Static Multiple Path Prediction on VLIW Multimedia DSPs

Dirk HERRMANN and Rolf ERNST
Technische Universität Braunschweig, Institut für Datenverarbeitungsanlagen
Hans-Sommer-Straße 66, D–38106 Braunschweig, Germany

Abstract. Recent digital signal processors (*DSPs*) like the Texas Instruments TMS320C60 or the Philips TriMedia which are targeted towards multimedia applications provide an optimization friendly VLIW architecture. This allows to use higher level languages like C even for performance critical parts. However, since these processors don't support dynamic branch prediction, control statements (if) can limit the potential instruction level parallelism (*ILP*). Static multiple path prediction (*MPP*) is a technique developed for high-level synthesis that can vastly improve the ILP and reduce loop execution time for code with control statements. We will present MPP as a source level transformation strategy that can be efficiently exploited on VLIW DSPs. MPP can, however, be automated and included in the compiler as it has been demonstrated for high-level synthesis. The experiments show a significant reduction in memory size and/or a decrease in execution time as a result of source code optimization.

1. Introduction

Earlier digital signal processors (*DSPs*) were focused on signal processing algorithms with a fixed control flow such as static loops with a fixed number of iterations which were supported by special loop instructions and stacks. Thus, even for newer signal processing algorithms one of the design objectives is to keep the control flow independent from input data.

Recent VLIW DSPs, such as the Philips TriMedia [13] or the Texas Instruments TMS320C60, also support input data dependent control flow, thus widening the application range. Since VLIW DSPs execute several operations in parallel, control dependencies on input data can degrade performance, in particular in case of pipelined execution. This has been thoroughly investigated in general processor architectures and spurred the development of sophisticated dynamic branch prediction strategies which are standard in modern super scalar architectures [6]. However, the hardware overhead is high, since to achieve good prediction accuracy branch history tables have to be kept and controller and register file complexity increases.

On the contrary, VLIW compilers make use of static techniques for branch prediction and speculative execution like trace scheduling [5] that are typically based on profiling or tracing data. In many approaches additional hardware support (e. g. a specialized instruction set) is assumed to handle exceptions during the execution of predicted operations [3]. To allow more aggressive scheduling or prediction across multiple paths, some static prediction approaches also depend on specialized register files [14, 2].

In this paper we will present multiple path prediction (*MPP*), a compile time technique originally developed for high-level synthesis [8]. MPP is a code transformation strategy that improves the optimization potential when *software pipelining* or *loop pipelining* [1] is applied for loop scheduling. MPP can lead to performance and/or memory size improvements on the above VLIW DSPs without need for additional architectural support. MPP can easily

This work was supported by the Deutsche Forschungsgemeinschaft (DFG)

be combined with other approaches to speculative execution like [15] and[12] and also integrates nicely with architectural support for conditional (*guarded*, *predicated*) execution of operations [10], which both of the above VLIW DSPs provide and which can be efficiently supported by compilers [11, 4].

We will first explain how MPP allows to minimize the average loop execution time, then we will show how to apply the new speculation strategy to the VLIW DSPs as a source level transformation strategy. The results show that a significant reduction in memory size and/or a decrease in execution time can be achieved as a result of source code optimization.

2. Multiple Path Prediction

2.1 Loop Pipelining

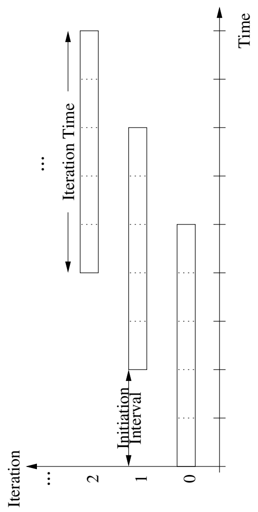


Figure 1: Pipelined Loop Schedule

Since MPP is a transformation technique targeted towards the effective exploitation of loop pipelining (*LP*), we will first give a short introduction to the concepts of LP. LP is a loop scheduling technique that tries to optimize throughput by overlapping the execution of several consecutive iterations of a loop. The performance of the resulting schedule is defined by the *iteration time* t_i, which determines the number of time steps to execute a single iteration of the pipelined loop, and the *initiation interval*, which determines the number of cycles t_n after which the next iteration will be started (see figure 1). The major goal of the scheduler is to minimize the initiation interval since this way throughput is maximized [1].

