EVALUATION OF INSTRUCTION SETS FOR SUPERSCALAR EXECUTION

Virginia Escuder Cabañas,
Raúl Durán Díaz,
Rafael Rico López

Dpt. de Automática, Escuela Politécnica, Universidad de Alcalá
28871 Alcalá de Henares, Madrid

ABSTRACT

Instruction set design is a fundamental aspect of computer architecture. A critical requirement of instruction sets design is to allow for concurrent execution, avoiding those constructs that may produce data dependencies. Therefore, it is important to count on methods and tools for the evaluation of the behavior of instruction sets and quantify the influence of particular features of its architecture into the overall available parallelism. We propose an analysis method that applies graph theory to gather metrics and evaluate the impact of different characteristics of instruction sets as sources of coupling, quantifying available parallelism. We present a case study using the x86 instruction set and obtain some measures of the influence of condition flags in code coupling.

KEYWORDS

Instruction Set Architecture, Instruction level Parallelism, Graph Theory.

1. INTRODUCTION

The design of instruction sets is a fundamental part of Computer Architecture. One of the most critical aspects of the design of instructions has to do with concurrency: those requirements that must be fulfilled to allow for superscalar execution. A crucial aspect for the parallel execution of programs is to avoid code coupling caused by data dependencies amongst instructions whenever this is not imposed by the computational task but by the architecture of the language used. It is therefore important to count on tools for the evaluation of existing instruction sets behavior and quantify the influence of particular features of its architecture into the overall available parallelism.

Despite its importance, there is no much research in this subject as could be expected and the majority of studies about instruction set architecture evaluate the distribution of instruction types utilization (i.e. on VAX [3] or x86 [1]) rather than data utilization [6] using simulation mainly.

In contrast to the approach of simulation, the method we propose is a mathematical method: it applies graph theory to gather metrics, evaluate the impact of different characteristics of instruction sets as sources of coupling, and quantify available parallelism according to the length of dependency paths [5, 11]. To demonstrate the application of the model on a case study, we show the results of analyzing the influence of condition flags in a program code that uses the x86 instruction set.

2. THE ANALYTICAL MODEL

Modeling parallelism using dependency graphs has been a popular approach for medium and large grain parallelism, but not much in fine grain. In our approach we model instruction sequences that show data dependency as directed graphs and then use matrix representation of graphs for data processing.

*This work was partially supported by the Vicerrectorado de Investigación de la Universidad de Alcalá, 28871 Alcalá de Henares, Spain
Matrix $D$ is the Dependency matrix defined as:

$$d_{ij} = \begin{cases} 1, & \text{whenever instruction } i \text{ depends on } j; \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

In this matrix, a row is a vector $d_i$ that specifies all direct data dependencies of instruction $i$. And the set of all the rows in the matrix correspond to the sequence of a code under analysis.

As this matrix is to be used for the analysis of instruction level parallelism consistently, we define a set of restrictions and properties in $D$:

- **Labeling in the matrix corresponds to the order of appearance of instructions in the code (program order labeling)** as it is the most intuitive approach. Permutation of rows and columns doesn't affect the matrix's properties.
- **As an instruction does not depend on itself,** the matrix's diagonal is null
- **If an instruction depends on another instruction,** the latter cannot depend on the former; this asymmetry implies that the matrix is not symmetric.
- **If we use program order labeling,** the matrix $D$ is lower triangular. In this case we have the canonical matrix $D_c$.
- **The $l$-power of matrix $D$:** $D^l$ contains all dependency paths of length $l+1$.
- **In a sequence of $N$ instructions,** $D^N$ is always null (the upper limit, meaning no parallelism at all)
- **There are no dependency cycles** as an instruction can never depend on itself in any dependency path, therefore the diagonal of any power of $D$ is also null.

A dependency path is composed of computing steps, which in turn, are the successive processing states of an ordered sequence of instructions. For example, a completely serial $n$ instruction sequence (with no parallelism at all) would correspond to a maximum of $n$ computing steps, one per instruction. Conversely, a totally independent sequence could, in theory, be processed in 1 (parallel) computing step.

Algebraically, we can obtain some useful information about the potential parallelism of the code (see [4] for formal definitions and demonstrations). These are:

- **Coupling:** a tightly coupled code shows more dependencies, so we use coupling to quantify the potential ordering of the code and find limits for it. That is:

$$C = \sum_{i=0}^{n-1} \sum_{k=0}^{n-1} d_{ik} = \sum_{i=1}^{n-1} \sum_{k=0}^{i-1} d_{ik} \quad ; \quad 0 \leq C \leq \binom{n}{2} \quad (2)$$

- **Data reuse:** or minimum data lifetime which is the longest length of the dependency paths existing from a data producer instruction and the different consumers instructions. We state this as

- **Critical Path Length $L$:** the longest dependency path of the instruction sequence “L” can be obtained when $D^L=0$. It is bounded by 1 and $n$ computing steps.

