Predication and Eager Execution

Lecture #6: Monday, 17 April, 2000
Lecturer: Dean Liu
Scribe: David Lie

1 Overview

Three papers will be discussed in this section: *Effective Compiler Support for Predicated Execution Using the Hyperblock*, by Mahlke et al.; *Disjoint Eager Execution: An Optimal Form of Speculative Execution*, by Uht and Singdagi; and *Selective Eager Execution on the Poly Path Architecture*, by Klauser, Paithankar and Grunwald. These papers will close out the "processor front-end" section of the course. We will examine each of the papers in order.

2 Effective Compiler Support for Predicated Execution Using the Hyperblock

A hyperblock is a set of predicated basic blocks in which control may only enter from the top, but may exit from one or more locations [1]. We note that this implies that hyperblocks only have one entry point, and have no loops within them. This is significant since, if this were not the case, instructions within hyperblocks could not be arbitrarily reordered. The hyperblock is effectively a superblock with predicated instructions.

The architecture on which the studies were performed is illustrated on page 3 of [1]. The central architectural element of interest is the predicate register file. The two bits in each predicate denote the true and false values of the predicate. These values will then be used to determine if instructions, predicated on these values will be execute or not. We note that the true and false values of a given predicate, are not necessarily complements of each other. This is done to support predication down nested branches. To understand this, consider Figure 1. We can see that the true branch (in bold) of the first *if* statement is taken. Thus regardless of what the second branch evaluates to, its true value and false value are not to be executed and, thus have value zero. Thus this method allows entire subgraphs of predicates to be killed simply by and-ing the values with the parent.

Another characteristic of the architecture was that it squashed predicated instructions that speculated incorrectly at the writeback stage. This was done so that a smaller distance between the predicate define and subsequent use to predicate an instruction on

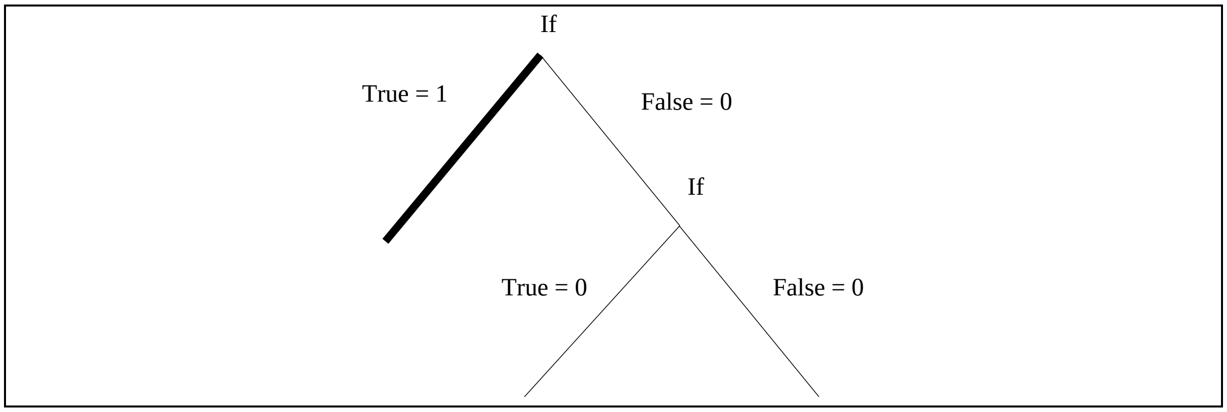

Figure 1: Predicate registers

it could be supported. The drawback is that instructions which speculated incorrectly still continue to use up resources in execution if only squashed at writeback. On the other hand, this method is much simpler the predicate registers are only checked in one execution stage.

The rest of the ensuing discussion concerned the various compiler techniques used to extract hyperblocks out of code. These include: *Tail Duplication*, which duplicates entry points moving them out of the hyperblock; *Loop Peeling* which splits off and unrolls several iterations of a loop into the hyperblock; *Node Splitting* which duplicates reconvergent paths so that one duplicate is in the hyperblock and the other is not; and *If-conversion* which converts branches to predicated instructions, effectively converting control dependencies to data dependencies. Diagrams on pages 4, 5 and 6 in [1] illustrate some of these techniques. The important thing to note is that all techniques increase the instruction footprint. However, Mahlke assumes a 100% cache hit rate and thus does not model the negative cache effects that larger instruction footprint would have.

