Early Performance Estimation of Super Scalar Machine Models

Pradip Bose
IBM Research Division
T. J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598

Abstract

This paper presents a new methodology for estimating the performance of high-performance machine models with concurrent, pipelined execution modes. Such performance assessment is typically useful at early stages of the design cycle. Critical design decisions made with the aid of early estimators of the kind discussed in this paper, can be expected to yield near-optimal final designs.

I. Introduction

Accurate performance estimation of high-performance computers, prior to actual implementation, is a critical aspect of arriving at an optimal or near-optimal design. For modern systems, which incorporate varying degrees of concurrent execution modes, the task of a priori performance prediction is conceptually more difficult than that for serial, von Neumann machines. In the latter case, rough back-of-the-envelope calculations can often produce satisfactory results; whereas, with modes of overlap, pipelining, decoupling and multi-dispatch [1-3] introduced to get super-scalar performance, hand estimation becomes difficult and inaccurate. For true multi-processor parallel systems, the task of performance prediction through analysis or simulation becomes even more difficult in spite of crude approximations and simplifying assumptions.

Software tools like simulators and "timers" [4,5] are used in current industrial practice to predict performance. However, such tools are usually quite machine specific; in other words, once the compiler-architecture-organization (c-a-o) triple is frozen, reasonably accurate estimation is possible for given benchmark programs. Minor tuning of the above triple is possible, based on the timing estimations. Major changes usually require major rewriting of the simulator-timer and compiler codes.

In this paper, we propose a novel methodology for early estimation of performance for a class of super-scalar machines, exhibiting highly concurrent execution modes. Although we restrict ourselves to this class in this presentation, the general technique is applicable to a wide variety of other parallel processing systems. By "early" estimation, we refer to the need for getting a good cycles-per-instruction (CPI) prediction based only on a high-level (source) benchmark and basic, high-level parameters (characteristics) of the c-a-o triple. Such an early estimator should not have to depend on an existing compiler or instruction set simulator. The benefit of using such a tool would be the ability to make key early decisions about the c-a-o triple. The need for a separate set of simulator-timer tools should be largely unnecessary, since the early estimator can be designed to handle partial to full specification of the c-a-o triple.

II. Timers

A timer, in IBM terminology, is a cycle-by-cycle simulator of a candidate machine organization. Its ultimate purpose is to print out an overall cycles-per-instruction (CPI) figure for a given instruction execution trace. As a side benefit, the detailed timer outputs are useful in identifying compiler deficiencies and organizational bottlenecks. Usually, dynamic instruction traces generated by a separate instruction set simulator, form the timer inputs. The normal scenario for "timing" a large source program is therefore (Figure 1):

(a) compile the source into assembly code;
(b) assemble the above code into a binary file;
(c) run the machine code, with a given data set, through the instruction set simulator;
(d) gather the trace (with addresses) and feed it to the timer.

In order to utilize a timer of the above type to the fullest extent, one needs a working compiler, which...
implies a reasonably frozen instruction set architecture. Besides, a separate simulator for generating dynamic instruction/address traces must also exist.

III. Multi-Dispatch, Concurrent Execution Model

In this section, we introduce the notion of a generic, parameterized machine model for a class of super scalar machines, incorporating various modes of pipelining, overlap and decoupling. Figure 2 shows this generic organization at the highest functional block level. The major constituents are:

(a) A main memory module, MEM, containing instruction and data.
(b) A central instruction dispatcher, DISP, capable of accessing an instruction stream in MEM and dispatching up to d instructions every cycle to the functional units.
(c) A set of d distinct functional units, FU_1, FU_2, ..., FU_d acting effectively as concurrent co-processors, being fed by appropriate instructions from the DISP buffer.

Details of the internal organization of each block and the bus architecture, etc. are left out of this high-level description. Also, it is implicitly assumed that the FU's are mutually distinct in function. Broadly speaking, this means that the instruction set of the machine is divided into d disjoint classes, each of which is handled by a separate co-processor. This restriction is not a strict one; i.e., one may conceive of two or more co-processors catering to the same class: e.g., floating point instructions. However, for the purposes of this paper, we will adhere to our stated restriction.

Even for such a vague, generic organization, with virtually all details unspecified, it is possible to understand some of the issues and tradeoffs for achieving high performance. Let us look at a more specific example, in which d = 2, FU_1 is called FIX and FU_2 is called FLT (Figure 3). In this case, FLT handles all floating point arithmetic instructions, while FIX processes all the rest, including branches and loads/stores to/from registers. Consider now, the simple FORTRAN loop shown in Figure 4. Our problem is to estimate the number of machine cycles required to execute (all iterations of) the loop under the following assumptions:

(i) Only the following instructions need be considered: load, store, floating add, floating subtract, floating multiply. For the purposes of this example, assume that fixed point registers and operations are absent.

