Studies on Different Modeling Aspects for Tight Calculations of Worst Case Execution Time

A. Hergenhan and A. Siebenborn and W. Rosenstiel

Forschungszentrum Informatik (FZI)
Haid-und-Neu-Str. 10-14
76131 Karlsruhe, Germany

Universität Tübingen
Sand 13
72076 Tübingen, Germany

{hergen,siebenbo}@fzi.de, rosenstiel@informatik.uni-tuebingen.de

Abstract

One architectural feature of modern processors, is speculative branch processing. *In this paper we present a method, to consider branch processing with static timing analysis. The determination of path information is crucial for the correctness of WCET estimation. To maintain this information during development and provide it to the analysis tool is important to know the correspondence between source code and assembly code structures. In the second part of this paper we propose a method to perform a mapping between source code and assembly code, using structural information of the program as well as compiler generated debug information.*

1 Introduction

In the area of real time systems static timing analysis of software is a subject of quite some time. Compared to simulation or profiling, static timing analysis collects information about the program as well as about the architectural properties of the processors at compile time, rather than at runtime. The timing analysis should provide timing estimations about *worst case* or *best case* scenarios. Our experimental analysis tool GROMIT [3] is based on the *integer linear programming (ILP* approach of LI AND MALIK [5]. It is a general approach by means of applying general techniques for solving optimization problems to static timing analysis. The *ILP* methodology works well to a certain extend with respect to reported estimation errors. Although estimation errors are plausible in general, you might wondering about their sources. It is widely accepted, that they are generally introduced by

- path modeling where inaccuracies are introduced by infeasible paths considered to be feasible or conservative assumptions of path counts
- architectural modeling where inaccuracies come from nor or insufficient modeling of architectural properties

Furthermore, since both aspects are coupled by the *ILP* methodology, their effects are mingled and so the individual inaccuracies do.

Thus, this paper is organized as follows. In section 2, we describe a modeling technique in order to handle branch processing architectures. Section 3 will touch the problem of path annotation in order to provide information about necessary loop bounds as well as about desirable path information.

2 The modeling of branch prediction

Analysis block modeling is about finding an appropriate schedule of the analysis according to the properties of the instruction pipeline of the processor architecture. The latency of the schedule of the analysis blocks are then taken as costs for the analysis. Besides others, the methodology of breaking the entire execution path into peaces introduces additional errors. For the sake of being worst case, every analysis modeling starts with the assumption of an empty instruction pipeline. This modeling artifact means, that edges between analysis blocks embody pipeline stalls. This holds true for some minor counts of analysis block transitions, but not in general.

2.1 Critical points at traditional modeling

For now, the analysis structure does not consider interblock relations given along analysis block transitions, at all. The possible modeling artifacts causing modeling inaccuracies are summarized in table 1.

Case *A* of table 1 shows an entire artificial transition by the means that there is no pipeline stall at all. The transition is generated by the labeled target of a branch transition into analysis block AB_j. Note, that this type of transition is only possible if the involved analysis blocks are basic blocks

Table 1: Modeling artifacts due to the analysis structure

case	transition
A) AB_i → AB_j	• transition due to labeled analysis block AB_j • artificial transition causes modeling error
B) AB_i → AB_j	• transition due to a unconditional branch, a function call or return from a call • inaccurate modeling due to branch modeling within the analysis block AB_i
C) AB_i → AB_j, AB_k	• transition due to a conditional branch • inaccurate modeling of the *taken* and *not taken* paths of the branch due to branch modeling at the end of analysis block AB_i

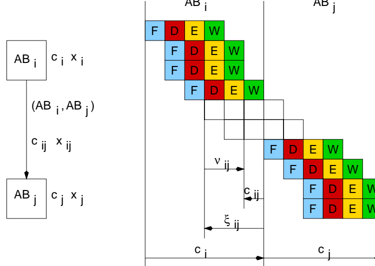

Figure 1: Magnitude and meaning of the values of the analysis edge (AB_i, AB_j)

or if the analysis blocks are connected by the affiliated basic block transition. Thus, the modeling error of this artificial transition is specified by the possible folding of both instruction pipelines.