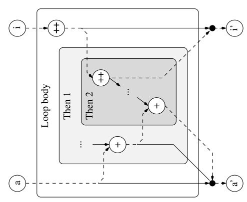
Although finding the minimum initiation intervals for the loops of a given application on a certain target architecture is an NP complete problem, a number of heuristics are applicable to determine lower bounds for the initiation interval:

- Let l determine the length of a loop of data dependencies (a *loop carried data dependency*) in the CDFG which crosses i iteration boundaries. Then $\lceil \frac{l}{i} \rceil \leq t_n$ determines a lower bound for the minimum initiation interval.

- For a functional unit type f of the target architecture let $O_{f,p}$ denote the set of operations which belong to the control path p through the loop body and which are to be executed on f. Let $\sigma_{f,p} = \sum_{o \in O_{f,p}} t(o)$ be the sum of the execution times of the operations in $O_{f,p}$. With u_f giving the number of functional units of type f, another heuristic for the lower bound of the minimum initiation interval is given by $\lceil \frac{\sigma_{f,p}}{u_f} \rceil$.

```
a=0; i=0;
do {
  data_lo = x[i++];
  if (data_lo & 0x8000 == 0) {
    /* THEN 1 */
    a += data_lo & 0x3fff;
    if (data_lo & 0x4000 != 0) {
      /* THEN 2 */
      data_hi = x[i++] << 14;
      a += data_hi;
    };
  };
} while (a < limit_a);
```

a



b

Figure 2: Example Code

Figure 2a shows an example C-loop that works on a field of specially encoded input values. The computations of the variables `i` and `a` form a loop carried data dependency, since the results of the computations for `i` and `a` from iteration n are used in iteration $n + 1$. This is illustrated in figure 2b, which shows those operations of the corresponding CDFG which form the loop carried dependencies. The shaded boxes indicate to which part of the control structure of the input description each operation belongs. Black circles represent the joining of values from the three different paths through the loop body.

Both critical paths through the loop carried dependencies (shown as dotted lines in figure 2b) require a sequence of two operations to be performed. Thus, (if we assume one clock cycle for the execution of each operation) the lower bound for the minimum initiation interval is two cycles. Further, assume that all arithmetic operations except shifts and address calculations in figure 2 (add, increment, logic-and, compare) will be executed on the same type of ALU. In this case the control path which is executed when both `if` conditions evaluate to `true` contains ten ALU operations. If the target architecture only provides four ALU instances, this results in a lower bound of $\lceil 10/4 \rceil = 3$ for the minimum initiation interval.

2.2 Transforming the Loop

The figures 2a and b show, that the code of the second `then` block contributes one increment and one add operation to the computation of the lower bound of the initiation interval. Further, these operations add to the critical path of the computation of `i'` and `a'`. Assume that it is known (a priory or by profiling or tracing) that this path through the loop body will be very rarely taken. Still, the additional operations on that path influence the lower bound for the initiation interval for a pipelined schedule of the loop body.

```
a=0; i=0;
do {
  do {
    data_lo = x[i++];
    if (data_lo & 0x8000 == 0) {
      a += data_lo & 0x3fff;
      if (data_lo & 0x4000 != 0)
        goto label;
    };
  } while (a < limit_a);
  break;

  label:
  data_hi = x[i++] << 14;
  a += data_hi;
} while (a < limit_a);
```

Figure 3: Multiple Path Prediction Example

Figure 3 shows how MPP transforms the code from figure 2a. The resulting inner loop consists of a subset of the paths (the *predicted* paths) of the original loop. This subset of paths allows for a tighter schedule of the inner loop. The operations of the remaining paths are extracted to the outer loop. For the code in figure 3 this means that a initiation interval of two clock cycles can be achieved, in contrast to three cycles before. As long as only predicted paths are executed the performance is increased due to the reduced initiation interval. If it becomes evident during the execution of the inner loop that one of the extracted paths has to be executed, we call this a *misprediction*. In case of a misprediction the pipelined execution of the inner loop is stopped, the corresponding code of the outer loop is executed and the pipeline of the inner loop is restarted.