Indexing and index registers are absent; all addressing is direct, explicit in the instruction field.

(ii) Assume a load/store, RISC-style instruction set, with only register-to-register arithmetic operations.

(iii) Assume infinite number of (floating) registers and infinite caching for a fixed, 1-cycle load/store from/to memory.

(iv) Assume a fixed 3-stage, 3-cycle FIX pipeline and a similar 2-stage, 2-cycle FLT pipeline.

(v) Ignore the effect of branches; in other words, assume that branches are for free and that successive iterations may be initiated as soon as data dependency constraints permit.

(vi) Assume realistic values, if needed, for other unspecified parameters for getting an exact cycle count.

Figure 5 shows the symbolic "assembly" code for straightforward compiling (i.e. no fancy scheduling or optimization) of the source in Figure 4. (Since no compiler parameters were specified, this vanilla compiler is assumed). Figure 6 shows the cycle-by-cycle timing chart of the code, for the first three iterations. If we were interested in only three iterations, our required answer for execution time would be 17 cycles.

Implicit in the above timing chart is the following assumption about the instruction issue logic of DISP. Every cycle, DISP examines its buffer and looks at the next two consecutive instructions. If they belong to different classes, then both are dispatched to their respective units (FIX and FLT). If they are of the same class, only the first one is dispatched to the appropriate unit. (Limits on buffer sizes of FIX and FLT are not considered). Also, we do not worry here as to how DISP changes its fetch stream on a branch.

The question we address now is the following: how can an estimate of the real timing (as above) be inferred from direct analysis of the source program? Can a software tool for early estimation be built for use by machine architects and compiler writers? By direct analysis and experimentation, we have found that the answers are in the affirmative. The methodology used in implementing an early loop timer (ELT) is presented in the following section, as an example.

IV. "Early" Loop Timing

In this section, we consider the task of estimating the run time performance of a given, nested, high-level loop (HLL). By HLL, we mean a loop written in some high-level language, e.g. FORTRAN, Pascal, C, etc. We focus attention here on perfectly nested
[6] FORTRAN 'DO' loops. Treating the starting (c-a-o) triple as an incompletely specified, malleable model, our objective is to estimate actual machine cycles and CPI performance as accurately as possible. The detailed specification parameters defining a more frozen machine model, are to be treated precisely as that: parameters. By fixing the parameters as required, one should be able to get desired modifications to the base performance figures and timing charts.

Let us go back to our earlier example loop (Figure 4). In order to estimate performance, without actual compilation and timing, we adopt a table-driven technique for predicting the cycle time performance. In other words, parametrized formulas of performance for known HLL code segments are initially stored in a table for look-up. The associated "assembly code" segments are also stored. In naive, syntax-directed, left-to-right translation, the various code segments can be juxtaposed in a straightforward manner. The corresponding performance can be estimated quite precisely by combining the look-up cycle times using a simple linear formula. With varying degrees of code optimization and scheduling, the estimation procedure needs to be modified; independent assembly code segments must be "merge-interleaved" for best exploitation of the decoupled co-processors. We explain these terms and procedures in the rest of the section. We also outline the method of obtaining the combined cycle-time figure for a merge-interleaved code sequence.

Source statement segmentation

The intent of this procedure is to divide each assignment statement of the source program into segments, such that the corresponding "canned" (or, look-up) assembly code segments can be concatenated to generate the overall code sequence. The procedure is similar to parse tree construction prior to code generation in standard compiler technique. The rules of operator precedence defined by the language must be followed in segmenting the source statement. For the lone assignment statement in Figure 4, the segmented form would be:

\[
\{A(I)\} = \{B(I)\} \{+\} \{(I)\} \{(D)\}\] ....(a)

For each of the entities within a single pair of braces, a simple look-up op-code (depending on the architecture) can be associated. Atoms to the right of '=' correspond to LOADs and OPs, whereas the atom to the left corresponds to a STORE: for an isolated statement. A super or compound atom such as the doubly nested segment after the '+' above, ensures correct ordering of OPs in accordance with precedence rules. The algorithm for individual statement segmentation is straightforward and is omitted in this abstract for brevity. The complexity of the algorithm is clearly linear in the number of tokens (atoms).

