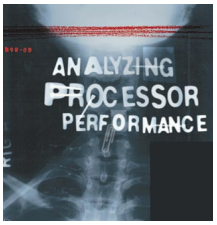
Cover Feature

Performance Simulation of an Alpha Microprocessor



If there ever was a time when the architecture of a high-performance microprocessor could spring completely formed from the mind of a single engineer, that time has passed. Modern microprocessor architectures are the result of invention and progressive refinement by a team. The team developing a future Alpha processor was guided by the performance model described here.

Matt Reilly
Digital Equipment Corp.

John Edmondson
Analog Devices Inc.

Although producing a finished microprocessor takes the effort of many engineers in many disciplines, the first step requires that an architecture team sketch out the organization of a better/faster/cheaper chip.

This effort involves searching for a solution to a design problem in a space of possible solutions. Some solutions in the space are bad, some are better, only a few are satisfactory. The goal is to find the solution that satisfies the product goals. The architecture team must invent and refine the design until it converges on an implementable architecture.

In designing Digital's Alpha processors, our teams are guided in large part by an executable performance model. The model allows us to measure the effect or utility of each invention and improvement. For example, using this model allowed us to answer the question How much faster does a benchmark program run if we double the size of a branch predictor table? The same model helped suggest areas for further development by answering questions like Why does benchmark X spend so much time in subroutine Y?

In this article, we describe the performance model that guides one of our current Alpha processor design projects.

GOALS OF MODELING

Designing and implementing a processor performance model is a substantial engineering task on its own. As with most design efforts, a successful design must reconcile conflicting requirements.

First, the model must be extremely flexible. There ought not be any intrinsic assumptions built into the modeling language or tools that inhibit searching the solution space. For example, a modeling infrastructure that does not allow us to simulate out-of-order execution would not be useful in designing a flexible, modern, high-performance microprocessor.

Second, the model must be fast. Typically we execute several different benchmark programs for each experiment, and each experiment in turn may simulate several hundred million instructions. A simulator that only executed 1,000 instructions per second would take more than a day to simulate the execution of each benchmark program. Given our successive-refinement approach, it is important to provide experiment results in hours, not days. Our goal was to maintain a simulation rate of approximately 50,000 instructions per second. In practice, we achieve about 10,000 instructions per second, so a typical benchmark run takes about three hours on a modestly configured Alpha workstation.

Third, the model must reflect reality. Caches are of finite size. Memory accesses take time. The number of ports into or out of a register file is fixed, and such ports are costly. At the start of our modeling effort, we ignore many of these reality issues to get the model up and running. As we explore the space, however, the model starts to more faithfully reflect our design choices. At the end of our search, we want the performance model to match the behavior of the register-transfer-level (RTL) model as closely as possible for key performance metrics. In this way, we can detect performance bugs in the RTL description that would not be detectable with normal verification procedures, since most design verification tests detect wrong answers and not unacceptably slow answers.

Of course, accurate models tend to be slower than more abstract models. Flexible models are slower than rigid ones. As the model becomes less flexible and more faithful to reality, exploring radical new ideas becomes harder. It is often difficult to describe new structures or policies in ways that fit into what has become a very complex piece of software.

HOW THE MODEL WORKS

Most performance-modeling experiments boil down to one question: Is processor configuration X faster than Y? We answer this question by using a benchmark program as a stimulus to the performance model and measuring the model's response.

0018-9162/98/$10.00 © 1998 IEEE

We could approach the problem by building a model that actually executed each instruction in the benchmark exactly as the target design might execute it. In this case, the performance model would faithfully represent the intended architecture down to the behavior of the functional units. On the other hand, such a model would be needlessly slow and might be very hard to modify. Writing a program that correctly interprets each instruction is also time-consuming. If this feature were included in the performance model itself, small modifications of the model might cause bugs in the interpreter. In the course of a design, we make many changes to the model: Time spent debugging the interpreter would be better spent on designing new experiments.

Atom and Aint

Rather than executing each instruction in a detailed processor model, we run an instrumented version of the benchmark program on a standard Alpha workstation, using an instrumentation tool called Atom.[1] We could alternatively use a separate instruction set emulator called Aint[2] to interpret a benchmark program. The instrumentation or the emulator transforms the instructions from the benchmark program into stimuli for the abstract performance model.

Atom was developed by Alan Eustace and Amitabh Srivastava at Digital's Western Research Laboratory. It allows an experimenter to insert *software probes* into a program compiled under the Digital Unix operating system. Probes can be inserted before or after any (or even all) instructions in a program. When executed, the probes can then pass information about the state of the processor to a trace file or another program. In our case, the probes pass information to our performance model.