Case B of table 1 shows a transition between the analysis blocks AB_i and AB_j which is generated by an unconditional branch at the exit of analysis block AB_i, if the analysis blocks are connected by the affiliated basic block transition or the sequence of analysis blocks (line blocks) within a basic block. In first case, the branch instruction will redirect the execution of the instruction flow. This may result in a pipeline stall. But, since the target point of the redirection is known, the number of stall cycles are much less than the modeled by an empty pipeline of analysis block AB_j. This leads to a constant degree of folding for the adjacent analysis block pipelines. This is also the case for function calls or returns from a function call. The second case is equivalent to case A of table 1.

Case C of table 1 shows a transitions between the analysis blocks AB_i and AB_j and AB_i and AB_k due to a conditional branch at the exit of analysis block AB_i. In addition to the remarks of case B, this case lacks a diversification of the *taken* and *not–taken* exit of the branch.

The effective magnitude of the described modeling artifacts due to branches depends on the quality of the used branch processing of the processor architecture. In general, the quality of the branch processing is defined by performance measures of branches and the quality of the prediction of the branch target in case of a conditional branch, known as branch prediction. To summarize, there is a need for modeling the control behavior, if the processor architecture to be analyzed contains sophisticated branch processing. This will also lead to an adapted analysis structure, since the control behavior of the processor is rather a property of the analysis block transition then the analysis block itself. The adapted analysis structure introduces a new analysis class, here referred to as analysis edge, which represents the transition between two adjacent analysis blocks. Figure 1 shows an example. Here, the transition (AB_i, AB_j) is characterized by cost c_{ij} and a counter x_{ij}.

2.2 Determination of the costs

The cost c_{ij} depends on the maximum possible folding of the two pipeline constructions ξ_{ij} and the branch penalty ν_{ij}. The value for ξ_{ij} depends on the construction of the analysis blocks AB_i and AB_j.Thereby, different possible pipeline construction have to be taken into account (instruction cache hits/misses). It can be shown, that there is a common value for all edges ξ_{model} which fits for all pipeline constructions without a great loss of accuracy. With that, the cost c_{ij} can be written as:

$$c_{ij} = \nu_{ij} - |\xi_{model}| \qquad (1)$$

Note, that the costs of the analysis edges are less then zero or zero! The performance measure of the branch processing is the branch penalty. In the general case of using a branch prediction, the possible branch penalties are distinguished by the following notation: ν_{tc} (the penalty for a correctly predicted taken branch), ν_{tm} (the penalty for a mispredicted taken branch), ν_{ntc} (the penalty for a correctly predicted not-taken branch), ν_{ntm} (the penalty for a mispredicted not-taken branch).

2.3 Modeling constraints

With those preliminary remarks on modeling transitions between analysis blocks, the modeling constraints for the three possible cases of table 1 can be specified. In the following we want to exemplify this at case C, the transitions

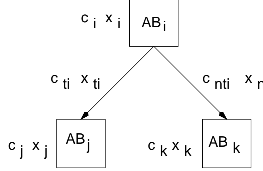

Figure 2: Model for a conditional branch

at a conditional branch. Figure 2 shows the detailed model for a conditional branch.

The detailed cost for both transitions at a conditional branch can now be specified by:

$$c_{ti} = \nu_{ti} - |\xi_{model}| \tag{2}$$

$$c_{nti} = \nu_{nti} - |\xi_{model}| \tag{3}$$

The counters x_{ti} resp. x_{nti} are connected to the counters of the affiliated analysis blocks by:

$$x_{ti} = x_i - x_{nti} = x_j \tag{4}$$

$$x_{nti} = x_i - x_{ti} = x_k \tag{5}$$

For the general case of speculative branch processing, costs and counters can be given for correct x_{tci}, x_{ntci} and incorrect x_{tmi}, x_{ntmi} branch predictions:

$$x_{ti} = x_{tci} + x_{tmi} \tag{6}$$

$$x_{nti} = x_{ntci} + x_{ntmi} \tag{7}$$

According to the differentiated counters, there are specified costs $c_{tci}, c_{tmi}, c_{ntci}, c_{ntmi}$ as well. They can be derived from equations 2 and 3 and the vendor given values for $\nu_{tc}, \nu_{tm}, \nu_{ntc}$ and ν_{ntm}.