Code template look-up, interleave, merge

The next step is to generate an overall (pseudo)-code block (per iteration) from individual, atomic code templates. This step is distinct from syntax-directed code generation from trees or dags; the method used here is merging and concatenating small linear templates, after doing the necessary data dependence analysis [6,7] and eliminating unnecessary LOADs/STOREs. For our running example, the initial step would be to apply the LOAD, STORE and OP generating functions (L, S, A, M in this case) as indicated below:

\[
L(B(I)) \ L(C(I)) \ L(D(I)) \ M \ A(S(I))\] ....(b)

This is not unlike infix-to-postfix translation, except that the code generation method is not tree-based; it is based on template matching and merging with the previously generated code string. The goal is code timing estimate, not perfect (optimal) code generation. This is because, the actual architecture itself is unknown, and we must deal with a few high-level parameters defining the (c-a-o) triple. Note that at this stage, the highest-level code generating sequence (sans the braces) is always of the form:

\[
L \ \ L \ \ O \ \ P \ \ S\] .........................(c)

If the argument of a \( L \) is a single atom, then a simple LOAD opcode is emitted. If not, the argument itself is hierarchically expanded, and the enclosing (outer) \( L \) is ignored. Thus, the timing code sequence emitted by this look-up and merge method for our example statement is as shown in Figure 5. In the majority of cases, given a basic (c-a-o) specification, it is possible to calculate an excellent cycle-time estimate based only on the "generating" code function of the form (b) shown above. In general, a full loop body, comprising of assignment statements can be converted into a single generating function for timing analysis.

For two code strings \( s_1 \) and \( s_2 \), where \( s_1 \) is the generated string (so far), and \( s_2 \) is a new string template (independently created), a new string \( s \) may be created: \( s = \text{merge}(s_1, s_2) \). The function 'merge' may, in the simplest case, return a concatenation of its arguments, in this case: \( s_1 || s_2 \). However, in general, based on data dependence constraints, the machine
organization and compiler parameters, an altogether different string may be formed. In the general case, two directed graphs, corresponding to s1 and s2, will have to be considered. Nodes in each graph are labelled by LOADs, STOREs and OPs. Arcs denote forward dependence, and arc labels give the minimum cycle-time distance separating dispatch (initiation) of the connected nodes. After merging the two graphs, conceptually the nodes and arcs have to be rearranged to reflect the new job, with global constraints. This part is equivalent to a NP-complete scheduling problem. The heuristic algorithm used in our tool gives us very satisfactory final code blocks for estimating performance. The perfect or ideal merging of two strings is a case when the merged string is a perfect interleaving of the d disjoint instruction classes. Thus, a (...LOAD, ADD, LOAD, MIP, STORE, ...) segment is an example of a perfect code sequence for d = 2 (Figure 3). Clearly, the naive code sequence of Figure 5 may have been rearranged for better interleaving and hence better timing, by using a smarter merge algorithm.

V. Experimental Results

In this section, we present one graphical result depicting the accuracy of our high-level performance estimation method. The result shown (Figure 7) is based on a doubly nested FORTRAN loop with increasing statement/operation counts. The statements are pseudo-randomly generated, with linearly increased counts of data dependencies with increasing statement counts. The loops are similar and incrementally evolved: the set of statements for one loop is a subset of its next higher one. Curve 1 shows the (infinite cache) cycle-time estimate using our early estimation tool ELIT, and curve 2 shows the actual timing using our existing elaborate toolkit. For curve 1, only high-level (c-a-o) parameters, e.g. full interleaved scheduling, RISC, 2-way FIX/FLT (3-stage/2-stage), were assumed. For curve 2, detailed information, provided by architecture and compiler documents are implied in the use of the tool complex (see Figure 1). The number of iterations for all cases was 50. The percentage error of estimation for this case stayed within a maximum of about 11.3%.

VI. Conclusion

In this summary, we have described a new methodology and software tool for early estimation of cycle-time performance for a class of high performance, super scalar machines. Our initial experimentation has demonstrated the relatively good accuracy of estimation even for randomly generated synthetic benchmarks. Such a technique and its associated tool, when perfected, will be extremely useful to architects and compiler writers alike; they will be able to make the right key decisions in defining the best organization, architecture and compiler. The analytical aspects of the methodology have largely been omitted for brevity in this summary. Details are available in a technical report [8] to be published shortly.

REFERENCES

Fig. 1. "Timing" a program: s/w tools

Fig. 2. Generic model: super scalar

DIMENSION A(100), B(100), C(100)
DO 10 I = 1, N
A(I) = B(I) + C(I) * D(I)
10 CONTINUE

Fig. 3. Pipelined, 2-way decoupled

Fig. 4. Example FORTRAN loop
1. LOAD R1, addr(B)
2. LOAD R2, addr(C)
3. MPY R2, R2, R
4. ADD R1, R1, R2

Fig. 5. Assembly code: no optimization

Fig. 6. Cycle-by-cycle timing (3 iter.)

Fig. 7. Cycle-time perf.: est. vs real