Aint was written by Dirk Grunwald and Abhijit Paithankar based on work by Jack Veenstra and Robert Fowler on the MINT program.[3] MINT was an interpreter for Mips assembly language programs. Aint is an interpreter for Alpha assembly language programs. Aint maintains a record of the programmer-visible state of an ideal Alpha processor. Although Aint interprets each instruction, it does not model a processor's microarchitectural structure: It does not simulate the detailed behavior of an actual processor. Nor does Aint maintain processor states that are not visible to a programmer; such states support features like branch prediction and out-of-order execution.

The performance model, on the other hand, maintains very little of the processor's architectural state. It focuses on maintaining, in an abstract form, the nonarchitectural state necessary to support speculation and scheduling. For this reason, the combination of Aint and our abstract performance model is faster than a performance model that describes each of the processor's functional units in detail. The partitioning also allows an experimeter to make changes to the performance model without worrying that the changes might introduce bugs into the instruction interpreter.

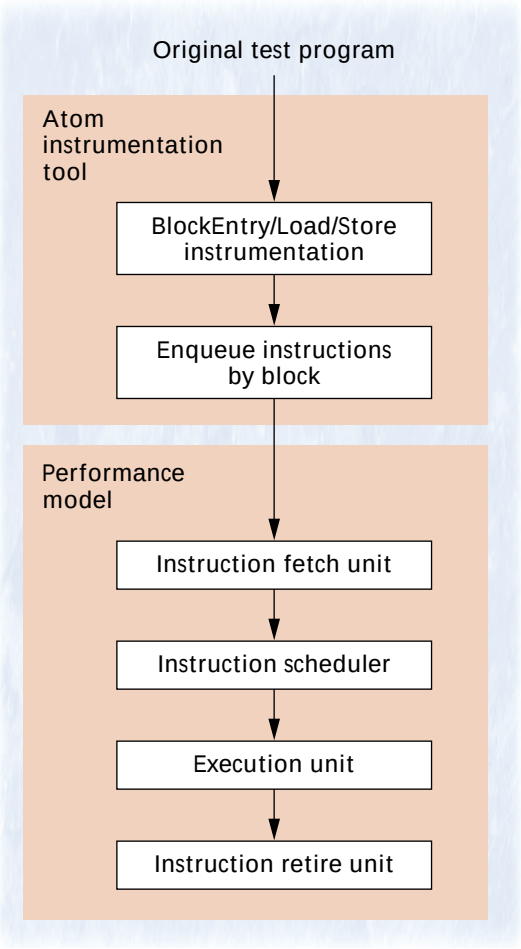


Figure 1. Flow diagram showing Atom code instrumentation and the performance model structure.

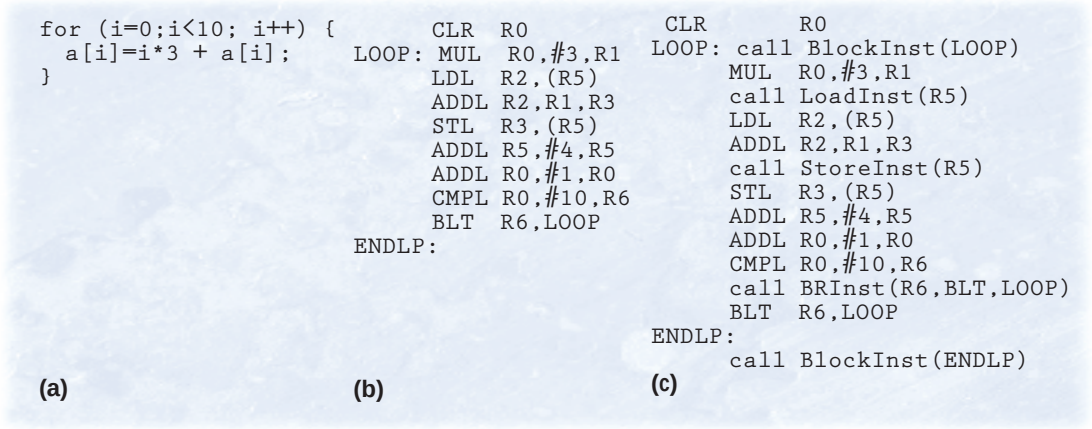
In the simplest case, we use Atom to place software probes into the benchmark program. This modifies the program so that it calls a subroutine before branch instructions and every load and store instruction. These instrumentation subroutines provide a stream of instructions and memory addresses to the performance model.

Figure 1 shows the model's top-level structure. The Atom-inserted instrumentation code collects information on the program's flow, enqueues the information in large "chunks" of a thousand or more instructions, and then passes these chunks to the performance model. A token representing each executed instruction passes through the major pipeline stages represented in the model.