The equations for case A and case B can be specified in the same manner as shown for conditional branches.

Finally, the objective function for the *ILP* problem must be extended according to the analysis edge model (figure 1):

$$t_{wcet} = max(\sum \ldots + \sum_{\forall (AB_i, AB_j)} c_{ij} x_{ij}) \tag{8}$$

2.4 Modeling branch processing architectures

Since the different costs for different prediction states are given,we want to focus on the counts of $x_{tci}, x_{tmi}, x_{ntci}, x_{ntmi}$ for those different prediction events in this section.Those counters are strongly dependent on the used branch processing architecture. There are different types of branch processing architectures which have to take into consideration in case of a conditional branch. In case of an unconditional branch, the target is known at compile time and no dynamic branch prediction is necessary.

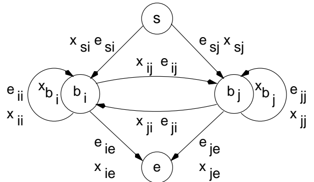

Figure 3: An exemplary BHT conflict graph

The most trivial architecture is the *blocking* branch processing architecture. Here, the amount of the effective pipeline folding is zero and so all costs of the analysis edges coupled with conditional branches. This means further, the traditional analysis structure fits best.

In case of speculative branch processing architectures we want to distinguish between *conservative* and *progressive* modeling alternatives.

The conservative modeling alternative is characterized by a conservative prediction, say the prediction is incorrect for all branches. It is easy to itemize the values for the possible counter variables $x_{tci}, x_{tmi}, x_{ntci}, x_{ntmi}$ for some special variants of *fix*, *static* or *dynamic* branch prediction schemes.

E.g., the values of the conservative modeling for any dynamic prediction scheme are:

$$x_{ntci} = 0 \qquad \text{resp.} \qquad x_{tci} = 0 \tag{9}$$

$$x_{nti} = x_{ntmi} \qquad \text{resp.} \qquad x_{ti} = x_{tmi} \tag{10}$$

A more challenging task is the progressive modeling alternative of dynamic prediction schemes. Since there are a few, we want to illustrate the methodology at the real world branch prediction architecture of the processor *PowerPC* MPC750 of Motorola. This processor has an implicit prediction given by the branch target instruction cache as well as an explicit prediction architecture by a branch history table and a 2 bit branch prediction.

The branch history table (BHT) contains the state (bits) for the branch prediction of the particular conditional branches. It behaves fairly like a direct mapped cache. Consequently, a conflict graph, here called BHT conflict graph, can approximately model this behavior.

Figure 3 shows an exemplary BHT conflict graph. The nodes of the directed graph represent the conflicting instructions for an entry in the branch history table independently of the type of instruction (every instruction gets an entry into the table).There are two additional nodes for the start s and the end of the control flow e. The edges between the modes are the possible control flow. If there are transitions between the nodes, there are removals of the history state bits of the prediction. The transitions that start and end in the same node represent the hits in the branch history tables, here given by x_{ii} or x_{jj}.

The execution counts of the particular instructions connect the BHT conflict graph to the analysis graph. For the

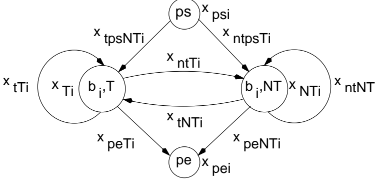

Figure 4: An exemplary branch state transition graph

instruction b_i of figure 3 signifies that:

$$x_{b_i} = x_{ii} + x_{ji} + x_{si} = x_{ii} + x_{ij} + x_{ie} \tag{11}$$

All other flow informations can be drawn in the same manner.

Finally, modeling of the state machine for branch prediction is left open. First of all and for the sake of simplicity, it shall be assumed, that the actual 2 bit prediction scheme can be reduced to the 1 bit prediction. In fact, it can be shown that both prediction schemes have the same worst case scenario. Now, the state machine for the branch prediction using the 1 bit is actually modeled by a directed graph shown in figure 4. In the following, we want to refer to this graph as the *branch state transition graph*.