Let P be the set of paths in the original loop body and let $\pi(p)$ be the probability that $p \in P$ will be executed. Further, let P_+ denote the set of predicted paths (the *prediction key*) and $P_- = P - P_+$ the set of remaining paths that will be extracted to the outer loop. The prediction key determines the iteration time t_{i,P_+} and initiation interval t_{n,P_+} of the pipelined schedule of the inner loop, which holds for all paths in P_+. In contrast, every path $p \in P_-$ determines an iteration time $t_{i,p}$ of its own: Different paths from P_- require different compensation code to be executed in the outer loop and there are different exit points from the inner loop for the different paths in P_-. With these parameters the average iteration time for the transformed loop can be approximated with

$$\tau = t_{n,P_+} \sum_{p \in P_+} \pi(p) + \sum_{p \in P_-} \pi(p) t_{i,p}$$

The objective of MPP is to determine a prediction key $P+$ such that the average iteration time is minimized. Due to the complexity of the scheduling problem, heuristics (like the two lower bound heuristics described in section 2.1) are applied in order to determine the parameters t_{n,P_+} and $t_{i,p}, p \in P_-$. We won't give any details here, since a description of MPP and its effectiveness in the context of high-level synthesis can be found in [8]. In the remaining parts of this paper we will describe our experiences when applying MPP on VLIW multimedia processors.

Table 1: VLIW Multimedia DSPs

	TriMedia	TMS320C60
instruction slots	5	8
branch delay slots	3	5
ALUs	5	6
multipliers	2	2
RAM ports	2	2
registers	126	32

3. MPP on VLIW Processors

Similar to high-level synthesis, VLIW architectures provide explicit parallel control over a set of functional units. Since the compiler is responsible for the exploitation of instruction level parallelism, both areas benefit equally from advanced scheduling techniques like loop pipelining or compiler controlled speculative execution. Consequently, the application of MPP can be expected to improve performance on VLIW architectures as well. However, the following considerations lead to the assumption, that the performance improvements will be smaller than with high-level synthesis:

- With MPP, in case of a misprediction a branch has to be executed to leave the inner loop. In high-level synthesis, branches are 'free' since they are encoded as state transitions in the finite state machine. In contrast, a branch on a VLIW machine has to be encoded like an ordinary operation and thus occupies an instruction slot.
- To be able to recover from a misprediction, a set of temporary values has to be kept for use in the outer loop. Since on VLIW machines the number of registers is limited, register pressure is increased.
- Recovering from a misprediction often requires additional data transfers between registers. Again, this operation is 'free' in high-level synthesis where a specialized register file can be generated [7] but it is costly in terms of occupied instruction slots on the VLIW architecture.

In our experiments we applied MPP on two multimedia VLIW DSPs, the Philips TriMedia and the TI TMS320C60 VLIW DSPs. Table 1 gives some core parameters of the processors. The number of instruction slots determines the maximum number of instructions that can be issued in one clock cycle. Due to the deep processor pipelines, both DSPs have a high number of branch delay slots. The number of functional units determines the number of operations of a certain class that can be executed concurrently. For example, both processors allow to perform two memory accesses in parallel. The last row shows the number of general purpose registers that are available. In addition, both processors provide full support for conditional execution, i. e. every single instruction may be guarded by the result of a condition.