Procedure

For the C fragment shown in Figure 2a, for example, the compiler might generate the instructions in Figure 2b. The Atom instrumentation software would insert calls to special routines by modifying the

Figure 2. Instrumenting a program for the performance model: (a) C fragment is transformed by the compiler into (b) assembly code. Atom software creates (c) instrumented assembly code by inserting calls to special routines that monitor program execution.



```
for (i=0;i<10; i++) {
  a[i]=i*3 + a[i];
}
```

(a)

```
       CLR  R0
LOOP: MUL   R0,#3,R1
      LDL  R2,(R5)
      ADDL R2,R1,R3
      STL  R3,(R5)
      ADDL R5,#4,R5
      ADDL R0,#1,R0
      CMPL R0,#10,R6
      BLT  R6,LOOP
ENDLP:
```

(b)

```
      CLR       R0
LOOP: call BlockInst(LOOP)
      MUL  R0,#3,R1
      call LoadInst(R5)
      LDL  R2,(R5)
      ADDL R2,R1,R3
      call StoreInst(R5)
      STL  R3,(R5)
      ADDL R5,#4,R5
      ADDL R0,#1,R0
      CMPL R0,#10,R6
      call BRInst(R6,BLT,LOOP)
      BLT  R6,LOOP
ENDLP:
      call BlockInst(ENDLP)
```

(c)

instruction sequence, as shown in Figure 2c. The special routines notify the performance model of each major event in the benchmark program's execution. When the program enters a new basic block, the BlockInst subroutine informs the model of the address associated with the start of the new block of instructions. For load and store instructions, the LoadInst and StoreInst routines inform the model of the location of the instruction and virtual address that will be loaded from or stored to. As each branch executes, the BRInst routine determines the branch's target address and whether the branch is taken. BRInst then sends this information to the model.

With information from the four instrumentation routines and an array containing all the instructions in the original benchmark program, the EnqueueInstructions section can generate all the relevant stimuli for the rest of the model. Instrumenting each load and store instruction tells the model which virtual address will be accessed. This allows accurate modeling of cache and main-memory behavior without having to actually interpret the instruction stream inside the performance model.

For example, executing the load instruction in a loop passes the effective source address and the instruction to the performance model's input queue. When the instruction reaches the head of the queue, the model sends it to the instruction fetch unit first.

If the fetch unit is not waiting to service an instruction cache miss or to correct a branch mispredict, the load instruction is passed to the scheduler. Otherwise, the model notes that the instruction stalls in the pipeline for the amount of time required to service the miss or to resolve the branch. Once the instruction arrives at the scheduler, it waits until its issue conditions have been satisfied before advancing to the execution unit. (An instruction does not go to an execution unit until the scheduler determines that all of its input operands will be available in the execution pipeline.)

The model also records this wait time. Once the load instruction passes to the execution stage, its effective address is sent to the memory and cache modeling routines. The model records the amount of time the instruction spends in each stage of the memory access pipeline, accounting for data cache misses, memory access time, and other effects. Finally, the instruction passes to the retirement unit. At each stage of execution, the model records the time spent by the instruction at that stage and the resources the instruction consumes.