Say, the node (instruction) b_i of figure 3 is a conditional branch. Then,figure 4 shows the affiliated branch state transition graph. Both graphs are connected by several relations.First of all, the execution count of the node (instruction) b_i is fix, which is equal to the sum of all predictions, predicted em taken (x_{Ti}) or predicted *note taken* (x_{NTi}):

$$x_{b_i}(BHT) = x_{Ti} + x_{NTi} \tag{12}$$

Then, it is pretty easy to set up the necessary flow equations for both possible nodes:

$$\begin{aligned} x_{Ti} &= x_{tpsNTi} + x_{tTi} + x_{tNTi} && (13) \\ &= x_{tTi} + x_{ntTi} + x_{peTi} && (14) \\ x_{NTi} &= x_{ntpsTi} + x_{ntNTi} + x_{ntTi} && (15) \\ &= x_{ntNTi} + x_{tNTi} + x_{peNTi} && (16) \end{aligned}$$

The node *psi* is the target for all those edges of the BHT conflict graph which comes from other nodes than from b_i. The node *pei* is the source for all edges of the BHT conflict graph which leave the node b_i, but do not target to b_i. Thereby, there is a uncertainty, whether these decisions about the branch direction are correct or not:

$$\begin{aligned} x_{peTi} &= x_{tpeTi} + x_{ntpeTi} && (17) \\ x_{peNTi} &= x_{ntpeNTi} + x_{tpeNTi} && (18) \end{aligned}$$

Finally, the interpretation of the particular variables of the branch state transition graph is the most important step. The following equations establish a connection between

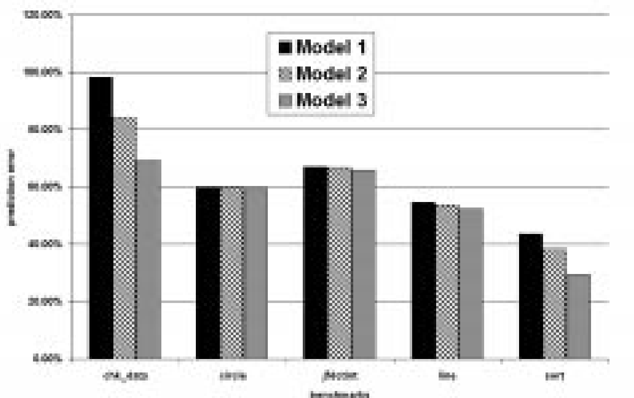

Figure 5: Experimental results for differents modeling techniques on branch processing

the branch state transition graph and the counter variables $x_{tci}, x_{tmi}, x_{ntci}, x_{ntmi}$:

$$\begin{aligned} x_{tci} &= x_{tTi} + x_{tpeTi} && (19) \\ x_{ntci} &= x_{ntNTi} + x_{ntpeNTi} && (20) \\ x_{tmi} &= x_{tsNTi} + x_{tNTi} + x_{tpeNTi} && (21) \\ x_{ntmi} &= x_{ntsTi} + x_{ntTi} + x_{ntpeTi} && (22) \end{aligned}$$

There are only safe assumptions about correct predictions (x_{tci} resp. x_{ntci}), in case of those transitions which start and end in the same node of the branch state transition graph. In addition, there is a possibility to leave the graph while a correct prediction occured (x_{tpeTi} resp. $x_{ntpeNTi}$).

2.5 Experiments

Figure 5 shows some experimental results of some common benchmarks. The benchmarks are measured on a *PowerPC* MPC70 plattform which provides a 2 bit explicit dynamic branch processing technique supported by a branch history table of 512 entries. The shown diagrams of figure 5 are then the relative (to the measurements) prediction errors of the analysis results. The different models are refering to the architectural modeling without modeling (*model 1*), with a conservative (*model 2*) and with a progressive modeling (*model 3*) of the branch processing of that processor. To summarize, figure 5 shows a significant prediction gain for the benchmarks *sort* and *chk_data* and a decent prediction gain for all others.