There are two compiler optimizations which result from the use of predicated instructions: *Instruction Promotion* and *Instruction Merging.* The former involves situations where idempotent instructions may be moved beyond their control dependence if that dependence has now been converted to a data dependence. Instruction Merging is a consequence of promotion where two promoted instructions now may perform the same operation at the same place, and may thus be removed.

Finally, some discussion about performance and the Block Selection Value (BSV) heuristic ensued. A critical point is the decision as to which basic blocks should comprise a hyperblock. Experiments where over-zealous inclusion of basic blocks along all paths showed lower performance. The BSV heuristic is used to decide which basic blocks to include in a hyperblock. Essentially, basic blocks which are long, risky or infrequently executed should not be candidates for hyperblock inclusion. The definition of risky is not well defined, but the implication is that risky code includes code that may cause faults or otherwise unpredictable delays. In addition, the author alludes to difficulties in combining speculative execution with predication, as motivation for hyperblocks. The audience could not really understand what was meant by this since predication is simply a way of implementing compiler controlled speculation. Some comment should be made about the rather unfair comparison to superblock techniques as well, since the comparison was made to with a large number of superblock optimizations turned off. It is interesting to note however, that some the same authors also co-authored another paper on superblocks [2].

3 Disjoint Eager Execution: An Optimal Form of Speculative Execution

It was universally agreed that this paper was difficult to understand. An idea called Disjoint Eager Execution (DEE) is alluded to but not explained until well into the paper. The author claims speedups on the order of 26, which seems unrealistic. In addition, the simulation model used requires a processor with about 200 execution units. The justification for a static instruction window involves the assumption that we will have to predict and speculatively execute all paths down a line of 20 branches. A lot of these assumptions are a little beyond reason.

The discussion on this paper was rather short, but the essential contribution is that path should be allocated resources in proportion to its probability of execution. The static window mentioned above acts as a loop buffer. Fixing the static window size at 32 instructions causes loops with greater than 32 instructions to perform horribly. Sudden discontinuities in performance characteristics such as this are usually undesirable. Finally, it must be understood that the techniques in this paper rely heavily on the accuracy of branch predictions, but what is more, required the ability to ascertain quantitatively the accuracy of those predictions. Only with this information may the processing elements be allocated correctly.

The architecture uses hardware to track which instructions are Really Executed (RE) and which are Virtually executed (VE). VE instructions are ignored and not executed – the VE bits act much as predicate bits do. Renaming is performed through a Instruction Shadow Sink (SSI) register file. Each instruction executes as soon as its operands are ready allowing full out of order execution.

4 Selective Eager Execution on the Poly Path Architecture

The technique of Selective Eager Execution (SEE)is closely related to that of DEE in that resources are allocated according to branch probability. However, a more complete treatment of it is given here comparing it to dual path execution and monopath execution. Essentially, both paths of branches with low confidence (or low accuracy) are executed.

The architecture squashes mispredicted instructions by passing a Context Tag (CTX) along with each instruction. When a prediction is resolved, the mispredicted instructions are eliminated on the fly by comparing the tags. Registers are mapped or renamed to the appropriate instruction through these tags. What these tags allow are multiple concurrently executing contexts.

Like in Disjoint Eager Execution, the branch predictor must be extended to include confidence information. Again, the performance of SEE is heavily dependent on the accuracy of the confidence information.

A fair comparison of SEE is made with dualpath and monopath execution with oracle and gshare branch predictors as well as Jacobsen et al. (JSR) confidence estimators. In addition, studies on the performance versus branch predictor sizes, instruction window sizes, number of functional units and pipeline depth were performed. One of the odd things was that instructions window sizes between 64 and 1024 were simulated. These are rather large instruction window sizes by today's standards. A study on the effect of tag size (effective limit on number of concurrent contexts) would also have been useful in this case. In addition, the performance of this mechanism improves if the branch predictor behaves poorly. This is a bit unfair since simply allocating more resources to improve the predictor can reduce the effectiveness of this scheme.

For this architecture, high fetch bandwidth is a requirement since many extra instructions will be speculatively executed. In addition, misprediction penalties will be exacerbated since squashed instructions will consume functional unit resources as well as fetch bandwidth.

References

[1] S. Mahlke et al., *Effective Compiler Support for Predicated Execution Using the Hyperblock.*

[2] Hank, R.E.; Mahlke, S.A.; Bringmann, R.A.; Gyllenhaal, J.C.; Hwu, W.W. *Superblock formation using static program analysis.* Proceedings of the 26th Annual International Symposium on , 1993 , Page(s): 247 -255