Handling speculative instructions

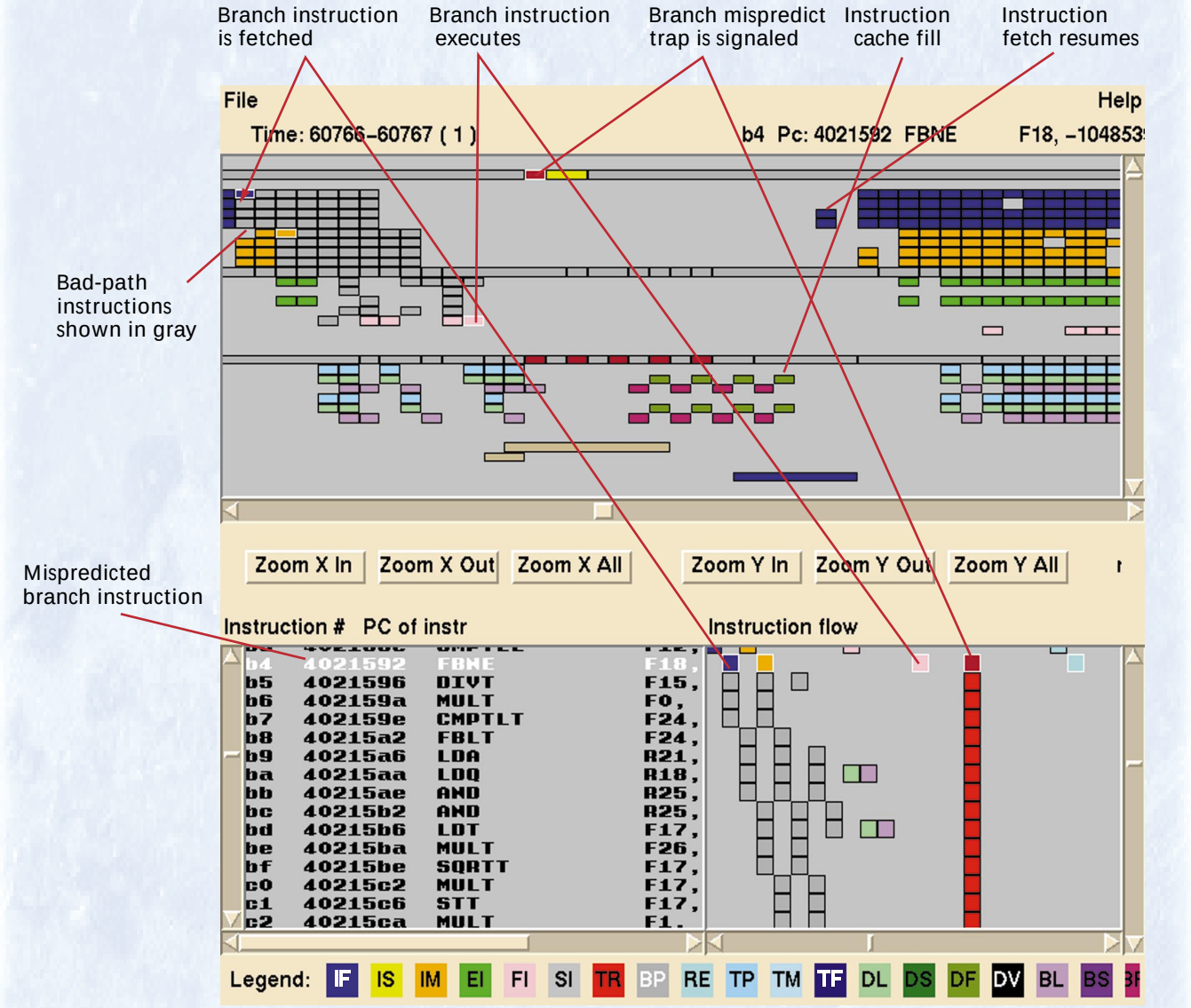
The Atom instrumentation scheme only informs the model of instructions that are actually executed. Modern high-performance microprocessors frequently "execute" instructions on speculation. That is, when the processor encounters a conditional branch in a program, it makes an informed guess as to whether the branch will be taken or not taken. It then executes the instructions along this speculative path. Sometimes the guess is wrong, in which case the processor will roll back the machine's state to what it was just before the misprediction and restart along the correct path. In the meantime, however, resources are consumed by instructions along the incorrectly chosen path. We say that the instructions fetched as the result of a mispredicted branch are along a bad path.

In our performance model's initial version, the instruction fetch unit would stall any time it was informed of an unexpected block. (That is, it would stall when the branch predictor guessed incorrectly.) The instruction fetch unit would delay all further instructions until the execution unit had encountered the mispredicted branch and resolved it.

This, of course, ignores the fact that instructions from the bad path consume resources. In an out-of-order-issue machine, they can even delay the resolution (execution) of the mispredicted branch that caused the bad instructions to be fetched.

If we had ignored this effect, we might have settled on an inadequate solution. Instead we use a second mechanism for introducing stimuli to the performance model: an instruction interpreter for Alpha code called Aint. Modeling experiments do not require modifications to Aint, so new experiments should not introduce bugs into the interpretation software.

In Aint mode, the part of the performance model that describes the instruction fetch unit directs Aint to interpret basic blocks of instructions. The Aint model then stimulates the rest of the model using information it acquired while interpreting each instruction in a basic block. Instructions provided by



Aint follow the same path through the pipeline that we described earlier. Using Aint, the branch predictor in the performance model can create speculative instruction streams that are used to stimulate the model. By interpreting the speculative instructions in Aint, the model has access to the memory addresses and processor state that bad-path instructions might generate. Aint can then undo the effects of these bad-path instructions when a mispredicted branch is finally resolved.

The end goal of the architectural development effort is not, however, a model. The end goal is the definition of a microarchitecture. What information does the team extract from the model to advance this effort?

WHAT THE MODEL TELLS US

At the conclusion of each benchmark run, the model reports

- the number of cycles required to execute the simulated instructions and
- the average number of instructions executed per machine cycle.

Of course, these numbers provide no insight into why design A is better than B. For this reason, the model also reports a dazzling variety of statistics on functional-unit usage, time spent in each major pipeline stage for each instruction type, frequently executed code sequences, branch prediction and other prediction accuracy reports, lists of hard-to-predict or problematic instructions, and even statistics on the performance of the model itself.

At times, we use the model to produce traces of machine activity on a cycle-by-cycle basis. Most often, the model presents these traces in a textual form as a log of events. We also have a graphical presentation tool that draws "waterfall" or pipeline diagrams showing the path of each instruction through the pipeline stages over time. Figure 3 shows a diagram of pipeline activity for Digital's 21264 Alpha proces-

Figure 3. Graphical presentation of pipeline usage and instruction paths.