The experiments were performed by transforming a set of algorithms on the source code level as it has been illustrated with the example code in figure 2 and 3. Each of the compilers follows a different loop optimization strategy: The C60 compiler uses loop pipelining, whereas the TriMedia compiler generates *decision trees* [9] for scheduling, a technique which is basically a generalization of loop unrolling. Although MPP was developed for loop pipelining, it was possible to achieve improvements with the TriMedia compiler as well. Still, on both systems the application of MPP on the source code level did not allow for fine grain optimization in interaction with the instruction scheduler. However, it allowed to measure the performance gain of MPP in combination with other optimizations of the native compilers like exploitation of guarded instruction execution.

Table 2: Execution Time and Executable Size

TMS320C60	cycles per iteration			instruction words		
	w/o MPP	with MPP	gain	w/o MPP	with MPP	gain
circle	4.0	3.5	12.5%	51	45	11.8%
huffman	20.0	16.7	16.5%	88	65	26.1%
median3	28.3	—	—	46	29	37.0%
plot	13.4	5.30	60.4%	27	—	—
wc	29.0	8.47	71.1%	40	—	—

TriMedia	cycles per iteration			instruction words		
	w/o MPP	with MPP	gain	w/o MPP	with MPP	gain
search	10.3	8.3	18.9%	145	46	68.3%
huffman	23.8	20.0	15.9%	41	37	9.8%
median3	11.2	10.4	6.9%	84	95	—

4. Results

Table 2 shows the improvements in execution time and code sizes that could be achieved for a set of algorithms. Each of the algorithms has a core loop with several possible execution paths: A binary search algorithm as used in a histogram based median filtering algorithm (2 paths), Bresenham's circle drawing algorithm (2 paths), a Huffman decoder (8 paths), a median filter of depth 3 (7 paths), the code piece from figure 2 which is part of a plotting routine (3 paths), and the core of the UNIX word count program (6 paths). For each of the DSPs we applied MPP with different prediction keys on the core loop of each algorithm. Since in each case the full set of available compiler optimizations was chosen the results show the effective improvements due to the application of MPP. Table 2 shows the shortest possible execution times and smallest code sizes that could be achieved on each processor. Benchmarks which could not be improved on a processor are left out from the corresponding table. Also, dashed entries indicate that no improvement was achieved. The execution time numbers for the TriMedia were obtained by running each algorithm with a set of random input data on a cycle true simulator. The results for the C60 were obtained by inspection of the assembly code that was generated by the compiler.

According to table 2 for most of the algorithms a performance improvement or a code size reduction could be achieved with MPP on the C60. However, only four of the benchmarks could be sped up because the compiler was not able to generate a pipelined loop if there were too many paths in the loop body. Reducing the set of predicted paths too much meant, on the other hand, that the frequency of mispredictions became too high. The remarkable gain with the plot and word count algorithms was achieved since without MPP no loop pipeline was found, but after MPP was applied the reduced number of paths allowed the compiler to generate a pipelined loop with an initiation interval of four respectively three cycles.

The effects of MPP on the TriMedia showed to be much harder to foresee than on the C60. This was to be expected since MPP is more closely related to loop pipelining than to loop unrolling. Still, reducing the number of paths in the loop body allowed the compiler to apply guarded execution more efficiently and thus reduce the actual number of branches that had to be taken. Since the TriMedia has a large instruction cache of 32 kByte, the compiler generates large decision tree structures. This showed to be a very efficient technique to exploit the ILP, especially for the word count algorithm: The average number of instructions per cycle was 4.79, which is close to the theoretical limit of 5 and could not be improved.

5. Industrial Significance

Algorithms for real-time video and image processing are performance critical parts of multimedia systems. Today it is common practice to implement core routines in hand written assembly code in order to avoid compilation overhead, in particular since today's compilers can not efficiently exploit irregular architectural features like multimedia extensions for standard processors. Due to the low level of abstraction, coding in assembly has enormous drawbacks with respect to productivity as well as to the maintainability and portability of the resulting code.

