registers, if necessary. Otherwise, they are discarded. Additional hardware for speculative execution is not required. Our evaluations show a maximum speedup factor of 2.08 for a standard benchmark using GAP.

A side effect of the static speculation algorithm is that moving instructions over a loop back branch is similar to software pipelining. In the future we will focus more on this aspect. As example, it would be possible to add an additional step to reschedule the instructions of the source block before modification to increase the number of instructions that can be moved to the target blocks.

Another topic that we will examine is the real-time capability of the proposed approach. Speculative execution is also applied within out-of-order processors but, in contrast to our approach, its timing behavior is nearly unpredictable because of its dynamic nature.

References

1. D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. *SIGPLAN Not.*, 26(6):241–255, 1991.
2. R. A. Bringmann. *Enhancing instruction level parallelism through compiler-controlled speculation*. PhD thesis, University of Illinois, Champaign, IL, USA, 1995.
3. D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, June 1997.
4. S. Cotofana and S. Vassiliadis. On the design complexity of the issue logic of superscalar machines. In *EUROMICRO '98: Proceedings of the 24th Conference on EUROMICRO*, page 10277, Washington, DC, USA, 1998. IEEE Computer Society.
5. J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30:478–490, 1981.
6. D. Gregg. Comparing tail duplication with compensation code in single path global instruction scheduling. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 200–212, London, UK, 2001. Springer-Verlag.
7. M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. pages 3 – 14, dec. 2001.
8. J. Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, page 80, Washington, DC, USA, 1996. IEEE Computer Society.
9. P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.
10. S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. mei W. Hwu, B. Ramakrishna, R. Michael, and S. Schlansker. Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11:376–408, 1993.
11. B. Shehan, R. Jahr, S. Uhrig, and T. Ungerer. Reconfigurable grid alu processor: Optimization and design space exploration. In *Proceedings of the 13th Euromicro Conference on Digital System Design (DSD) 2010, Lille, France*, 2010.
12. J. E. Thornton. Parallel operation in the Control Data 6600. In *AFIPS '64 (Fall, part II): Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, pages 33–40, New York, NY, USA, 1965. ACM.
13. J. E. Thornton. *Design of a Computer—The Control Data 6600*. Scott Foresman & Co, 1970.
14. P. Tirumalai and M. Lee. A heuristic for global code motion. In *ICYCS'93: Proceedings of the third International Conference on Young Computer Scientists*, pages 109–115, Beijing, China, China, 1993. Tsinghua University Press.
15. R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J Res Dev*, 11(1):25–33, 1967.
16. S. Uhrig, B. Shehan, R. Jahr, and T. Ungerer. The two-dimensional superscalar GAP processor architecture. *International Journal on Advances in Systems and Measurements*, 3:71–81, 2010.