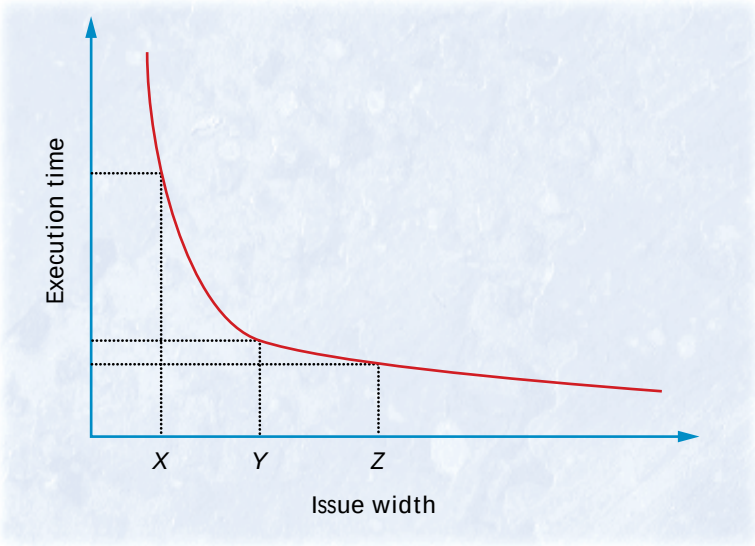


Figure 4. Execution time versus issue width.

sor. It was produced by a tool similar to the model we are now using to develop the next-generation Alpha processor.

The model also supports a *debug trace* mode. In this mode, the model generates a very detailed, cycle-by-cycle log of activity for each major section of the processor. Users can select a level of detail at runtime by using a command line option. This feature proved extremely useful in analyzing performance problems as well as in debugging the model.

EXAMPLE EXPERIMENT

The power and flexibility of the simulator is best illustrated with a simple example. Suppose we want to determine the issue width of a new processor—the maximum number of instructions that can be sent to the processor's functional units for execution in any one cycle. Cost and complexity considerations force a compromise. While an issue width of 100 might prove blazingly fast for some applications, providing the necessary paths among 100 functional units would probably prove infeasible in current technologies. On the other hand, a small issue width might prevent an architect from taking advantage of advances in instruction fetch bandwidth.

The object of our example experiment is to find a compromise. Figure 4 shows the relationship between issue width and execution time, if other factors are equal. This graph suggests that an issue width of Y will result in a much faster processor than an issue width of X. (The optimal issue width depends strongly on the effective instruction fetch rate and on the workload characteristics.) The graph suggests that an issue width of Z does not offer much of an improvement over Y. Since Y is near the knee of the curve, the optimal choice is probably closer to Y than Z.

There are several ways to find this knee. Obviously, we could parameterize the performance model so that the issue width was programmable. We would then simulate the execution of each benchmark in our suite and build a graph like that shown in Figure 4.

This approach requires performance of one measurement experiment for each point along the issue width axis. We often used an alternative arrangement that required much less runtime. First, we set the model parameters so that the issue width is very large—for all practical purposes, infinite. Second, we assume a perfectly predicted instruction stream. (That is, we ignore the effects of branch misprediction.) Asking "What is the probability that the processor will issue N instructions in a cycle?" yields additional insight into program behavior. Figure 5 shows a histogram that answers this question. We could gather the histogram for just one benchmark or collect it over the entire benchmark set.

Figure 5 shows that for this hypothetical workload there are many cycles in which no instructions are issued. Further, during a slightly larger number of cycles, N instructions were ready to issue. While we could calculate an approximation of Figure 4 from the data in Figure 5, the histogram shows the *bimodal* characteristic of instruction-level parallelism in bold relief. The gap between the two peaks in the graph indicates that the execution units are starved for data-ready instructions. Since we are assuming "perfect" instruction stream behavior, these gaps are likely caused by long-latency operations that either the processor's instruction-scheduling technique or the compiler could not hide. Such evidence might encourage us to explore alternate algorithms for scheduling or to find other ways to ameliorate the effects of long-latency operations. In any case, Figure 5 suggests that an issue width greater than N would offer little benefit.

While part of the architecture team is performing issue width experiments, another part is refining a solution to the instruction-fetch and the instruction-stream-management problems. After integrating this work into the performance model, the team repeats the issue width experiments. The histograms generated by the new experiments will reflect the impact of nonideal control flow prediction and instruction cache behavior.