3 Determination of Path Information

In general it is not possible to determine the WCET of a program without any knowledge about the input data. However it can be difficult to determine the input data, that causes the *worst case*. Instead of information on in-

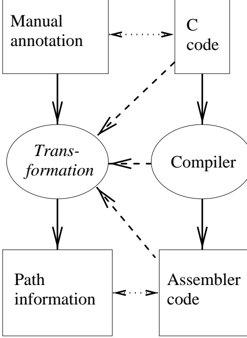

Figure 6: Transformation of manual annotation

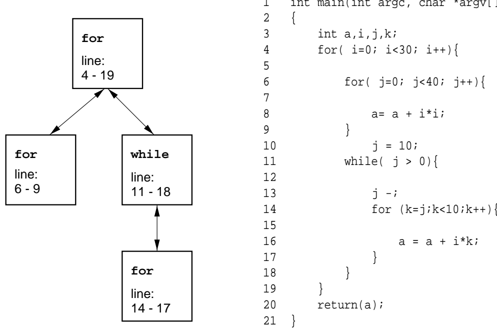

Figure 7: The control tree and the corresponding C code

put data, static timing analysis needs information that restrict the number of possible program paths. The maximum iteration count of loops is necessary to guaranty program termination. Information on infeasible paths describes dependencies between program parts. Though its annotation is not absolutely necessary, it leads to tighter estimation. This information can be provided by manual annotation of the user or by automatic flow analysis.

Both, automatic flow analysis and manual annotations are easier performed at source code level. In many cases it is a simple task for a programmer to bound a loop in his own C code, but much more difficult to find the corresponding loop in the assembly code. However actual calculation of the execution time has to be performed at machine code level. Only at this level hardware effects like caches and pipelines can be considered. To combine the two steps, the path information determined by flow analysis has to be transformed from source level to machine level. For this purpose the correspondence between constructs at source level and machine level has to be found. This is particularly difficult in case of optimising compilers. The control flow graph at source level and machine level may differ significantly.

A simple approach is to use the debug information. In case of optimizing compilers this is not sufficiant. Our idea is to use structural information of the program in addition to the debug information. For this purpose a structural analysis of the code is performed, both on C-level and machine level. This analysis classifies all structure elements, like loops or branches, and the *control tree*, which shows the hierarchical representation of control structures, is constructed. Figure 7 shows an example for a control tree and the corresponding looptree. The following mapping looks for correspondence between structure elements at both level, using the debug information. This way it will be ensured, that a loop will be mapped to a loop and not to a fragment of straight line code. Figure 6 shows the framework of the transformation.

4 Conclusion

For accurate WCET analysis it is important to determine architectural properties as well as path information, especially loop bounds. We presented methods for modeling branch processing architectures and for transforming path information from source level to assembly code level.

References

[1] J. Engblom, A. Ermedahl, and P. Altenbernd. Facilating worst-case execution times analysis for optimized code. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, June 1998.

[2] Christopher Healy and David Whalley. Tighter timing prediction by automatic detection and exploitation of value-dependent constraints. In *Proceedings of the IEEE Real-Time Technologie and Applications Symposium*, pages 288 – 297, 1999.

[3] A. Hergenhan and W. Rosenstiel. Static Timing Analysis of Embedded Software on Advanced Processor Architectures. In *Proceedings of the Date Conference 2000*, pages 552–559, March 2000.

[4] Sung-Soo Lim, Jihong Kim, and Sang Lyul Min. A worst case timing analysis technique for optimized programs. In *Proceedings of the fifth International Conference on Real-Time Computing Systems and Applications (RTCSA); Hiroshima, Japan*, pages 151–157, October 1998.

[5] S. Malik, M. Martonosi, and Y.-T. S. Li. Static Timing Analysis of Embedded Software. In *Proceedings of the 34th Design Automation Conference*. ACM, June 1997.

[6] Steven S. Muchnik. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[7] Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. In *Proceedings of the 1990 IEEE Real-Time-Systems Symposium*. IEEE Computer Society Press, November 1990.