In contrast, processors like the TMS320C60 or the TriMedia which are targeted towards multimedia applications provide an optimization friendly regular VLIW architecture. Thus, coding of performance critical routines in higher level languages like C becomes feasible, since these architectures allow a good exploitation of instruction level parallelism by the compiler. Moreover, for code pieces that meet certain criteria, it becomes almost impossible for a human designer to produce assembly code that outperforms compiler generated code on these processors. However, control statements (if) in core loops of signal processing applications can degrade the performance. Thus, the efficient compilation of such control structures is still subject to research.

Static multiple path prediction, as was presented, adds another optimization technique for VLIW processors that can be used to improve the performance and/or the code size of loops with control statements. Although we presented how this technique could be performed manually on the source code level, we do not suggest to apply it that way since the maintainability of the code gets much worse. Further, the code would become less portable since the selection of the prediction key is dependent on the target architecture. In contrast, we believe that the simplicity of the MPP transformation makes it easy to integrate the technique in future VLIW compilers.

6. Conclusion

We have presented how multiple path prediction, a loop transformation technique, can be used to improve the performance and/or the code size of loops with control statements. It was demonstrated that MPP, although originally developed for high-level synthesis, can effectively be applied in the domain of VLIW processors. Our experiments were performed manually on the source code level and showed performance improvements of typically more than 10%, going up to 71% in extreme cases. These experiments were performed on two different VLIW DSPs, both of which are specifically targeted towards multimedia applications. The results also showed that the combination of MPP with other advanced compiler optimizations can easily be achieved. Moreover, in some cases certain optimizations became only possible after MPP was applied.

Application of MPP on the source code level, however, is not advisable due to the negative effects on code maintainability. In contrast, our results suggest to implement MPP as an automated technique in VLIW compilers as it has already been done for high-level synthesis. This brings the additional benefit that even better optimization results are to be expected when the technique is applied in interaction with the instruction scheduler of the compiler.

References

[1] I. E. Bennour and E. M. Albouhamid. Lower bounds on the iteration time and the initiation interval of functional pipelining and loop folding. *Design Automation for Embedded Systems*, 1:333–355, 1996.

[2] M. Chang and F. Lai. Efficient exploitation of instruction-level parallelism for superscalar processors by the conjugate register file scheme. *IEEE Trans. on Computers*, 45(3):278–293, March 1996.

[3] P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu. Three architectural models for compiler-controlled speculative execution. *IEEE Trans. on Computers*, 44(4):481–494, April 1995.

[4] A. E. Eichenberger and E. S. Davidson. Predicated register allocation. In *Micro*, 1995.

[5] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. on Computers*, 30(7):478–490, July 1981.

[6] J. Hennessy and D. Patterson. *Computer architecture – A quantitative approach*. Morgan-Kaufmann, second edition, 1996.

[7] D. Herrmann and R. Ernst. Register synthesis for speculative computation. In *ED&TC*, pages 463–467, 1997.

[8] U. Holtmann and R. Ernst. Combining MBP-speculative computation and loop pipelining in high-level synthesis. In *ED&TC*, pages 550–555, 1995.

[9] S. H. Huang, Y. L. Jeang, C. T. Hwang, Y. C. Hsu, and J. F. Wang. A tree-based scheduling algorithm for control-dominated circuits. In *DAC*, pages 578–582, 1993.

[10] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *ISCA*, pages 138–150, 1995.

[11] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ILP processors. In *ISCA*, pages 120–129, 1994.

[12] I. Radivojević and F. Brewer. Incorporating speculative execution in exact control-dependent scheduling. In *DAC*, pages 479–484, 1994.

[13] S. Rathnam and G. Slavenburg. An architectural overview of the programmable multimedia processor, TM-1. In *COMPCON*, 1996.

[14] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *ISCA*, pages 344–354, 1990.

[15] K. Wakabayaschi and H. Tanaka. Global scheduling independent of control dependencies based on condition vectors. In *DAC*, pages 112–115, 1992.