After we introduce the more complex instruction stream model, gaps in the issue histogram could result from several causes: instruction cache misses, branch misprediction, data cache misses, or inefficient scheduling decisions, for example. Untangling the causes of the wasted execution slots that these gaps indicate would be nearly impossible without the model's detailed statistical reports. For this reason, the model

is instrumented such that each instruction can record in histograms or trace files the time it spends at each stage in the execution pipeline. Further, each module or pipeline stage is instrumented to gather cumulative use and delay statistics that the model reports at the end of an experiment. In many cases, though, we relied on the model's detailed cycle-by-cycle trace records to explain experimental results.

In trace mode, the model prints information relating to each of the major pipeline stages in the processor for every cycle within a user-selectable range. The report describes the instructions at each pipeline stage, the dependencies between instructions, and the resource requirements for each instruction. From this information, we can often determine the cause of pipeline stalls, wasted issue cycles, and other performance problems.

Eventually, we use the histograms and other information to choose an issue width. After making this choice, the team models and refines the instruction scheduler design. Comparing the benchmark execution on the refined model to earlier experiments verifies that our choice resulted in a reasonable balance between complexity, cost, and performance.

HOW DID WE DO?

The issue width studies and hundreds of experiments like them supported our design decisions and tested our modeling techniques. Although we are pleased with the results of our modeling effort, the original performance model goals forced some compromises along the way.

Flexibility

Flexibility was a primary goal of the model development effort. Several choices contributed to making the model as flexible as it is.

First, we chose a code-management system that allowed many developers to work on the model at the same time. (We chose a package called CVS—Concurrent Versions System—which provided a layer on top of the standard Unix Revision Control System.) Using this system, a developer checks out a current copy of the model and modifies it. Later, these changes can be integrated back into the group's source pool.

CVS provides options that allow developers to create their own branches off the main source tree. At a later time, these branches can be reintegrated into the main development "trunk." The performance model (after more than a year of intense development involving about 20 people) includes between 50,000 and 100,000 lines of C. While not a large program compared with most industrial software engineering efforts, the combination of the program's size and its malleability often made reintegration into the main trunk a difficult task.

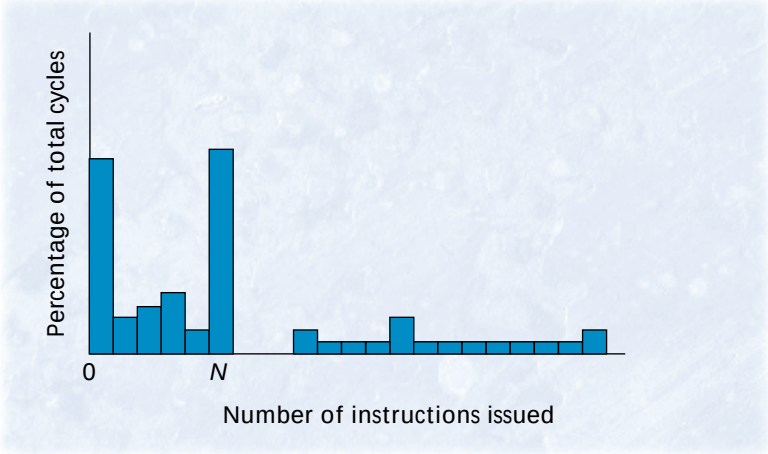


Figure 5. Instruction-level-parallelism histogram.

Second, we made extensive use of conditional compilation. When a developer inserts changes into the model, the changes are wrapped in conditional-compilation brackets that allow the compiler to ignore or include the changes at the option of later users. On the one hand, this produces some ugly code; on the other hand, it allows us to see what paths others have explored and use their earlier work. Our reliance on conditional compilation would probably spell disaster for an effort aimed at shipping a production program to outside customers. Our goals did not include shipping the performance model; it is an exploration tool. Time spent on fancy fasteners is wasted when duct tape will do the job.

We implemented our model in C. Certainly there are other languages tailored specifically for performance modeling, but C had a few hard-to-overlook advantages. For one, all of the developers on our team had extensive experience with the language. For another, C neither imposes any specific constraints on the style or mechanics of the simulation environment, nor is it biased toward any particular processor architecture. Finally, C is compatible with using Atom as the benchmark instrumentation tool.

In the end, our modeling methodology has proven sufficiently flexible to enable a wide range of experiments and a fairly sizeable team of experimenters. However, this flexibility was purchased at a cost in complexity, speed, and fragility.

Accuracy

We worked hard to accurately model the effects we thought were important. Along the way, we tested the model to ensure that its more abstract parts did not cause us to ignore important performance effects. In the process, we learned quite a bit about modeling techniques.

We implemented our model in C. Certainly there are other languages tailored specifically for performance modeling, but C had a few hard-to-overlook advantages.