- **Parallelism degree:** we identify the number of instructions that can potentially be executed in parallel as an expression derived from $L$: $G_p = n/L$. Its bounds are also 1 and $n$.

$$t_{max} = \max_{k,j \in \mathbb{N}} \left\{ k_j : \left[ D^k \right]_{ij} \neq 0, \left[ D^{k+1} \right]_{ij} = 0 \right\} \quad (3)$$

A very important property of the model proposed is the fact that matrix $D$ can be the resulting aggregation of different dependency sources if we represent each of these sources as a dependency matrix itself. So, we have:

$$D = D_{S_1} \text{ or } D_{S_2} \text{ or } \cdots \text{ or } D_{S_k} \quad (4)$$
where $D_{sn}$ is the matrix for the nth source of dependency. And all the properties and parameters defined for $D$ apply to each component matrix. In particular the Critical Path Length of the composition is bounded by the component's longest critical path:

$$\max_i \{L_i\} \leq L \leq \min \left\{ \sum_i L_i, n \right\}. \quad (5)$$

3. APPLYING THE MODEL ON THE X86 SET

One of the reasons why we have selected the x86 architecture is because its behavior in superscalar environments: it performs very poorly compared to non-x86 sets. The IPC (Instructions per cycle) is 0.5 to 3.5 in different x86 execution models [2, 7, 9]; compared to an IPC of 2.5 to 15 (and peaks of 30) of non-x86 processors [8, 12, 13]. That makes us think that the architecture of the instruction set is a limiting factor in the available parallelism. Another reason for selecting x86 is its current heavy utilization. It has been maintained across time for binary compatibility reasons, in despite of the fact that it was designed long ago and the criteria used then is now obsolete: it was conceived to decrease the semantic gap between high level languages and the assembler, and to minimize program code size.

In order to perform the analysis, we must define the different sources of data dependency we want to use in our study. The first criteria to use for a classification is the dependency type, namely:

- True dependencies (Read After Write)
- Anti-dependencies (Write after Read) and
- Output dependencies (Write after Write)

We further refine the classification by specializing upon operand types:

- Explicit operands, appearing in the instruction's code
- Implicit operands, involved in the computation but not present in the instruction's code

Another category we are interested on is the function (use) of the operand causing the dependency:

- for data processing
- for address computation
- for stack traffic
- condition flags

This results on 24 components corresponding to 24 different sources of data dependencies to be used in the analysis of a certain execution trace.

As testbench programs (see [10]), we use the execution trace of popular programs like GO form the SPECint95 suite (http://www.spec.org/cpu95). Aided by a software tool we built which disassembles the traces, finds dependencies and performs the necessary matrix transformations, we obtain the matrix of data dependencies for windows of 512 instructions at a time (configurable). The we get mean values and then gather the metrics explained in the previous section: matrix $D$, coupling $C$, critical path length $L$ and degree of parallelism $G_p$.

In order to quantify the impact of each different source of dependency, we apply the aggregation property, and obtain the results shown in Figure 1: the number of computing steps (critical path $L$) for each and every one of the 24 components and its aggregation which corresponds to the rightmost column labeled “D”.

For the purpose of demonstration, we are going to follow the process to estimate the influence of a particular operand function that is one of the most typical sources of dependency in the x86 set: condition flags.
We first obtain the Length of the Critical Path in the full aggregation $D$, obtaining $L = 307$.

If we do the composition again excluding the contribution from condition flags, the value for $L$ in the reduced aggregation is $L_{cf} = 273.2$.

The difference of 33.8 steps is the increment in $L$ due to condition flags. In other words:

- this feature is responsible of a 12.4% of increment in code serialization.

Now, let's analyze the impact of condition flags in another sub-aggregation: true dependencies. The critical path in the true dependencies aggregation is $L_T = 94.3$ computing steps. Again if we recompute the critical path in this composition excluding the contribution from condition flags, we find $L_{T,cf} = 92.9$ computing steps.

- The small difference means that condition flags do not contribute much as true dependencies in the serialization of the code.

If we only consider dependencies generated by condition flags we obtain the data summarized in Table 1, where the implicit/explicit category is also provided.

The contribution of condition flags to the length of the Critical Path is 197 computing steps and it is basically concentrated in the output dependencies component, with 146 computing steps. This reflects the typical trace of dependencies generated by condition flags in a block of code: state information is updated continuously upon execution of process type of instructions (write after write) and it is only once read upon evaluation of a jump condition.

Table 1. Mean value of computing steps caused by condition flags for different types of dependency sources.

<table>
<thead>
<tr>
<th>Sources of data dependencies for condition flags</th>
<th>Computing steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit True dependencies</td>
<td>2</td>
</tr>
<tr>
<td>Implicit True dependencies</td>
<td>1</td>
</tr>
<tr>
<td>Explicit Anti-dependencies</td>
<td>1</td>
</tr>
<tr>
<td>Implicit Anti-dependencies</td>
<td>2</td>
</tr>
<tr>
<td>Explicit Output dependencies</td>
<td>1</td>
</tr>
<tr>
<td>Implicit Output dependencies</td>
<td>146</td>
</tr>
<tr>
<td>Total of composition for condition flags</td>
<td>197</td>
</tr>
</tbody>
</table>

146 Computing steps is also a lower limit for $L$, as the critical path length of the composition $L$ is bounded by the component's critical paths as stated in mathematical relation (5).

Finally, and in aim of completeness, when we analyze the sub-aggregation of non-true dependencies (anti-dependencies plus output dependencies) and compare its critical path length $L_{NT} = 215.56$ with the critical path length in the composition excluding condition flags, we find $L_{NT,cf} = 161.1$ computing steps.

- Now the contribution from condition flags is considerable, being responsible for an increment of 33.8% in the serialization of the code.
4. CONCLUSION

The analysis of particular features of instruction sets and the quantification of its impact in concurrent execution environments is an important aspect of the design of Instruction Sets Architecture. The analytical method we present can be applied to any system using traces of the execution of real programs, without using potentially intrusive or limiting experimentation environments as measuring using simulation or particular physical layer implementations. As this is an ongoing activity, we plan to continue applying the model to several other aspects of Instruction sets' architecture that may limit parallelism as well as generalize results obtained so far applying the model to more applications and adapt the tools to analyze other processors too.

REFERENCES