For example, a useful performance model must account for speculation effects, and the Aint option has been important for modeling them. Contention between incorrectly speculated instructions and instructions along the correct path represents substantial execution overhead in some processor configurations. Further, some bad-path recovery mech- anisms may have a cost associated with them (in execution time) that is proportional to the number of mispredicted instructions in the pipeline. If our modeling methodology ignored mispredicted instructions, we would not notice such performance costs. Finally, mispredicted instructions can provide incorrect hints to various prediction schemes or confound (or even improve!) the predictors in other ways.

We also identified areas where accuracy is not important. Specifically, operations that enter one end of a pipeline segment and emerge from the other end without causing side effects need not be simulated on a cycle-by-cycle basis. Parts of the model are executed on every cycle regardless of what is happening in the processor (*cycle-by-cycle simulation*). Other parts, however, are only invoked if there is something for them to do (*event-driven simulation*).

Cycle-by-cycle simulation. In a cycle-by-cycle simulator, every part of the simulator is invoked on every cycle. Each part "wakes up," looks at its surroundings, finds out what work it needs to do, and passes its results to the next pipeline stage. This mimics our understanding of how actual hardware works and is a natural way to think about the processor. However, many units in the model wake up and find nothing to do for most machine cycles. For these units, writing in a cycle-by-cycle style—though intuitive—can produce very slow models.

Event-driven simulation. In an event-driven simulator, work items are posted to an *event queue* at the end of each simulation cycle. At the beginning of each pass through the simulator, a scheduler removes items from the front of the event queue (the queue is time-ordered) and dispatches them to the appropriate routines. If the current simulation time is cycle number 350, and the first item in the event queue occurs at cycle 400, then "simulation time" advances to cycle 400 before the simulator processes the first item. Event-driven simulators tend to be very efficient in terms of simulation versus execution time. Unfortunately, they can also be very hard to write.

Our model uses an event-driven strategy in areas where the function of the machine is extremely well understood and unlikely to change. At the start of our effort, the entire simulator was event-driven. In our efforts to more faithfully model real effects, we modeled those parts of the simulator that changed frequently or were very complex in a cycle-by-cycle style. This latter style improved modeling accuracy at the expense of simulation efficiency.

Sanity checks. Although event-driven and cycle-by-cycle simulation are important to ensuring accuracy, nothing can replace careful checking. Our confidence in the model comes, in part, from the fact that we developers question every model result and pursue any inconsistency. When the model produces a result that is wildly different from what we expect, our eyebrows go up. Developers must determine what caused the result and fix any modeling errors.

We also included assertion checkers within the model. A checker might, for instance, verify that an instruction executed after it was fetched (not before) or that an instruction that retires was not on a bad path.

As a further check, we periodically review results of well-understood test program runs and examine trace logs and graphs, searching for inconsistencies and surprises.

Speed

We started our modeling effort with a simulation speed goal of 100,000 instructions per second on our team's compute server, a four-processor AlphaServer 8400 system. The initial pass of the model ran at about 50,000 instructions per second. As the model matured and sprouted more features, the rate fell to about 10,000 to 20,000 instructions per second.

A decline in simulation speed is inevitable. Most of our developers are focusing their efforts on architecture, not high-performance programming. When the choice is between a quick answer to the current problem or coding the model for maximum simulation performance, we sacrifice simulation performance. In addition, many performance optimizations produce code that is very hard to read. While readability was never a primary goal, it often wins out over simulation performance.

We employed a few special-purpose tricks to improve model performance in cases where the effort was minimal and the return was great. In addition to the event-driven simulation scheme and careful use of Atom instrumentation hooks, we used benchmark sampling techniques. These techniques capture only the highlights of each benchmark without executing each program in its entirety.

Skip mode. For most experimental runs, we ran the simulator in *skip mode*, in which the simulator skips several hundred million instructions into the application. During this period, it keeps the cache and branch prediction tables "warm"—that is, it trains them. During this phase, the simulator processes about 700,000 instructions per second. When the skip phase ends, the simulator switches into *active mode* and

processes the next 100 million instructions. At the end of this phase, the simulator reports its information and stops. An earlier analysis of each benchmark determined how far into each benchmark the simulator should skip. This allowed us to skip over initialization phases that don't reveal much about the architecture's behavior. For instance, the compress benchmark from the SPEC95 suite spends a large part of its first two billion cycles building the data array that it will compress. Most processors will handle this task efficiently. We would rather concentrate on the part of compress that is difficult and unique to that program.

Sampling mode. Almost every designer who has used this technique has been unpleasantly surprised at one point or another when an "uninteresting" part of a benchmark was chosen as the stimulus and design decisions were based on this bad stimulus. For this reason, our simulator also has a *sampling mode* that Tom Conte and others[4,5] have suggested. In sampling mode, our simulator takes three parameters. The first is a number, *A*. The simulator will choose random-number *S* from a uniform distribution from 0 to *A*. It will then skip *S* cycles into the benchmark program. The second number, *W*, is the length of the warm-up period. After skipping *S* cycles into the program, the simulator will execute *W* cycles while training the caches and predictors, and filling the pipeline. During this warm-up phase, the entire simulator is active but collects no statistics. Finally the third parameter, *C*, defines the number of cycles we execute while collecting statistics. At the end of *C* cycles, the process repeats itself in the skip, warm-up, and collect pattern until the entire program executes.

This sampling technique gives us a statistical picture of the program's behavior over its entire span without running the whole program through the simulator. We also implemented a mode that reports the statistics of each simulation interval separately. We used this feature to choose the skip distance for day-to-day skip-mode simulations.

WHAT WE LEARNED

Perhaps the most important lesson we learned is that investing in a flexible simulation framework early in a project returns dividends as the project matures. Over the past year, we found that flexibility is not only important, but possible. This flexibility allowed many engineers to contribute to the design effort without undue conflict or management overhead.

We also found that some problems do not fit well into our basic performance-modeling scheme. As an example, when testing ideas for branch predictors, it is often useful to write a special-purpose simulator that only tracks branch instructions. Such simulators can be 50 times faster than our all-purpose model. After settling on a branch predictor organization by using such a simulator, we implemented the algorithm in the full performance model.

We found that event-driven simulation is only somewhat successful. As the model matured, the event-driven portion of the simulation time amounted to just 35 percent of the total simulation time. Even this part of the simulator is no longer purely event-driven. Expressing the behavior of a complex machine in an event-driven style can be extremely difficult.

For most experimental runs, we ran the simulator in *skip mode*, in which the simulator skips several hundred million instructions into the application.

On a few occasions "code rot" set in. (Older portions of the model either cease to behave correctly because of newer code or were made redundant by newer code.) The decay was inevitable given our reliance on conditional compilation and the existence of several parallel model development branches. Every once in a while, it was useful to go through the model, remove the rot, consolidate parallel branches of development, and delete obsolete simulation features.

Using Aint uncovered phenomena that would have been missed without modeling for speculation effects.

Graphical output tools, while useful in understanding program behavior, have not been as universally useful in debugging model problems. To be comprehensible to humans, graphical presentations must focus on a fairly narrow window in time—a few hundred cycles. Unfortunately, many of the really interesting modeling bugs manifest themselves as event sequences spanning hundreds or even thousands of cycles. For these problems, text-based log files generated by the model's debug trace facility have proven indispensable.

We are pleased with the range of experiments our performance model supports. It allowed us to conduct architectural explorations over a large range of processor organizations. In the process, we arrived at a chip organization that met our design goal of building a better/faster/cheaper microprocessor. ❖

Acknowledgments

We were not the sole or even primary contributors to the performance modeling effort described here. Michael Adler and Joel Emer designed the model's original framework. It was further developed by the architecture team working on a future-generation Alpha microprocessor, and was improved by the work of our colleagues at Digital's Cambridge Research Laboratories. We also thank John Brown, George

Chrysos, Tom Conte, Glenn Giacalone, Frank Fox, and Doug Sanders for their encouragement and extensive comments on early drafts of this article.

References

1. A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, ACM Press, New York, 1994.
2. A. Paithankar, "AINT: A Tool for Simulation of Shared-Memory Multiprocessors," master's thesis, Univ. of Colorado, Boulder, Colo., 1996.
3. J.E. Veenstra and R.J. Fowler, *MINT Tutorial and User Manual*, Tech. Report 452, CS Dept., Univ. of Rochester, Rochester, N.Y., 1993; revised August 1994.
4. K.N.P. Menezes, "An Accurate Sampling Method for Fast Processor Simulation," master's thesis, Univ. of South Carolina, 1995.
5. S. Laha, J.A. Patel, and R.K. Iyer, "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems," *IEEE Trans. Computing*, Feb. 1988, pp. 1,325-1,336.

Matt Reilly helps develop the microarchitecture for Digital Equipment's next-generation, high-performance Alpha microprocessor. He has held positions in circuit design and architecture most recently as a member of the 21264 development team. Reilly received a BSEE from Virginia Polytechnic Institute and State University, and an MSEE and a PhD in computer engineering from Carnegie Mellon University. He is a member of the IEEE.

John Edmondson is a DSP architect at Analog Devices Inc., where he develops digital signal processors. He led the architectural design of the Digital Alpha 21164 from the start of full-scale design through final product qualification. Subsequent to that, he led the development of a future high performance Alpha Risc processor. Edmondson received a BSEE from MIT. He is a member of the IEEE and the ACM.

Contact Reilly at Digital Equipment Corp., 334 South St. SHR3-1/S30, Shrewsbury, MA 01545; matthew.reilly@digital.com. Contact Edmondson at john.edmondson@analog.com.